INTRODUCTION TO JAVA

Drawbacks of C/C++

- **Platform Dependency**
- **Pointers**
- **Multi Threading**
- **Garbage collection**
- **Security**
- **Complex**

History of JAVA

James Gosling, Patrick Naughton, Chris Wrath, Ed Frank and Frank Sheridan

JAVA originated in 1992 with name Oak.

Oak was renamed to JAVA in 1995

JAVA Goals

- Simple, object-oriented, and familiar
- Robust and secure
- Architecture-neutral and portable
- High performance
- Interpreted, threaded, and dynamic

Features of JAVA

- Simple

(C++ based Syntax, no use of explicit pointers, operator overloading like rarely used features and Automatic Garbage Collection)

- Object-Oriented

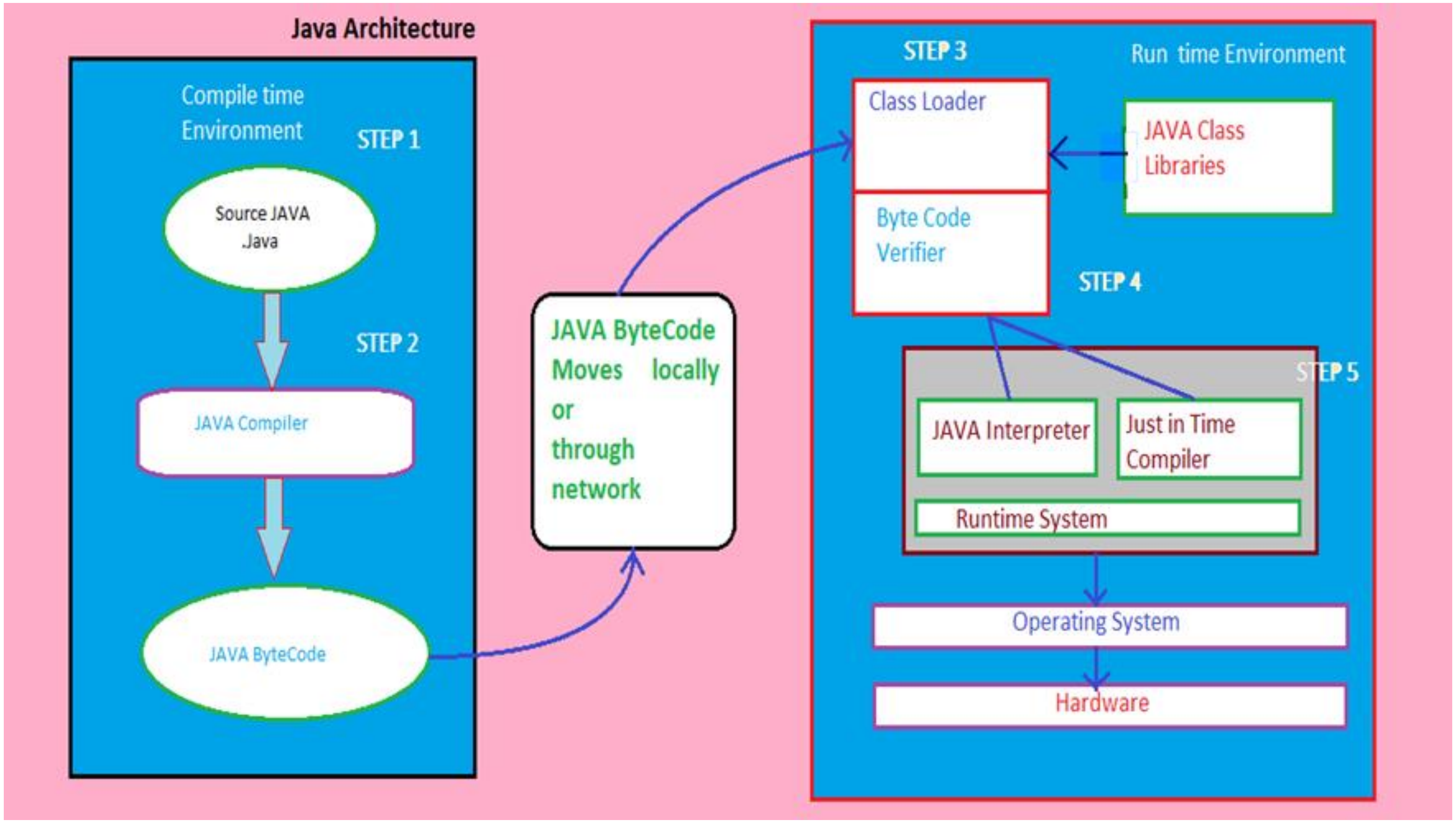
Object, Class, Inheritance, Polymorphism, Abstraction, Encapsulation

- Portable
- Platform independent
- Secured(Java Programs run inside virtual machine sandbox)
- Robust(strong memory management ,no pointers, automatic garbage collection

And automatic garbage collection)

- Architecture neutral
- Dynamic(loading of class files at runtime makes java Dynamic)
- Interpreted
- High Performance
- Multithreaded
- Distributed(features like EJB and RMI)

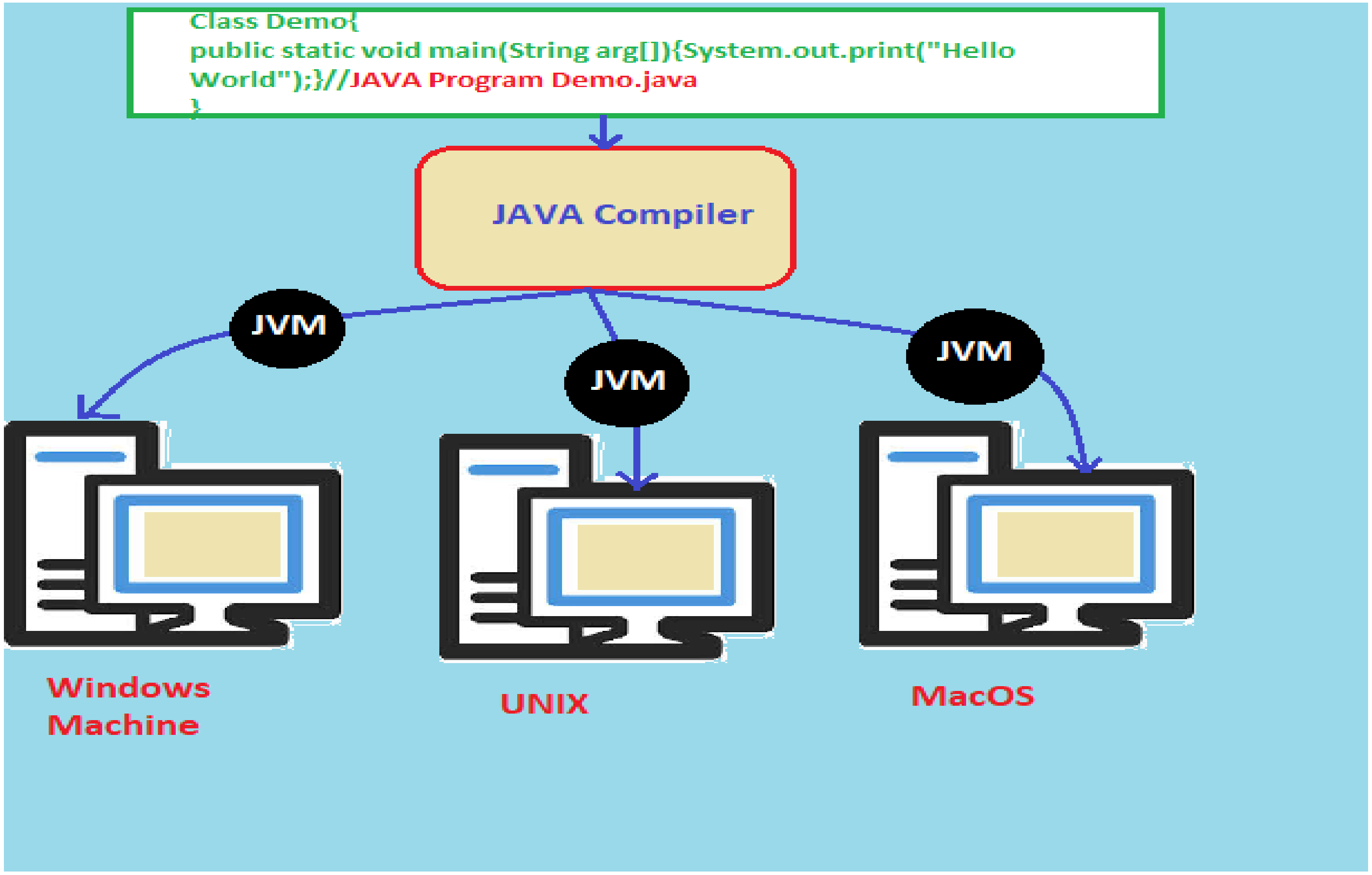
JAVA Architecture



Byte Code

- Is highly optimized, intermediate level, set of instructions and designed to be executed by the JAVA Runtime(JVM)
- Basically JVM is an interpreter for bytecode
- Implementation of JVM differ from platform to platform
- But every such JVM MUST and Does understand bytecode
- JVM inducts security to JAVA code
- Because JVM executes the bytecode which is highly optimised makes on-the-fly execution of this code

JAVA Virtual Machine



```
Class Demo{
public static void main(String arg[]){System.out.print("Hello
World");}//JAVA Program Demo.java
}
```

JAVA Architecture

Step 1:
Create java source code with .java extension

Step 2:
Compile the java source code using java compiler which will create bytecode with .class extension

Step 3:
Class loader reads both the user defined and library classes into the memory for the execution.

Step 4:
Bytecode verifier validates all the bytecode which are valid and do not violate java security regulations.

Step 5:
JVM reads the bytecode and translates into machine code for the execution. During the execution of program the code will interact to operating system and hardware.

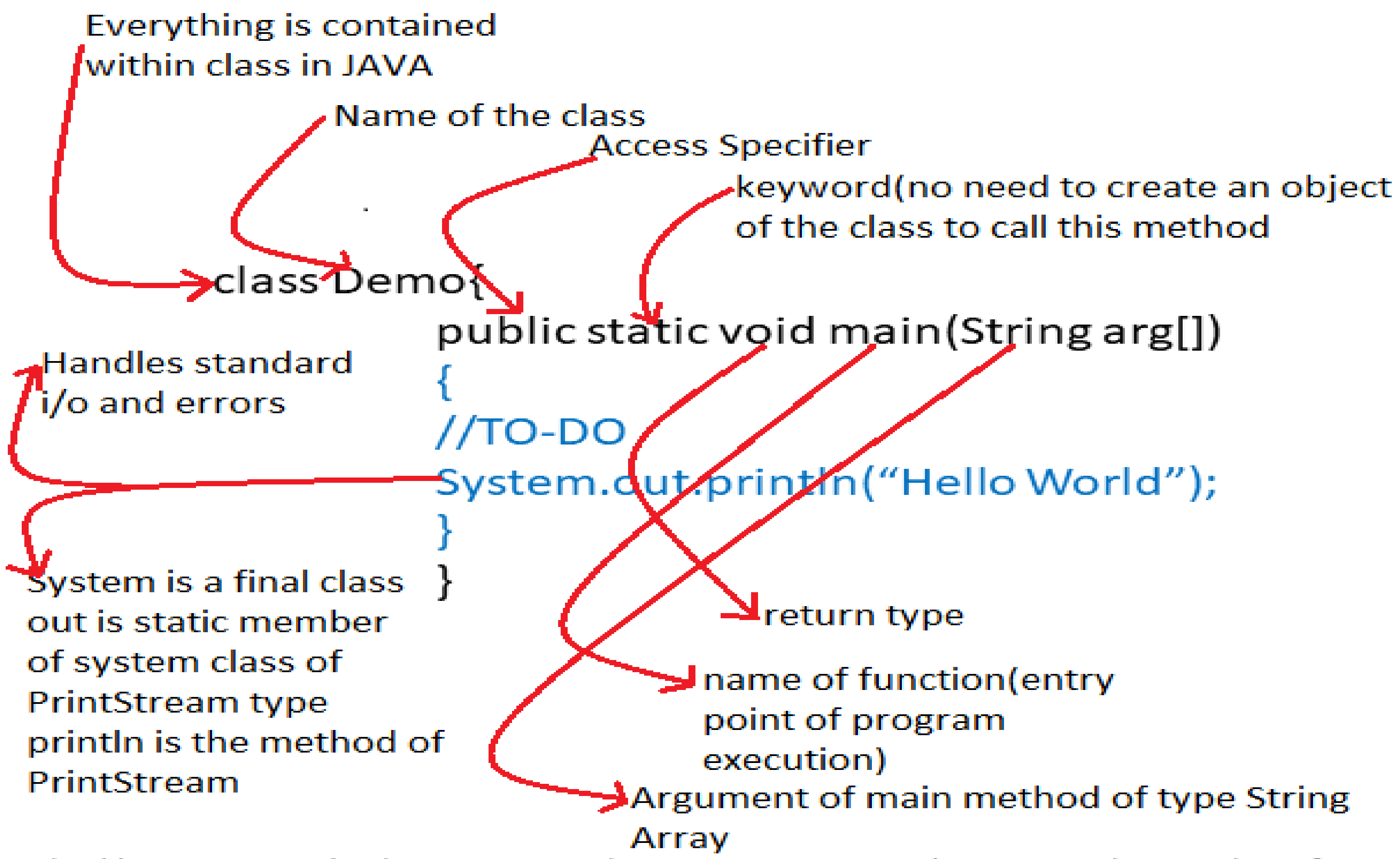
JAVA Virtual Machine

- The output of java compiler is Bytecode.
- The Bytecode are executed by JVM.
- It is an interpreter which converts the bytecode to machine specific instructions and executes.
- JVM is platform specific

- Most of the modern programming languages are designed to be Compiled.
- Compilation is one time exercise and executes faster.
- It is very difficult to execute compiled code over internet.
- Interpreting java bytecode facilitates its execution over wide variety of platforms.
- Only JVM needs to be implemented for each platform.
- If a machine comprises JVM any java program can be executed on it.
- JVM differs from platform to platform, and is platform specific.

- Interpreted code runs much slower.
- However bytecode enables to run java programs much faster.
- JAVA supports on-the-fly execution of bytecode into native code.

First JAVA Program



```
class Demo{
public static void main(String arg[])
{
//TO-DO
System.out.println("Hello World");
}
}
```

Everything is contained within class in JAVA

Name of the class

Access Specifier

keyword(no need to create an object of the class to call this method

Handles standard i/o and errors

System is a final class
out is static member of system class of PrintStream type
println is the method of PrintStream

return type

name of function(entry point of program execution)

Argument of main method of type String Array

The (.) operator is also known as member operator it is used to access the member of a package or a class.

Compile and execute

```
>javac Demo.java                    //to compile

>java Demo                          //to execute program
```

JAVA Keywords(can’t be used as identifiers)

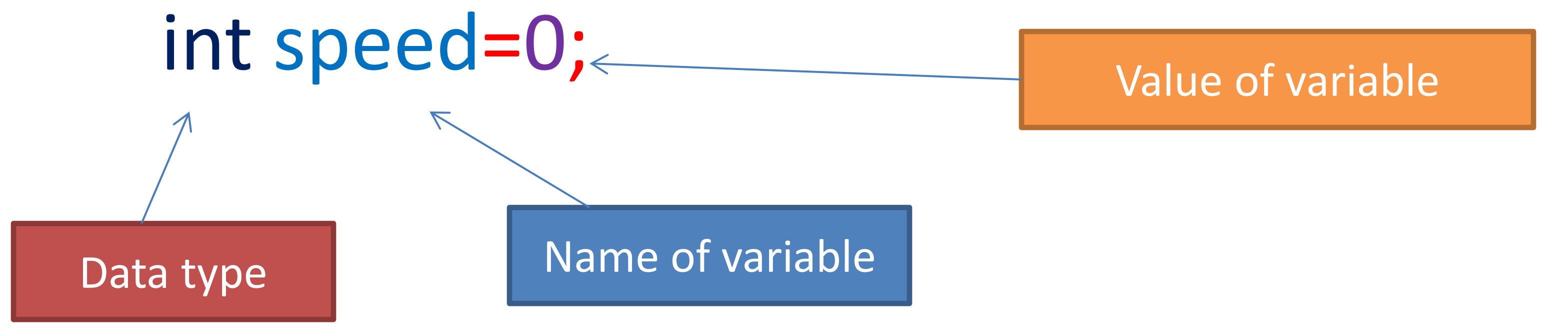
abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

*	not used
**	added in 1.2
***	added in 1.4
****	added in 5.0

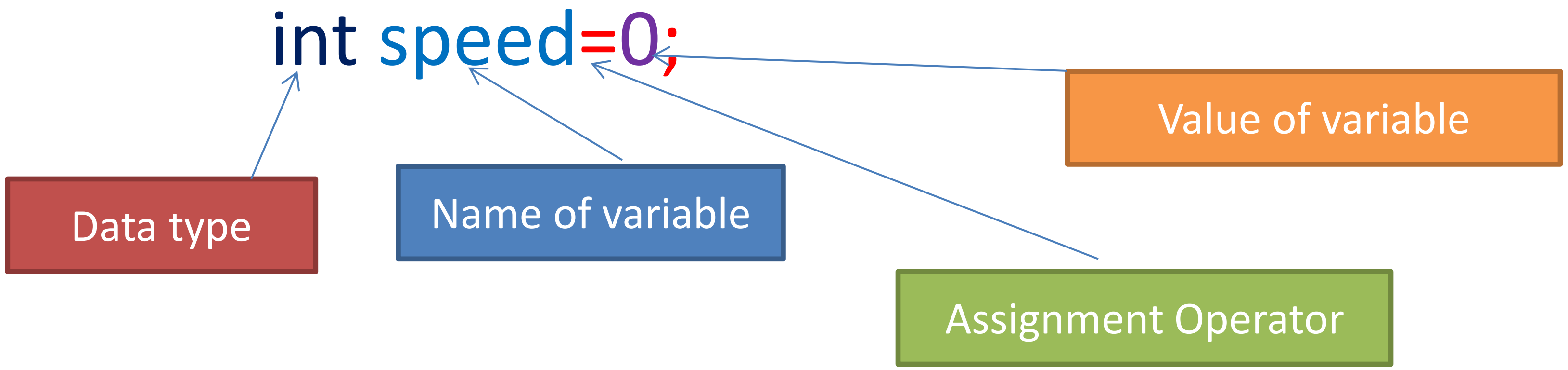
Variables

A variable is something which can hold some value, or this is like a container for holding some value; and off course this value is always subject to change. The name of variable is always preceded with the name of data type in JAVA.

For example



```
int speed=0;
```



```
int speed=0;
```

Value of variable
Data type
Name of variable
Assignment Operator

Type of variables in JAVA

The Java programming language defines the following kinds of variables:

- **Instance Variables (Non-Static Fields)**
- **Class Variables (Static Fields)**
- **Local Variables**
- **Parameters**

Instance variable/Field/ object Variable

Class/static variable

Parameters

Local variable

```
public class VariableDemo
{

int var1=0;                    //Each Instance  of this class will have
                               //its own copy of this variable
static int var2=6;                     //Only one(common) copy of class
                                       //variable will be
                               // available with all instances of this class
public void showArea(float r)
{
        float pi=3.144f;       // Local variable have local scope to that
                                       //particular
                               // method or code of block(example for loop;)
    System.out.println("Area of circle is = "+ pi*r*r);
}

}
```

Instance Variable

- The value of a variable is varied from object to object.
- A separate copy of instance variable will be created for every object.
- The scope of the instance variable is exactly same as of the scope of the object itself. Instance variables will be created at the time of object creation and will be destroyed at the time of object destruction.
- Instance variables cannot be accessed from static area (method or block). These are only accessible by objects reference outside the class.
- We can directly access instance members directly by name within the class.
- JVM can implicitly initialize instance variables by default values.
- Instance variables are also known as attributes, fields or properties of the class.
- Non Static variables cannot be used within a static method.
- If we are performing any changes in the values of instance variable these changes will not be reflected in others objects because each object maintains its own copy.

Static variables

- The value of a variable is NOT varied from object to object.
- Static modifier is required for creating a static variable.
- In case of instance variable(Non static), for every object a separate copy of the variable is created, but in case of static variables single copy is created at class level, and it is shared with all the class objects.
- There is no requirement of creating an object of the class to use static variable.
- All static members are accessible by class name itself.
- Static variables are created at the time of class loading and destroyed at the time of class unloading.
- Static members have same scope as of the underlying class.
- Static variables can be used with a non-static method.
- If we are performing any changes in the values of static variable these changes will be reflected in all others objects because same copy is shared among all objects.

Static Class members

- Static class members are the members of the class which do not belong to a instance of that class.
- We can access these members directly by referring to the class name

```
ClassName.staticMethod(….)
ClassName.staticVariable
```

- Static members are stored separately with class code.
- Public static final are global constants

Static methods

- Static methods can directly access static variables of the class and manipulate them.
- Can’t access non static members(instance variables or instance methods)
- Static methods can’t use this or super references.

Static Block

- Static

{

//To-Do

}

- The code within static block is executed automatically when the class is loaded by JVM Automatically.
- The order of execution of static blocks is same as the order of the occurrence.
- A class can have any number of static blocks
- JVM combines them and executes as a single block of code.
- Static methods can also be invoked within Static blocks.

(iii) Local variables :-

→ To meet temporary requirements of the programmer Sometimes we have to Create Variables inside method or Block or Constructor. Such type of variables are Called Local variables.

→ Local variables also known as Stack variables or Automatic variable or temporary variables.

→ Local variables will be Stored inside a Stack.

→ The Local variables will be Created while Executing the block in which we declared it & destroyed once the Block Completed. Hence, The Scope of Local Variable is Exactly Same as the Block in which we declared it.

→ For the Local variables JVM won't provide any default values, Compulsory we should perform initialization Explicitly, before using That variable.

Eg:- ①

```
Class Test
{
 p.s.v.m(String[] args)
 {
   int x;
✓  S.o.pln("Hello");
 }
}
```

o/p:- Hello

```
Class Test
{
 p.s.v.m(String[] args)
 {
   int x;
   S.o.pln(x);
 }
}
```

C.E:-
Variable x might not have been initialized.

→ The only applicable modifier for the local variables is "final".

If we are using any other modifier we will get Compile-time Error.

Eg:-

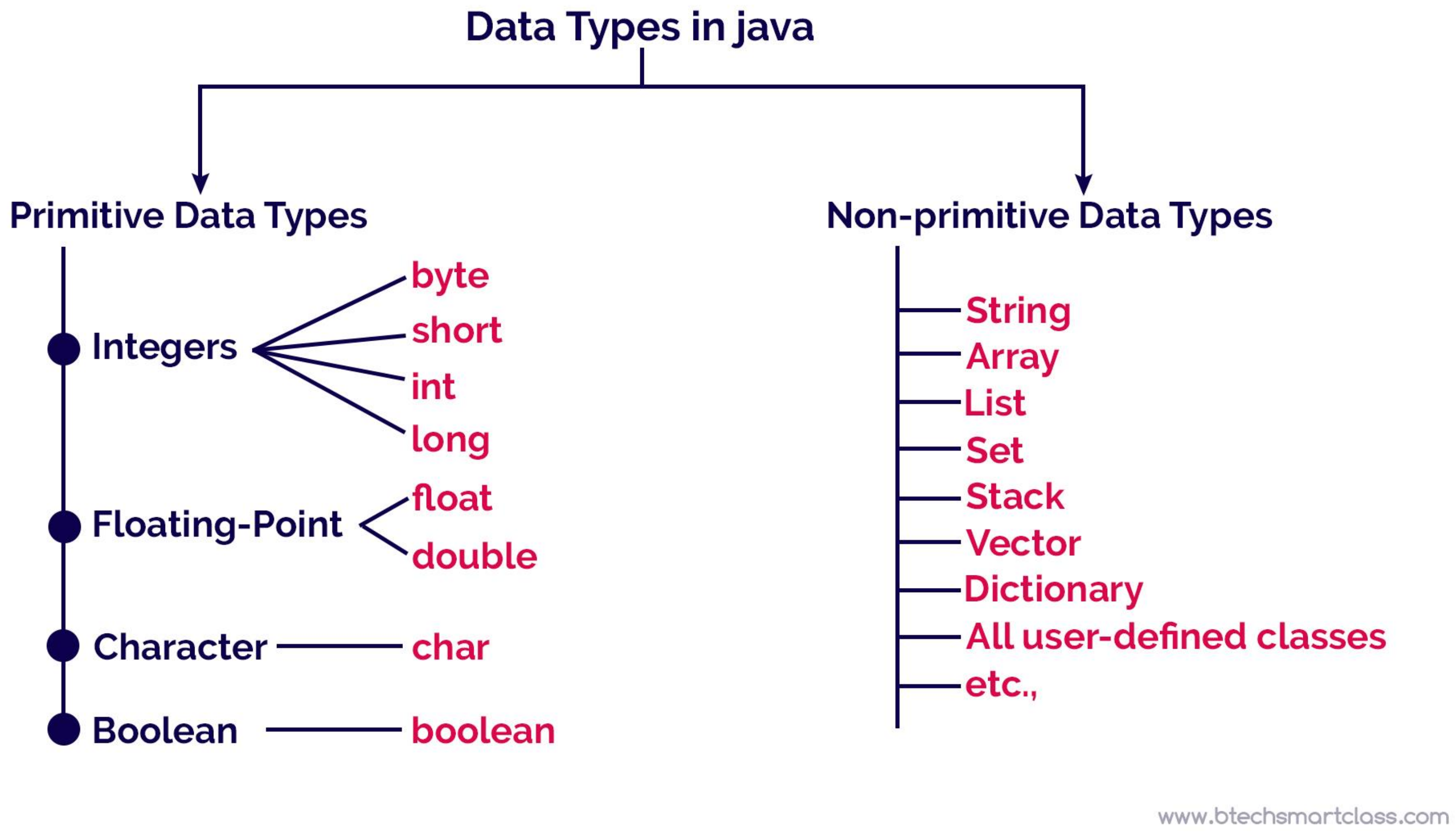
```
Class Test
{
 p.s.v.m(String[] args)
 {
✗  private   int x=10;   }
✗  public    int x=10;   }
✗  protected int x=10;   }  C.E:- Illegal Start of Expression.
✗  static    int x=10;   }
✓  final     int x=10;
 }
}
```

Java Data Types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

```
int gear = 1;
```

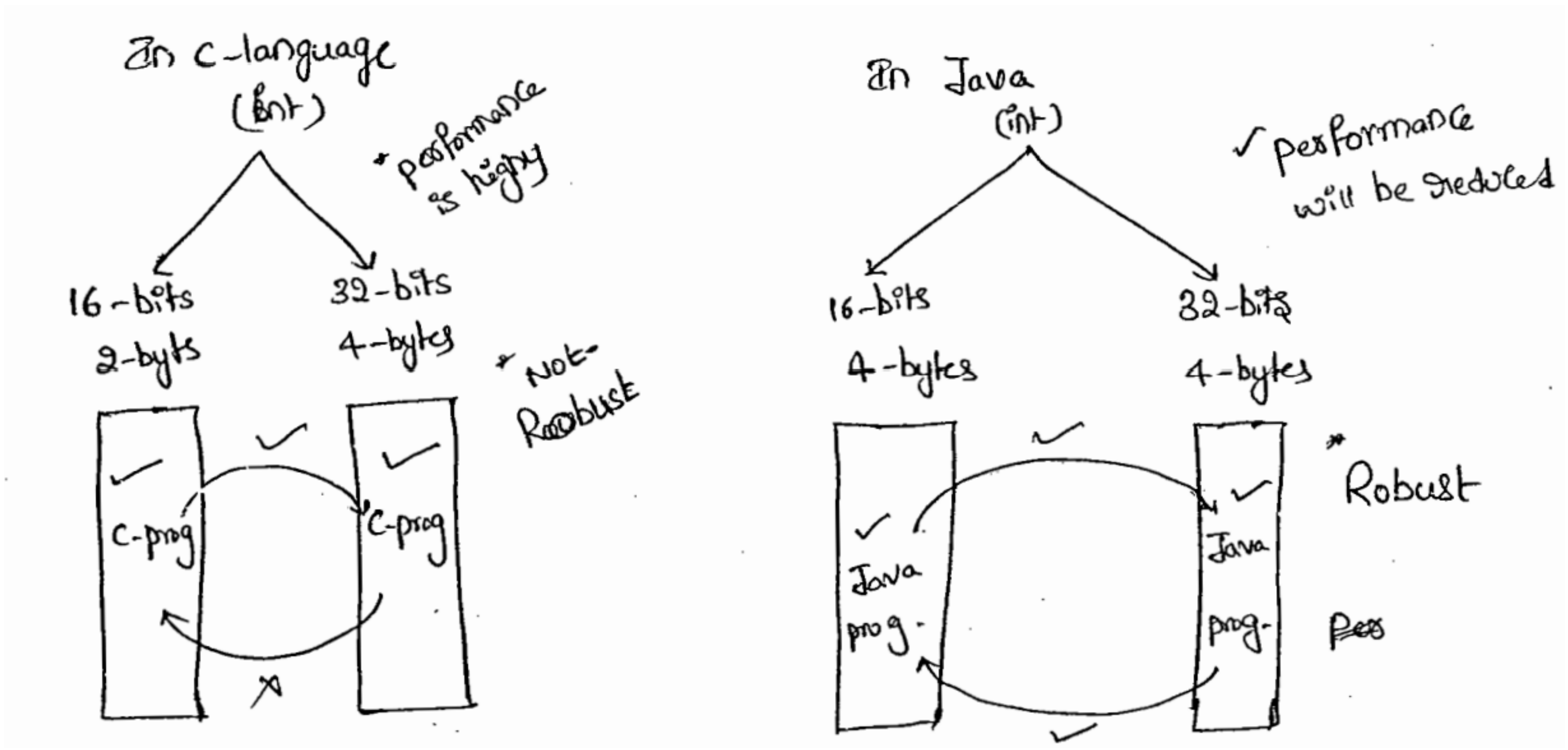
Java Data Types



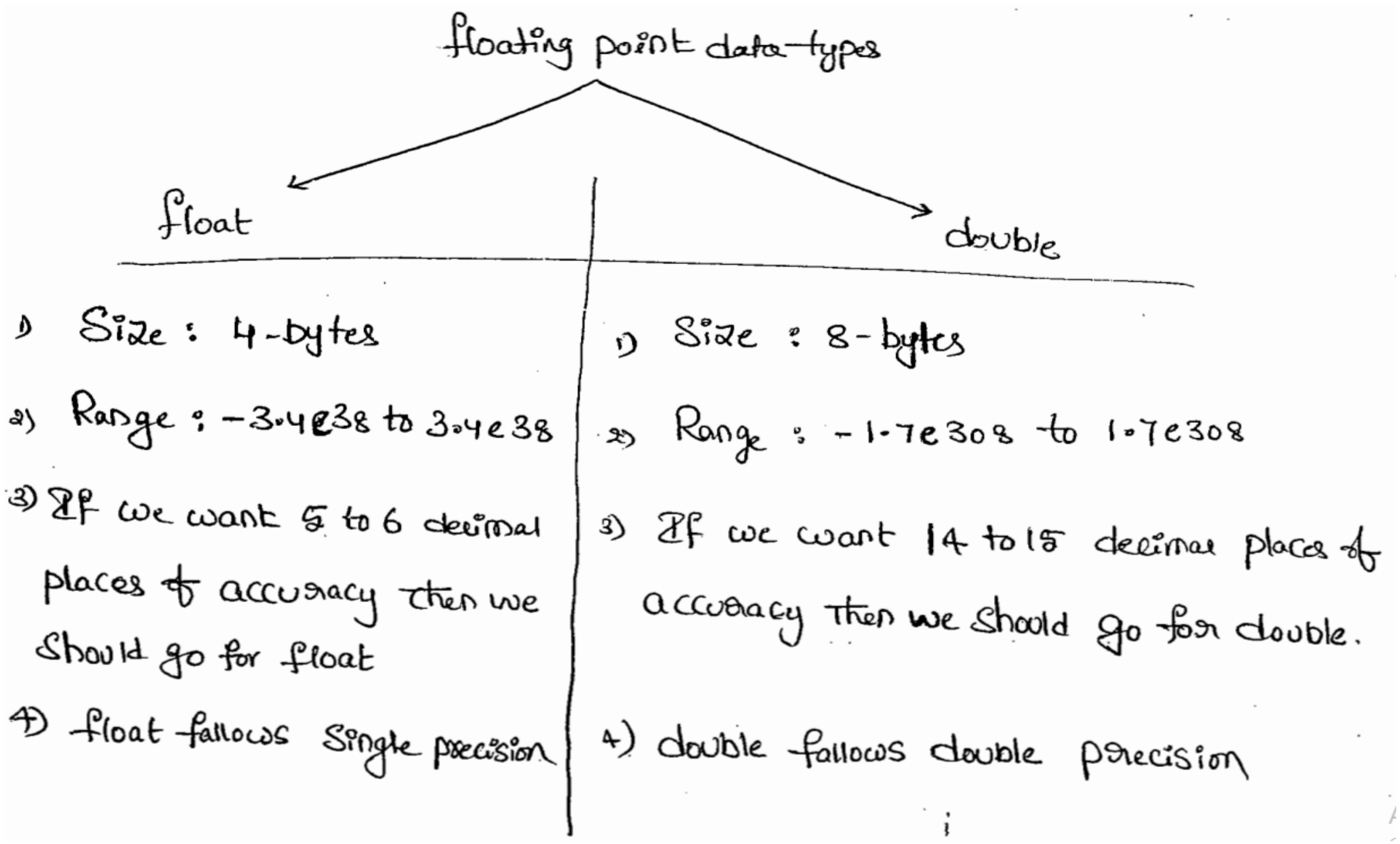
Java Data Types

Data type	Meaning	Memory size	Range	Default Value
byte	Whole numbers	1 byte	-128 to +127	0
short	Whole numbers	2 bytes	-32768 to +32767	0
int	Whole numbers	4 bytes	-2,147,483,648 to +2,147,483,647	0
long	Whole numbers	8 bytes	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0L
float	Fractional numbers	4 bytes	-	0.0f
double	Fractional numbers	8 bytes	-	0.0d
char	Single character	2 bytes	0 to 65535	\u0000
boolean	unsigned char	1 bit	0 or 1	0 (false)

Java Data Types



Java Data Types

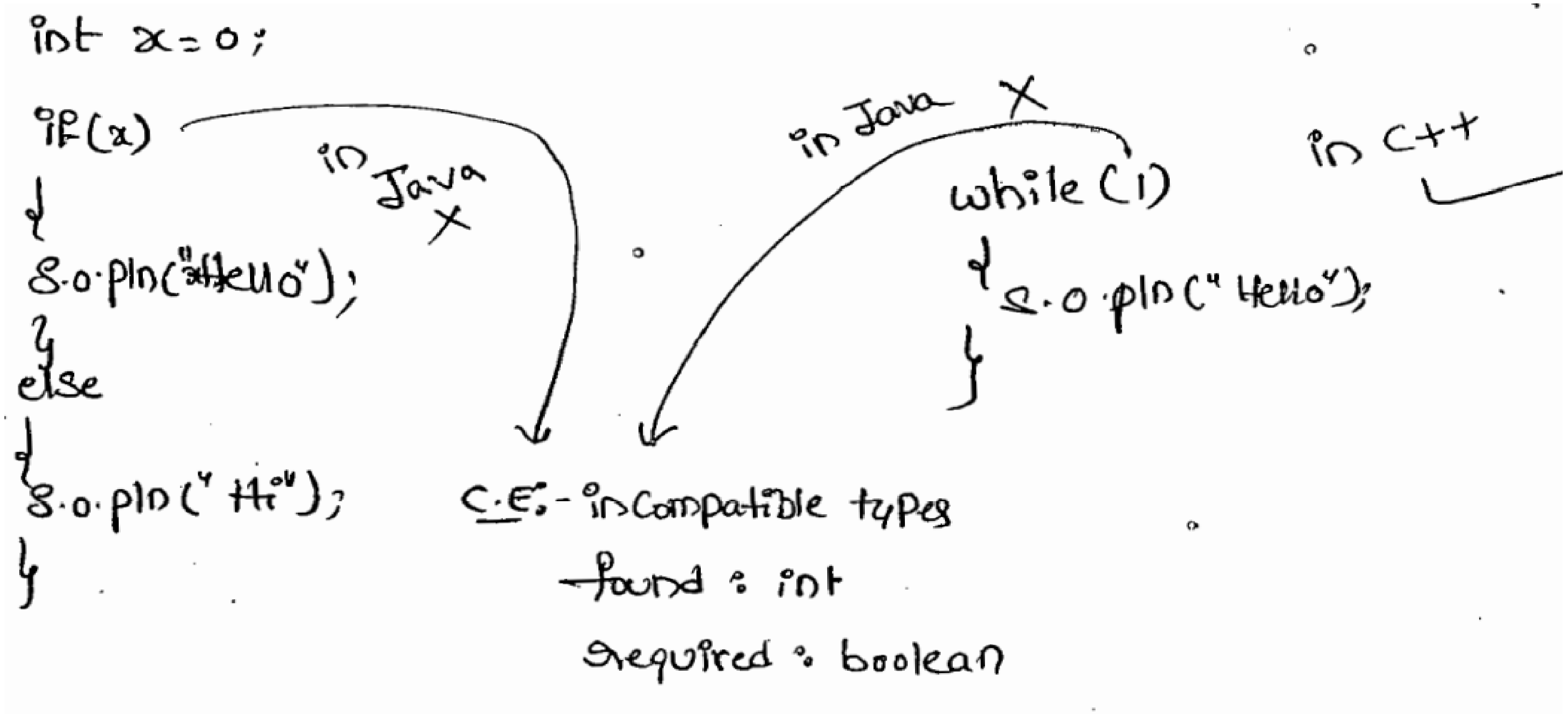


floating point data types

float	double
1) Size : 4-bytes	1) Size : 8-bytes
2) Range : -3.4e38 to 3.4e38	2) Range : -1.7e308 to 1.7e308
3) If we want 5 to 6 decimal places of accuracy then we should go for float	3) If we want 14 to 15 decimal places of accuracy then we should go for double.
4) float fallows single precision	4) double fallows double precision

Java Data Types

Boolean variables don't have any applicable range, but there are only two permitted values, **true** and **false**.



```
int x=0;
if(x)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
```

```
while (1)
{
S.o.pln("Hello");
}
```

C.E:- incompatible types
found : int
required : boolean

Literals

You may have noticed that the new keyword isn't used when initializing a variable of a primitive type. Primitive types are special data types built into the language; they are not objects created from a class. A literal is the source code representation of a fixed value; literals are represented directly in your code without requiring computation. As shown below, it's possible to assign a literal to a variable of a primitive type:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Integer Literals

An integer literal is of type long if it ends with the letter L or l; otherwise it is of type int. It is recommended that you use the upper case letter L because the lower case letter l is hard to distinguish from the digit 1.

Values of the integral types byte, short, int, and long can be created from int literals. Values of type long that exceed the range of int can be created from long literals. Integer literals can be expressed by these number systems:

Decimal, Hexadecimal, Binary

For general-purpose programming, the decimal system is likely to be the only number system you'll ever use. However, if you need to use another number system, the following example shows the correct syntax. The prefix 0x indicates hexadecimal and 0b indicates binary:

```
// The number 26, in decimal
int decVal = 26;
//  The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

Floating-Point Literals

A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double and it can optionally end with the letter D or d.

The floating point types (float and double) can also be expressed using E or e (for scientific notation), F or f (32-bit float literal) and D or d (64-bit double literal; this is the default and by convention is omitted).

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1  = 123.4f;
```

Using Underscore Characters in Numeric Literals

In Java SE 7 and later, any number of underscore characters (_) can appear anywhere between digits in a numerical literal. This feature enables you, for example. to separate groups of digits in numeric literals, which can improve the readability of your code.

For instance, if your code contains numbers with many digits, you can use an underscore character to separate digits in groups of three, similar to how you would use a punctuation mark like a comma, or a space, as a separator.

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Using Underscore Characters in Numeric Literals

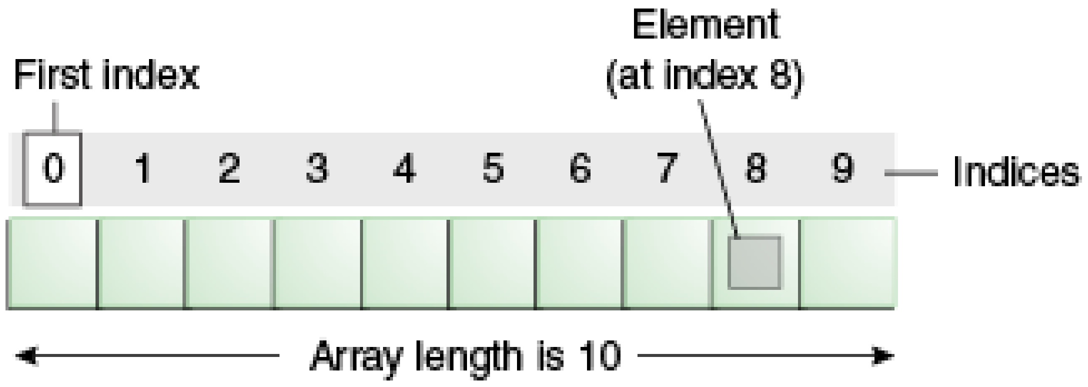
You can place underscores only between digits; you cannot place underscores in the following places:

At the beginning or end of a number
Adjacent to a decimal point in a floating point literal
Prior to an F or L suffix
In positions where a string of digits is expected
The following examples demonstrate valid and invalid underscore placements (which are highlighted) in numeric literals:

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
```

Arrays

An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.



Array:-

→ An Array is an Indexed Collection of fixed no. of homogeneous data elements.

→ The main advantage of array is we Can represent multiple values under the Same name. So, that readability of the Code improved.

→ But the main Limitation of array is Once we Created an array there is no chance of increasing / decreasing size based on our requirement. Hence memory point of view arrays Concept is not recommended to use.

→ we Can resolve this problem by using Collections.

Creating, Initializing, and Accessing an Array

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for 10 integer elements and assigns the array to the anArray variable.

```
// create an array of integers
int anArray[] = new int[10];
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

Declaring arrays

```
//One dimensional
int[] arr1;
int arr2[];
int []arr3;
//Two dimensional
int[][] arr4;
int arr5[][];
int [][]arr6;
int[] arr7[];
int[] []arr8;
int []arr9[];
```

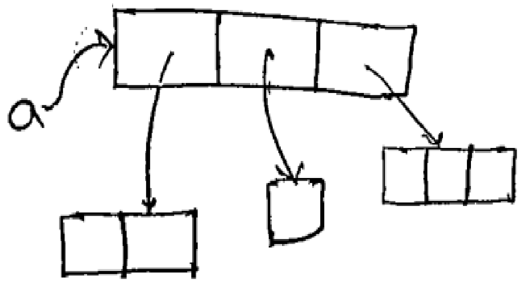
Creation of 2D-Arrays:-

→ In java multi dimensional arrays are not implemented in matrix form. They implemented by using Array of Array Concept.

→ The main advantage of this approach is memory utilization will be improved.

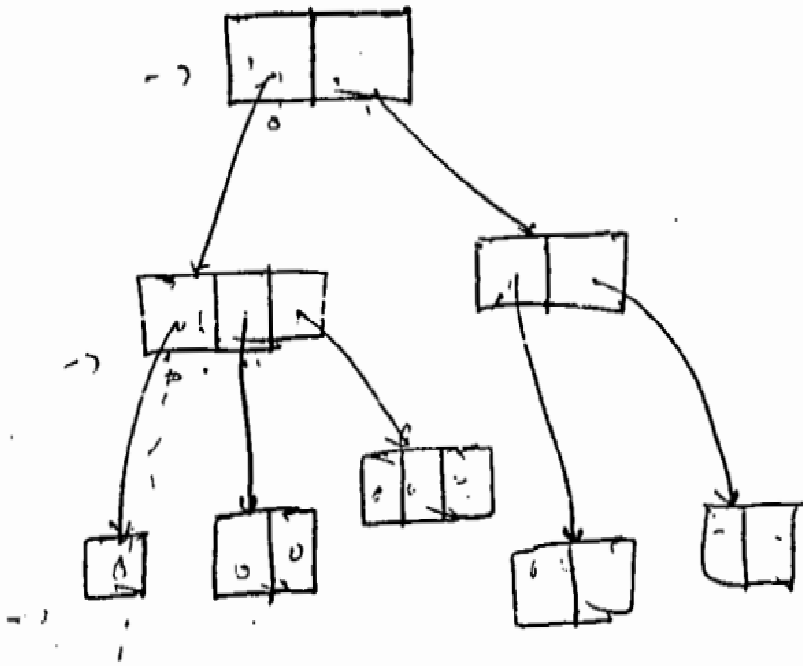
Ex:-

```
int[][] a = new int[3][];

 a[0] = new int[2];
 a[1] = new int[1];
 a[2] = new int[3];
```



Eg:

```
int[][][]   a = new int[2][][];

a[0] = new int[3][];
a[0][0] = new int[1];
a[0][1] = new int[2];
a[0][2] = new int[3];
a[1] = new int[2][2];
```



Q:- which of the following Array declarations are valid?

✗ ① `int[] a = new int[];`

✓ ② `int[][] a = new int[3][2];`

✓ ③ `int[][] a = new int[3][];`

✗ ④ `int[][] a = new int[][2];`

✓ ⑤ `int[][][] a = new int[3][4][5];`

✓ ⑥ `int[][][] a = new int[3][4][];`

✗ ⑦ `int[][][] a = new int[3][][5];`

Java String

Class `java.lang.String`
Class `java.lang.StringBuffer`

Characters
- "Building blocks" of non-numeric data
- 'a', '$', '4'

String
- Sequence of characters treated as single unit, May include letters, digits, etc.
- Object of class `String`
- `String name = "Mahendra";`

```java
public class StringDemo {
public static void main(String...strings) {
char charArray[] = { 'b', 'i', 'r', 't', 'h', ' ', 'd',
'a', 'y' };
byte byteArray[] = { ( byte ) 'n', ( byte ) 'e',
( byte ) 'w', ( byte ) ' ', ( byte ) 'y',
( byte ) 'e', ( byte ) 'a', ( byte ) 'r' };
String s = new String( "hello" );
// use String constructors
String s1 = new String( );
String s2 = new String( s );
String s3 = new String( charArray );
String s4 = new String( charArray, 6, 3 );
String s5 = new String( byteArray, 4, 4 );
String s6 = new String( byteArray );
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
System.out.println(s4);
System.out.println(s5);
System.out.println(s6);
}
}
```

Operators

Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result

Operators	Precedence
postfix	expr++ expr--
unary	++expr --expr +expr -expr ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Expressions

An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.

For example

int **empId = 97312**;

testArray[1] = 100;

System.out.println(**"Element 2at index 1: " + testArray[0]**);

Java programming language allows you to construct compound expressions from various smaller expressions

Like a*b*c;

However these can be ambiguous

Like a +b /100 → Ambiguous, not recommended

Or like (a+b)/100 → Not Ambiguous, recommended

Expressions must be balanced correctly using parenthesis

Statements

Statements are roughly equivalent to sentences in natural languages. A *statement* forms a complete unit of execution. The following types of expressions can be made into a statement by terminating the expression with a semicolon (;).

- Assignment expressions // speed=107;
- Any use of ++ or -- //a++;
- Method invocations //c= obj.add(15,17);
- Object creation expressions //Cal obj=new Cal();

Control Flow Statements

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling our program to *conditionally* execute particular blocks of code.

1. **decision-making statements (if-then, if-then-else, switch),**
2. **looping statements (for, while, do-while),**
3. **branching statements (break, continue, return)** supported by the Java programming language.

if

- if(condition)statement;
- The condition is Boolean expression, its outcome can be only true or false
- Relational operators like <,>,== can be used to create condition

If-else

```
if (condition)
 {
//do something here
} else
{
//do something else here
}
```

The switch Statement

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types* , the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer

Switch example

```
public class SwitchDemo {
   public static void main(String[] args) {
     int  day = 3;
      String  day;
      switch (month) {
         case 1:  day= “monday";
                break;
         case 2:  day = “tuesday";
                break;
         case 3:  day = “wednesday";
                break;
         case 4:  day = “thrusday";
                break;
         case 5:  day = “friday";
                break;
                  default: day = “holiday";
                break;
      }
      System.out.println(day);
   }
}
```

Summary :-

Case label

1. It should be Compile time Constant.
2. Expressions also allowed but should be Constant Expression
3. Value should be with the range of Switch argument type
4. Duplicates are not allowed.

(b) Iterative Statements :-

(1) while :-

→ if we don't know the no. of iterations in advance then the best Suitable loop is while loop.

Ex:-

①
```
while (rs.next())
{
  --
}
```
$\searrow$ Result Set

②
```
while (itr.hasNext())
{
  -
  -
}
```
$\searrow$ Iterator

③
```
while (e.hasMoreElements())
{
  --
  --
}
```
$\searrow$ enumeration

Syntax :-

```
while (b)
{
  Action
}
```
$\rightarrow$ boolean type

② do-while :-

→ if we want to execute loop body atleast once then we should go for do-while loop.

Syn:-

```
do
{
  Action
} while(b);
```

$\xrightarrow{}$ should be boolean type (b)

$\xrightarrow{}$ mandatory (;)

→ Curly braces are optional & without having curly braces we can take only one statement b/w do & while which should not be declarative statement.

Ex:- ①

```
do
  S.o.pln("Hello");
while(true);
```
✓

②

```
do ;
while(true);
```
✓

; is a valid java statement

③

```
do
  int x=10;
while(true);
```
✗

④

```
do
{
  int x=10;
}while(true);
```
✓

⑤

```
do
while(true);
```
✗ C.E:-

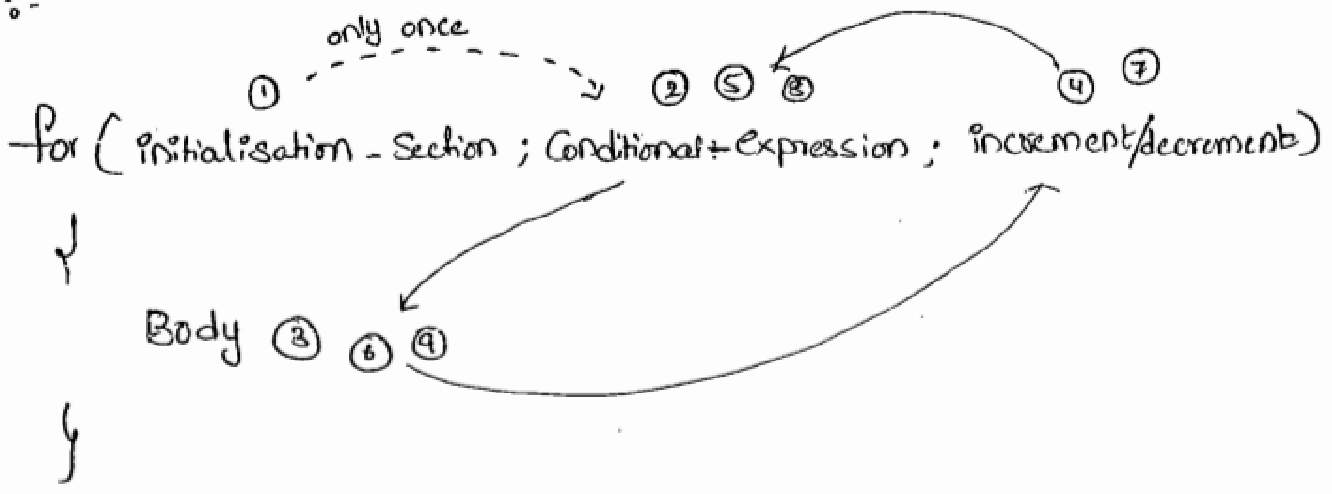
→ Compulsary one statement declare (or) take ";"

for

for() :-

→ This is the most Commonly used loop

Syntax :-



```
for ( initialisation_Section ; Conditional-expression ; increment/decrement)
{
    Body
}
```

→ Curly braces are optional & without Curly braces we can take only one Statement which should not be declarative Statement .

break statement

- We can use break statement in the following cases
- Within the switch to stop fall through
- Inside loops to the loop execution bases on some condition
- Inside labelled block execution based on some condition
- If we are using break statement anywhere else compilation error will occur.

break statement

CASE 1

```
switch(b)
{
//
break;
//
}
```

CASE 2

```
for(int i=0;i<5;i++)
{
If(i==3)
   break;
//do something
}
```

CASE 3

```
Class Test
{
   p.s.v.m(String… args)
   {
     Int i=5;
       l1:
       {
         // do something
         If(i==5)
            Break l1;

       }
     //
   }
}
```

continue statement

We can use continue statement to skip current iteration and continue for the next iteration inside loops. If we are using continue outside the loops the error will occur.

```
for(int i=0;i<10;i++)
{
        if(i%2==0)
        {
                continue;
        }
        System.out.println("i: "+i);
}
        //Output:3,5,7,9
```

Class

Class is set of objects. Class is used to classify , things having common attributes and behaviours. For example Student is class in which all people studying can be kept.

A class can have three kind of members:
Fields are the variables associated with the class and its objects and can hold the state of the object.
Methods contained all the executable code of the class and define the behaviour of the objects.
Nested classes and interfaces are declarations of classes and interfaces that occurs within the declaration of another class or interface.

Class Modifiers

public –makes class publically accessible. Without public modifier a class is only accessible within its own package.

abstract- incomplete class

final- a final class can not be sub classed.

strictfp-for strict floating point arithmetic

A class can not be final and abstract at the same time.

fields

Fields declarations can also be preceded by modifiers that control certain properties of the field.

Access specifiers- public, package(default), protected and private.

static

final

transient(related to serialized objects)

volatile(related to synchronised methods)

Access control

To support the concepts of encapsulation and data hiding there is a requirement of access control. It is achieved by access control modifiers, four possible modifiers are,

private: accessible within the same class itself.

package:(not specified) accessible within same package and same class only.

protected: accessible within the subclass of the class, within same package and within same class.

public: are available publically.

methods

A method declaration consists of two parts: the method header and method body. The method header consists of a set of modifiers, return type, signature and a throws clause listing the exceptions thrown by the method. The signature consists name of the method and list of parameters with their type separated by comma, enclosed within parenthesis.

The method can have following modifiers,

Access spcifiers,

abstract,

static,

final,

synchronised,

final,

native,

strictfp,

Abstract method cannot be static, final, synchronised, native or strict.

native modifier :-

→ native is the modifier applicable only for methods but not for variables and classes.

→ The native methods are implemented in some other languages like C & C++ hence native methods also known as "foreign methods".

→ The main objectives of native keyword are ① to improve performance of the system

① To improve performance of the System.

② To use already existing legacy non-Java code.

→ To use native keyword

Ex:-

```
                         class Native
                         {
                            static
                            {
① Load              →         ~~S.o.p~~
   native Library              System.loadLibrary("native Library")
                            }
② Declare               public native void m1();
   a native method  ↗
                         }

                         class Child
                         {
                           p.s.v.m(___)
                           {
③ Invoke a                 Native n = new Native();
   native method           n.m1();
                         } } }
```

① → for native methods implementation should be available in Some other Languages where as for abstract methods implementation should not be availabl hence abstract - native Combination is illegal Combination for methods.

② → native methods Cannot be declared with Strictfp modifier because There no guarunte that old language fallows IEEE 754 Standard.

③ → Hence abstract native - Strictfp Combination is illegal for methods.

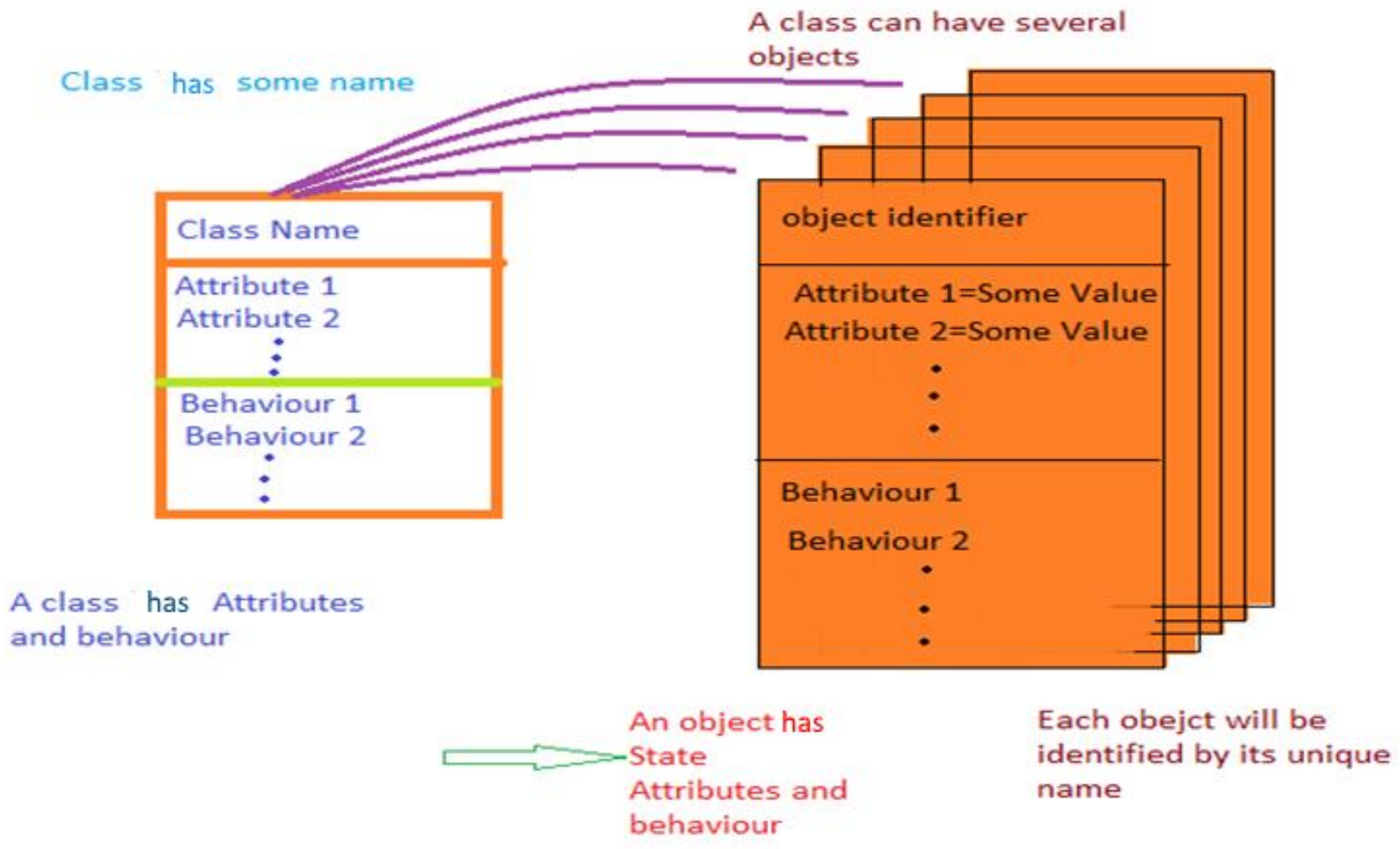
*** → The main disAdvantage of native keyword is ~~it~~ it breaks platform independent nature of Java. because we are depending on result of platform dependent languages

Object(an abstract data type)

- Attributes
- Behaviour
- Identity or State

* A language is said to object oriented if it is object based and has the power of polymorphism and Inheritance

A Detailed Picture

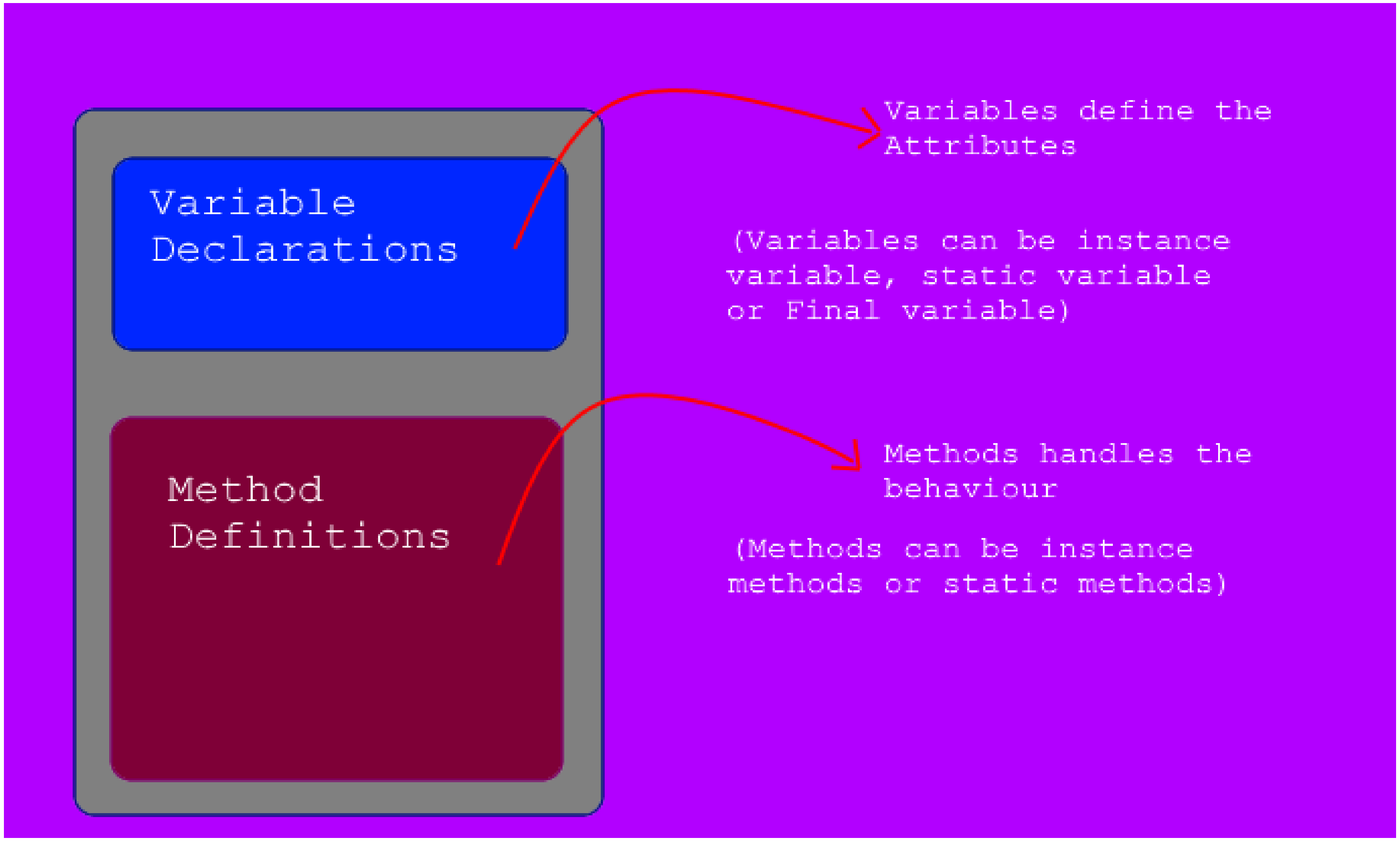

Class has some name
A class can have several objects
Class Name
Attribute 1
Attribute 2
Behaviour 1
Behaviour 2
object identifier
Attribute 1=Some Value
Attribute 2=Some Value
Behaviour 1
Behaviour 2
A class has Attributes and behaviour
An object has State Attributes and behaviour
Each obejct will be identified by its unique name

Thus a class is a set and objects are the elements of this class

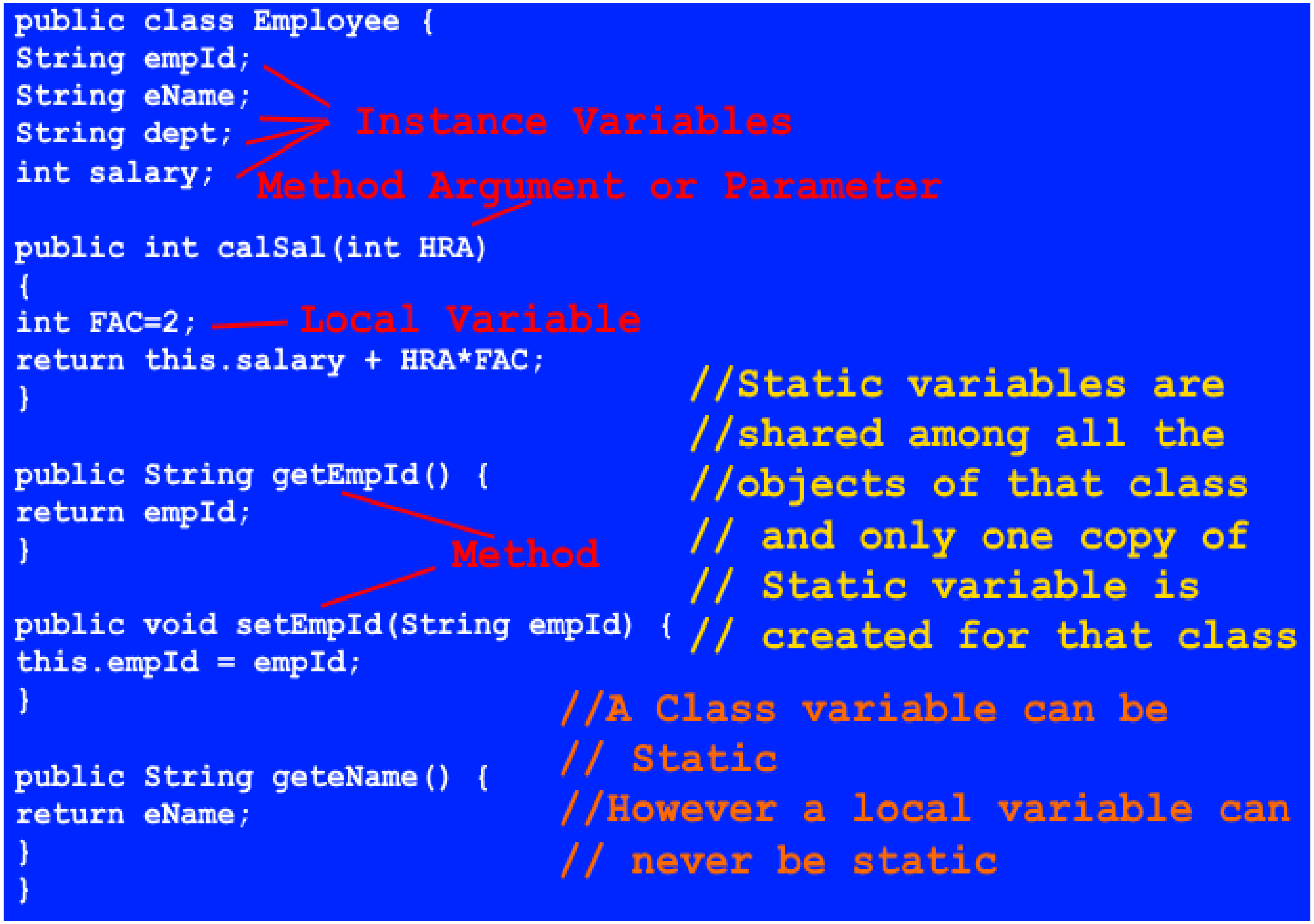
Relation between Class and Object

- Class is a logical construct
- Object is a physical reality
- Object is an instance of class

Class(JAVA)


Variable Declarations
Method Definitions
Variables define the Attributes
(Variables can be instance variable, static variable or Final variable)
Methods handles the behaviour
(Methods can be instance methods or static methods)

Class(JAVA)



```
public class Employee {
String empId;
String eName;
String dept;
int salary;

public int calSal(int HRA)
{
int FAC=2;
return this.salary + HRA*FAC;
}

public String getEmpId() {
return empId;
}

public void setEmpId(String empId) {
this.empId = empId;
}

public String geteName() {
return eName;
}
}
```

```
//Static variables are
//shared among all the
//objects of that class
// and only one copy of
// Static variable is
// created for that class
```

```
//A Class variable can be
// Static
//However a local variable can
// never be static
```

Object(JAVA)

- Objects can be declared using object reference like

```
Employee employee;
Student student;
```

- An object is created by using new keyword

```
Employee emp=new Employee();
Student std=new Student();
```

Object(Java)

- Object references are used to store the objects.
- References can be created for any class(Abstract or concrete class) or Interface.

“new” Keyword(JAVA)

- Dynamically allocates memory for the object on heap space.
- Creates object reference on the heap stack.
- Returns a reference to it.
- Reference is stored in a variable.

Constructor(JAVA)

Constructors are blocks of statements that can be used to initialize an object before the reference to the object is returned by the new keywod.

Constructor(JAVA)

- Constructor Look like a method(Or it is a special type of method)
- Have same name as of the class name
- Doesn't have any return type
- has access specifier(Public, private, protected or default)
- Constructor is automatically invoked when an object of that class is created.
- Constructors can be overloaded(means a class can have multiple constructors).
- In absence of a defined constructor default constructor(constructor of it’s super class) may be used.

Constructor(JAVA)

```
public class Employee {
        String empId;
        String eName;
        String dept;
        int salary;

public Employee(String empId, String eName, String dept, int salary) {
        super();
        this.empId = empId;
        this.eName = eName;
        this.dept = dept;
        this.salary = salary;
}
}
```

“this” (JAVA)

- Each class member function has an implicit reference to its own class object, named “this”.
- Created automatically by the compiler.
- It returns the reference of the object from which it is invoked.
- Resolves name collision between member variable and method arguments.
- Can be used to refer overloaded constructors.
- Can’t be used with static methods.

“super” keyword

- Creation and initialization of super class object is necessary for the creation of a sub class object.
- When sub class object is created, it creates a super class object first.
- It invokes the relevant super class constructor.
- Constructors of super class are never inherited by sub class.
- It is the only exception in the rule that sub class inherits all the properties of Super Class.

“super” keyword

- “super” if present it must be first statement to be executed in a constructor.
- Order of invocation of constructor follows class hierarchy.
- this(argList) refers to current class constructor.
- super(argList) refers to immediate super class
- If none of super or this constructor is inserted than compiler automatically inserts super(). Which is super class no arg constructor.

An Example to design a class

- Employee class with following attributes

Let’s take an Example(Problem)

Create a simple java Application about employees of an organisation
which excepts the id of Employee from console and displays following information.

Emp id	Emp Name	Date of joining	Designation code	Department	Basic	HRA	IT
1002	Rahul	12/5/2014	e	CSE	15800	2500	300
1003	Rajiv	25/4/2017	e	CSE	21500	2550	350
1004	Viraj	3/8/2011	f	IT	23200	2600	370
1005	Ajay	9/1/2015	c	ECE	15820	2500	320
1006	Sanjay	11/7/2013	f	Accounts	16540	2450	350

Create another class Results having main function to display results.
If a user inputs an emp id the results must be calculated and displayed.
Salary must be calculated as

Salary=Basic + HRA - IT

For example

```
Enter the ID of employee
1005
Emp id      Emp name      dept   Salary
1005         Vijay              CSE    23511
```

Encapsulation

- Encapsulation Binds code and data together
- Encapsulation keeps both safe from outside interferences and misuse
- The basis of encapsulation in java is “CLASS”
- “OBJECTS” are instances of class
- Class “DOES” encapsulate the complexity of implementation inside
- The mechanism of hiding this complexity is known as “ENCAPSULATION”

Encapsulation

- Encapsulation is “hiding details at the level of implementation”
- Abstraction is “Exposing the interface of the class to external world”
- Access specifier “private” is used to encapsulate or hide the implementation from external world.
- Access specifier “public” is used to expose the functions or interface to external world.

Encapsulation

- Every class in JAVA is not fully encapsulated.
- It can be partially encapsulated or fully encapsulated.
- JAVA Beans are the example of Fully Encapsulated JAVA classes.
- Encapsulation provides us complete control over data.

DATA HIDING

Hiding of the data members from outside the class is known as data hiding.(to achieve full control over data)

```
public class Employee {
//hiding the data
    private int id;
// achieved by using private
modifier
}
```

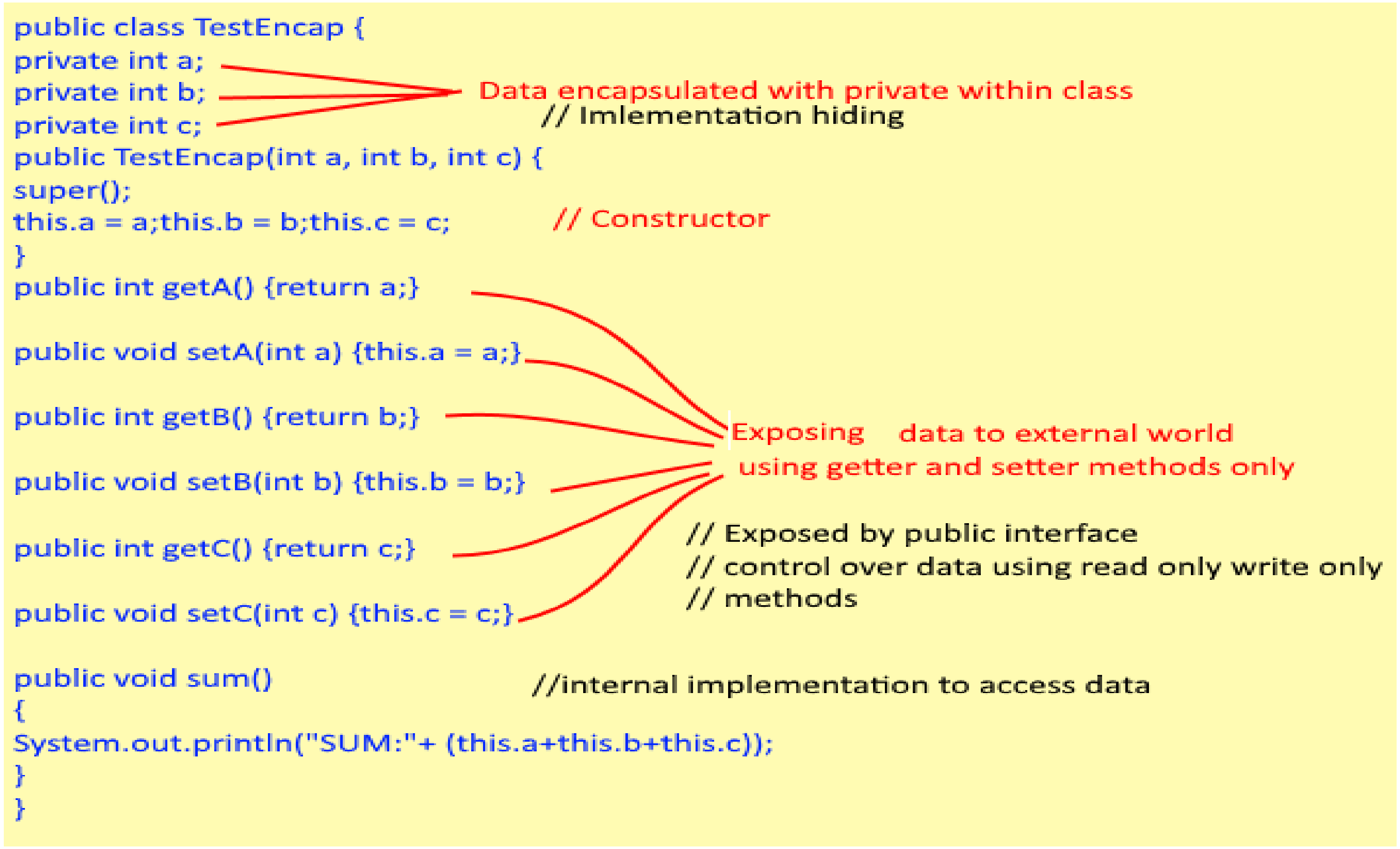
Tightly encapsulated class

A class is said to tightly encapsulated if all data members are declared as private.

All data members are accessible by getter and setter method(whether public or not).

```
public class Employee {
//hiding the data
    private int id;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

Example(Encapsulation)



```
public class TestEncap {
private int a;
private int b;
private int c;
public TestEncap(int a, int b, int c) {
super();
this.a = a;this.b = b;this.c = c;
}
public int getA() {return a;}

public void setA(int a) {this.a = a;}

public int getB() {return b;}

public void setB(int b) {this.b = b;}

public int getC() {return c;}

public void setC(int c) {this.c = c;}

public void sum()
{
System.out.println("SUM:"+ (this.a+this.b+this.c));
}
}
```

Data encapsulated with private within class
// Imlementation hiding

// Constructor

Exposing data to external world
using getter and setter methods only

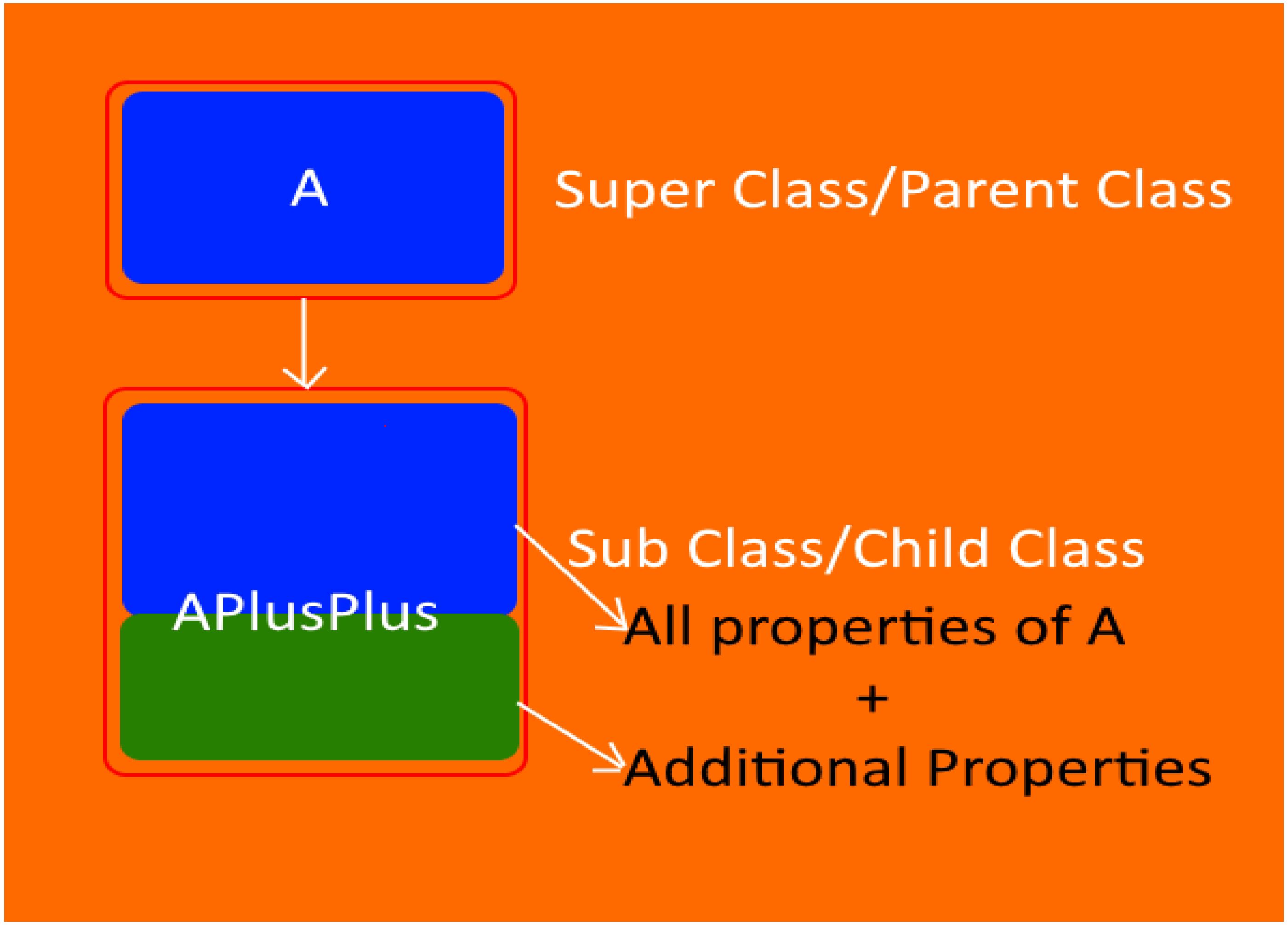
// Exposed by public interface
// control over data using read only write only
// methods

//internal implementation to access data

JAVA Inheritance

- Java uses single inheritance model.
- One subclass can have one(and only one) super class.
- “extends” keyword is used to implement inheritance.
- A subclass include all the members of its superclass.
- (Constructor are exception)

Inheritance



- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

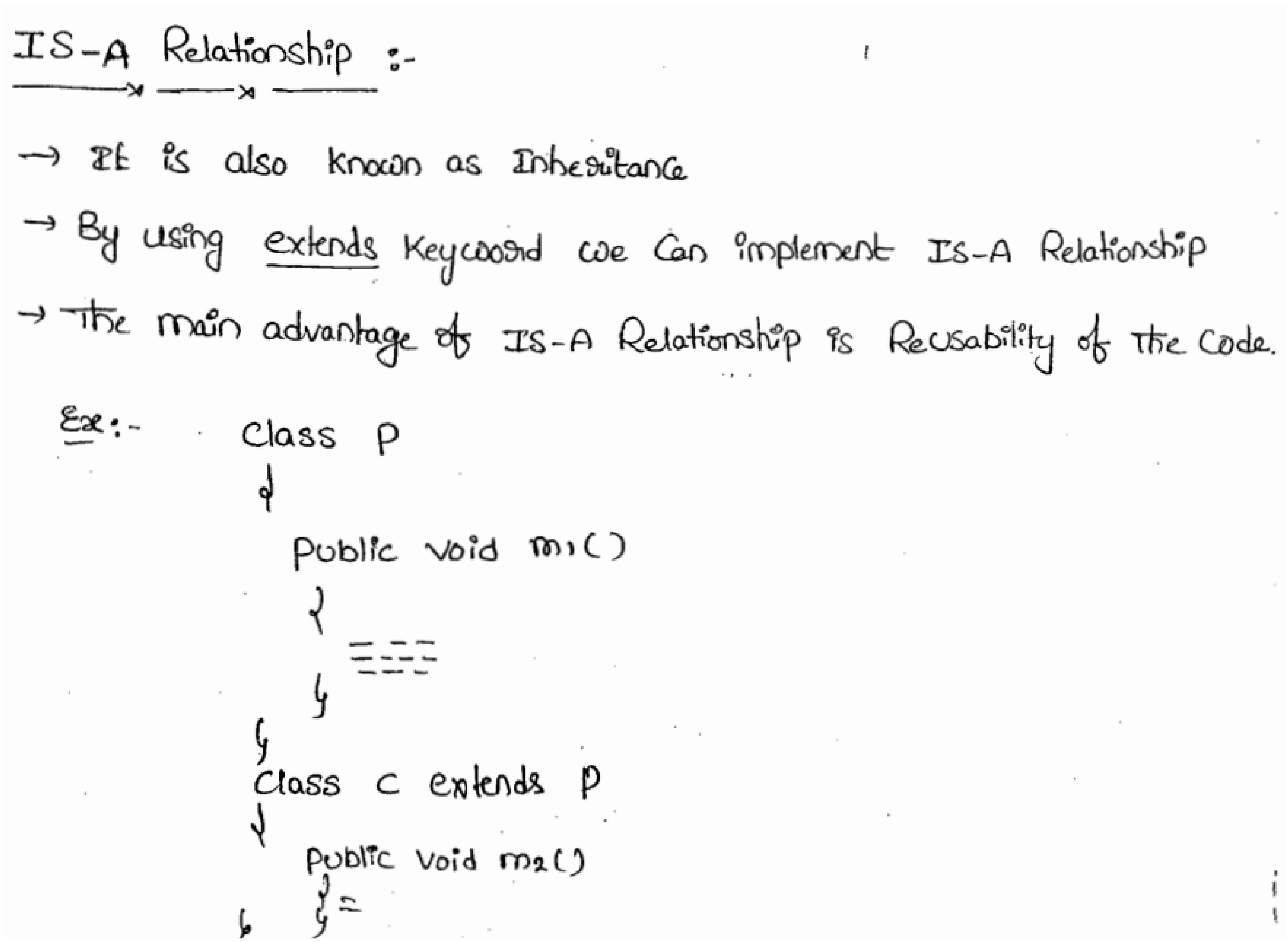
“extends” keyword

“extends” keyword is used to create new sub class from a super class

```
Class MySubClass extends MySuperClass
{
// To-Do
}
```

```java
class Shape {
    private String color;
    public boolean isFilled;
    public Shape(String color,boolean isFilled) {
        super();// will refer to Object class constructor
        this.color = color;
        this.isFilled=isFilled;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
}
class Rectangle extends Shape{
    int ln;
    int wd;
    public Rectangle(int ln, int wd, String color,
boolean isFilled) {
        super(color,isFilled);// will refer to shape
class constructor
        this.ln = ln;
        this.wd = wd;
    }
}
```

Inheritance



IS-A Relationship :-

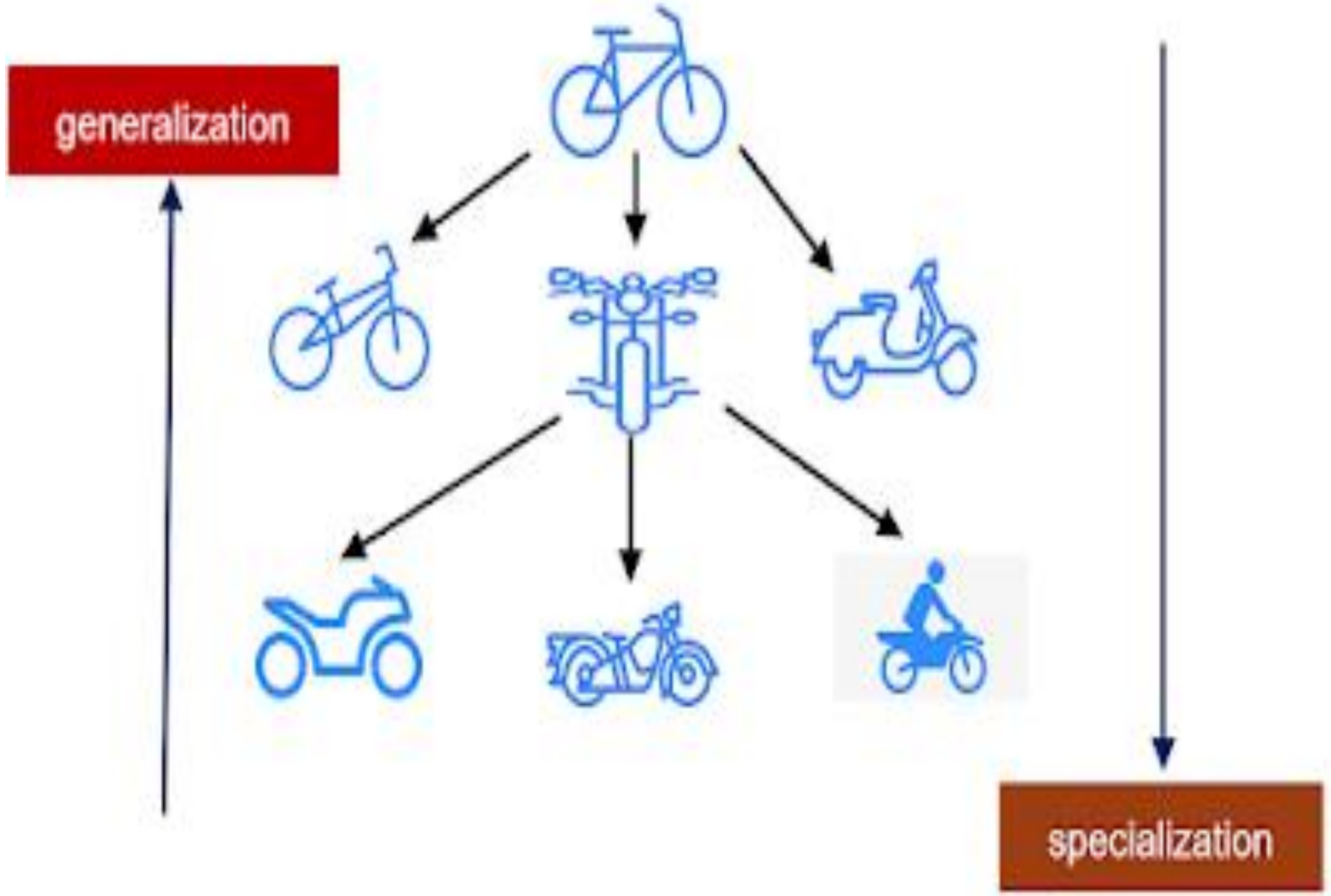
→ It is also known as Inheritance

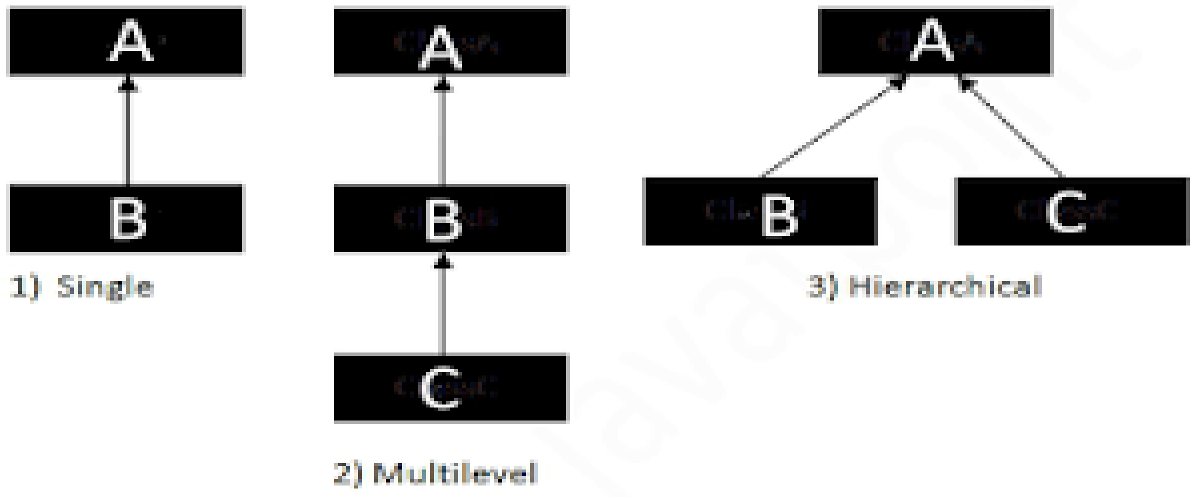
→ By using <u>extends</u> Keyword we can implement IS-A Relationship

→ The main advantage of IS-A Relationship is Reusability of the Code.

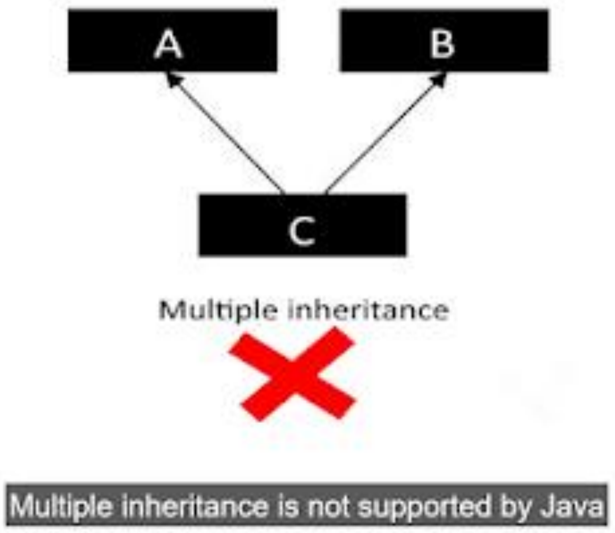
<u>Ex</u>:-

```
Class P
{
   Public void m1()
   {
      ===
   }
}
Class C extends P
{
   Public void m2()
   {=
   }
}
```


generalization
specialization



Types of inheritance in Java



6) Has - A Relationship :-

→ Has- A Relationship is also known as "Composition or Aggregation".

→ There is no specific keyword to implement Has - A Relationship the mostly we are using 'new keyword'.

→ The main advantage of Has - A Relationship is Reusability or (code Reusability)

ex:-

```
Class Car                          Class Engine
{                                  {
  Engine e = new Engine();           // Engine specific functionality
}                                  }
```

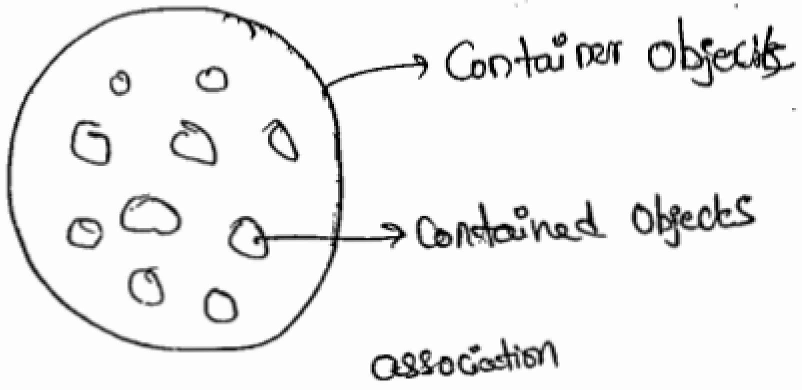
Class Car has Engine reference.

→ The main disadvantage of Has-A Relationship is it increases dependency b/w the classes and creates maintaince problems.

→ The main disadvantage of Has-A Relationship is it increases dependency b/w the classes and creates maintaince problems.

<u>Composition Vs Aggregation:-</u>

→ In the case of Composition whenever Container objects is destroyed all Contained objects will be destroyed automatically. i.e, without Existing Container object there is no chance of existing Contained object i.e Container & Contained objects having Strong association

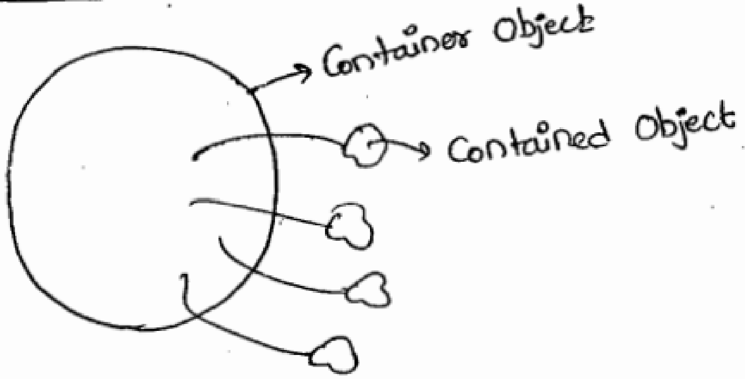


Example-> University is Composed of Several departments

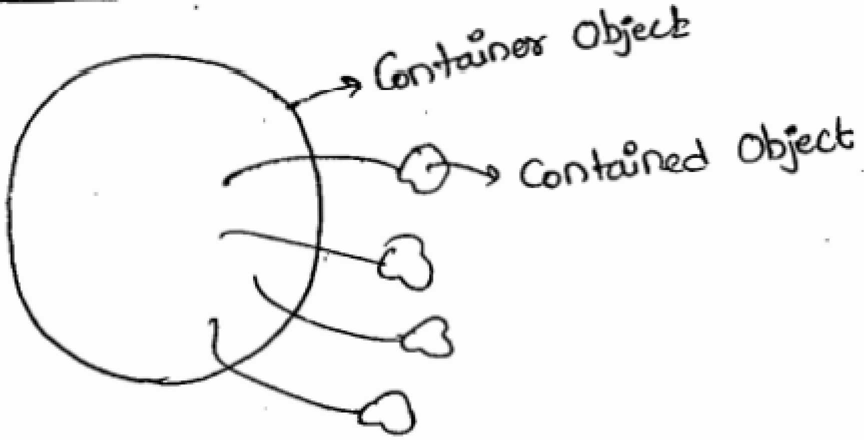
→ whenever you are closing University automatically all departments will be closed. the relationship b/w University object & department object is strong association which is nothing but Composition.

Aggregation :-

→ whenever Container object destroyed, There is no guaranty of destruction of Contained objects i.e, without existing Container object there may be a chance of existing Contained object. i.e, Container object just maintains References of Contained objects. This relationship is Called weak association which is nothing but "Aggregation".



"Aggregation".



Ex:-

→ Several professors will work in the department

→ whenever we are closing the department still there may be a chance of existing professors. The relationship b/w department & professor is called weak association which is nothing but Aggregation.

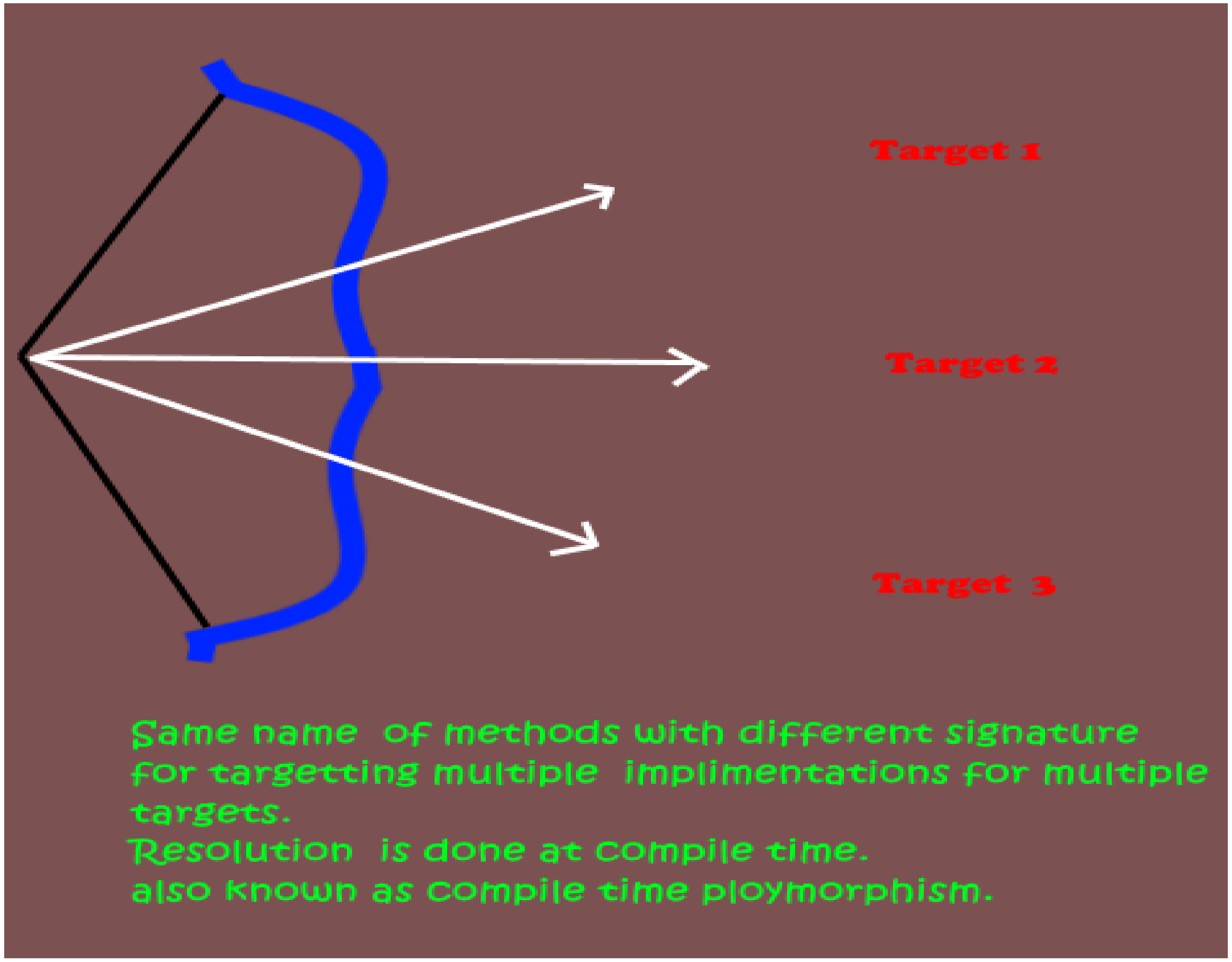
Polymorphism

Over loading, Over riding and Dynamic method dispatch

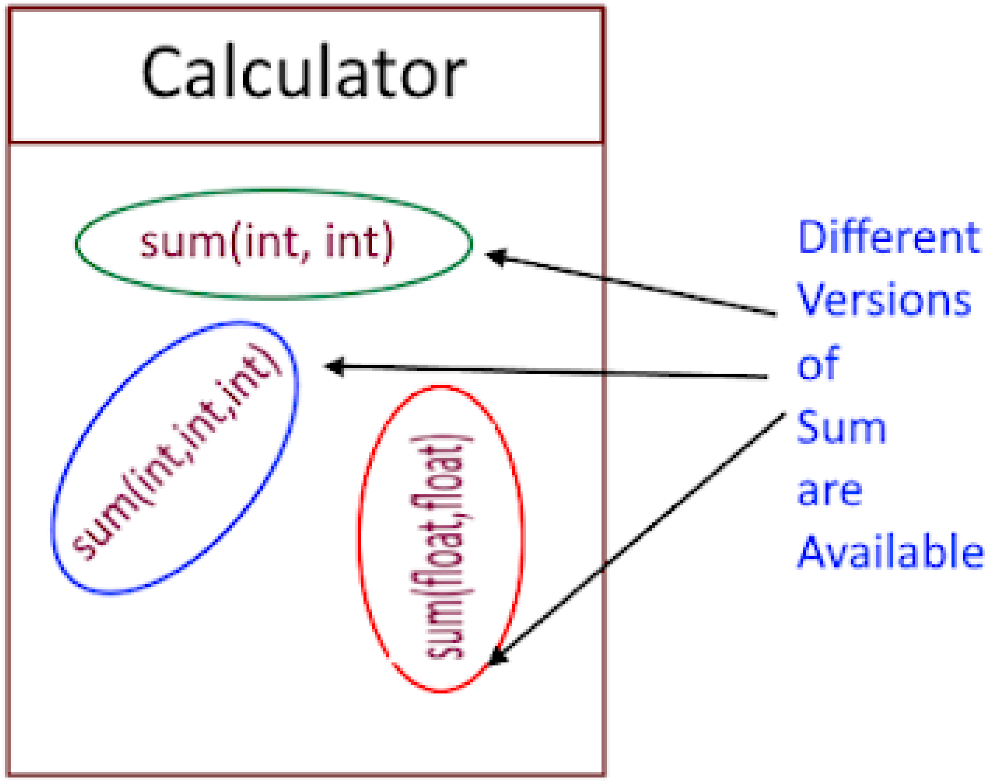
Polymorphism

- Polymorphism means “MANY FORMS”
- “One interface-multiple methods”
- Creation of generic interface to group of relative activities
- Polymorphism in JAVA is Achieved by method overloading and method overriding

Method Overloading



Same name of methods with different signature for targetting multiple implimentations for multiple targets.
Resolution is done at compile time.
also known as compile time ploymorphism.


Calculator
sum(int, int)
sum(int,int,int)
sum(float,float)
Different Versions of Sum are Available

Method Overloading

- In method overloading, there are two or more methods having same name in same class but having different signature(number, type and order of arguments).
- Changing only the return type of method is not overloading.
- It is just duplication of the method.

Method Overloading

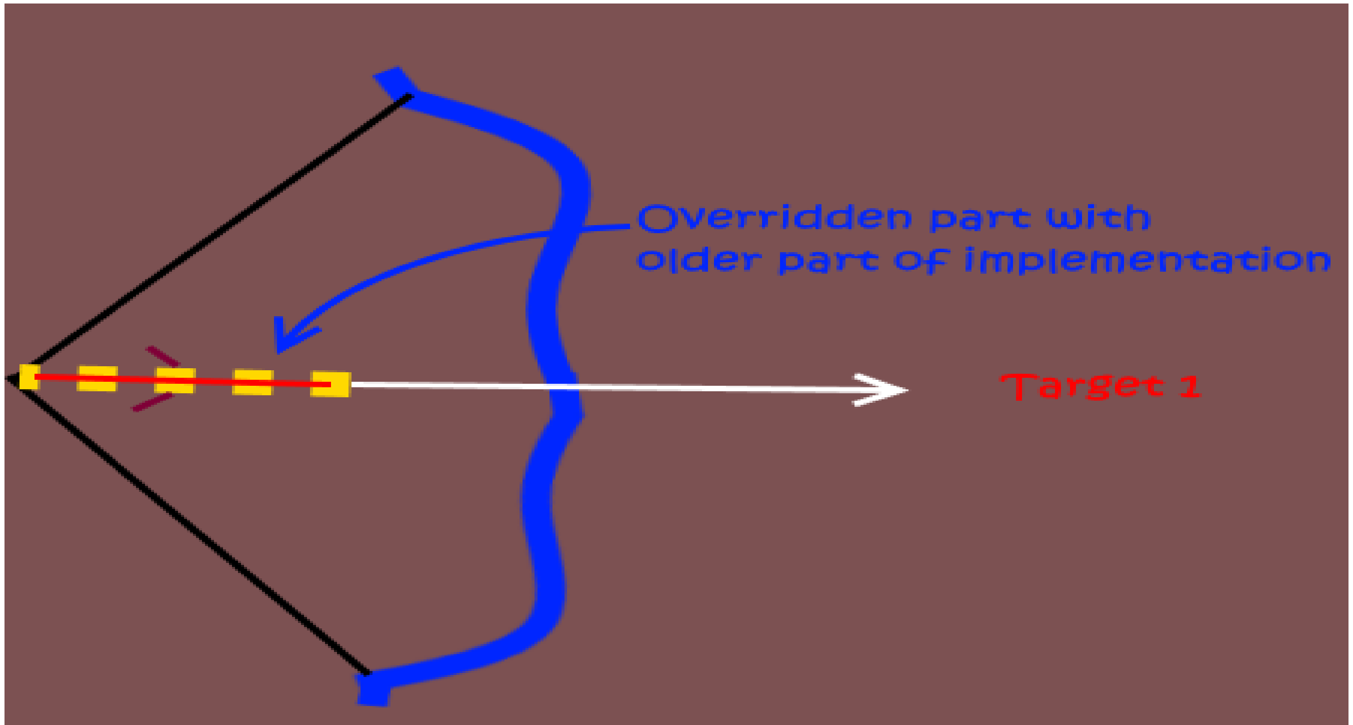
```java
class TestOverLoading {
    public void add(int a, int b) {
        System.out.println(a + b);
    }

    public void add(int a, int b, int c) {
        System.out.println(a + b);
    }

    public void add(int a, float f, int c) {
        System.out.println(a + (int) f + c);
    }

    public void add(float f, int c, int a) {
        System.out.println(a + (int) f + c);
    }
}
```

Method Overloading

```
public class Employee{

    public static void main(String... args)
    {
        TestOverLoading tl=new
TestOverLoading();
        tl.add(5, 4);
        tl.add(5, 7, 9);
        tl.add(5, 7.68f, 8);
        tl.add(9.56f, 7,8);
    }

}
```

```
Output: 9
        12
        20
        24
```

Method Overriding



->Method overriding is done at run time only
->Aslo known as run time polymorphism
->customized version of method is available in sub class.
->This method may or may not add previous method implementation
->This method is generally more refined version of prevoius one its super class.

Method Overriding

- When a method in a sub class has same prototype as a method of its super class, than the this method is said to be over riding method in sub class.
- When an overridden method is called by the object of sub class than the version defined in sub class will be always referred.
- The version defined in super class is hidden or overridden.
- Same prototype means

**same name*

**same signature(same type, order and list of parameters)*

**same return type*

Method Overriding

- A super class reference variable can refer to a subclass object
- Method calls in java are resolved dynamically at run time
- In JAVA all variable knows their dynamic type
- Method calls are bound to the methods on the basis of dynamic type of the receiver

Method Overriding(Rules)

- Same argument List and order
- Same return type
- Should not have more restrictive access specifier
- Must not throw new or broader checked exceptions
- Final methods can’t be overridden
- Constructors can’t be overridden

Method Overriding

- “single interface, multiple implementations” aspect of polymorphism
- The super class contains all elements which sub class can use directly.
- Provides sub class the functionality to provide it’s own specific implementations.

Dynamic method dispatch

- Method overriding forms the basis of “Dynamic Method Dispatch” which is one of the most powerful concepts of JAVA.
- It is capability of JAVA to resolve the calls at runtime dynamically to overridden methods.

Runtime polymorphism is useful for resolving

-super class reference variables

-overridden methods

Dynamic method dispatch

When a method is called through the super class reference variable, java determines which version of the method to call based upon the type of the object being referred at the time of call occurs.

Dynamic method dispatch

```java
class A {
    public void test()
    {
        System.out.println("i m From A");
    }
}
class B extends A{
    @Override
    public void test()
    {
        System.out.println("i m From B");
    }
}
class C extends B{
    @Override
    public void test()
    {
        System.out.println("i m From C");
    }
}
```

Dynamic method dispatch

```
public class Employee extends Person{
    public static void main(String... args)
    {
        A a=new A();
        A ab=new B();
        A ac=new C();
        a.test();
        ab.test();
        ac.test();
    }

}
```

Output:

```
i m From A
i m From B
i m From C
```

→ If child class method throws some checked Exception then compulsary Parent class method should throw the same checked Exception or it's Parent class Exception. otherwise we will get C.E .

→ But there is no rule for unchecked Exception.

<u>Ex</u>:-①

```
Class P
{
  Public void m1()
  {
  }
}
Class C extends P
{
  Public void m1() throws Exception  ✗
  {
  }
}
```

<u>C.E</u>!- m1() in C Can't override m1() in P;

Overridden method doesnot throw Exception .

Ex⑤:-

ⓐ ✓
P: public void m1() throws IOException
C: public void m1()

ⓑ ✗
P: public void m1()
C: public void m1() throws IOException

ⓒ ✓
P: public void m1() throws Exception
C: public void m1() throws IOException

ⓓ ✗
P: public void m1() throws IOException
C: public void m1() throws Exception

⑤ ✓
P: public void m1() throws IOException
C: public void m1() throws FileNotFoundException, EOFException

⑥ p: public void m1() throws IOException

✗ c: public void m1() throws EOFException, InterruptedException

⑦ p: public void m1() throws IOException

✓ c: public void m1() throws AE, NPE

⑧ p: public void m1()

✓ c: public void m1() throws AE, NPE

Overriding w.r.t Static method :-

→ We Can't override a Static method as non-Static.

$$\begin{tikzcd} \text{Static} \arrow[d, shift right, "\times"] \\ \text{non-Static} \arrow[u, shift right] \end{tikzcd}$$

Ex:-

```
Class P
{
  Public Static Void m1()
   {
    }
}
 Class C extends P
  {  public void m1()
      {                    X
      }
  }
```

C.E:- m1() is Can't overide m1() in P;
overriden method is Static.

→ If both parent & child Class method ~~Class~~ are Static Then We wont to get any C.E it Seems to be overriding is happen, but it is not overriding. it is "Method Hiding".

Ex:-

```
Class P
{
  Public Static void m1()
   {
   }
}
Class C extends P
{
  Public Static void m1()
   {
   }
}
```

it is not overriding it is "method Hiding"

Method Hiding :-

→ All rules of Method Hiding are Exactly Same as Overriding Except the following difference.

method hiding	Overriding
1) Both methods should be Static	1) Both methods should be non-Static
2) Method resolution takes Care by Compiler based on Reference type.	2) Metho resolution always takes Care by JVM based on Runtime object.
3) It is Considered as Compiletime Polymorphism or Static polymorphism or early Binding	3) It is Considered as Runtime Polymorphism or dynamic polymorphism or Late Binding.

```java
class Parent{// method hiding demo
    public static void m1()
    {System.out.println("Parent Class method");}
}
class Child extends Parent{
    public static void m1()
    {System.out.println("Child Class method");}
}
public class Demo {
    public static void main(String... arg)
    {
        Parent parent=new Parent();
        parent.m1();//op=> Parent Class method
        Child child=new Child();
        child.m1();//op=>Child Class method
        Parent parent1=new Child();
        parent1.m1();//=>Parent Class method
    }
}
```

See Here

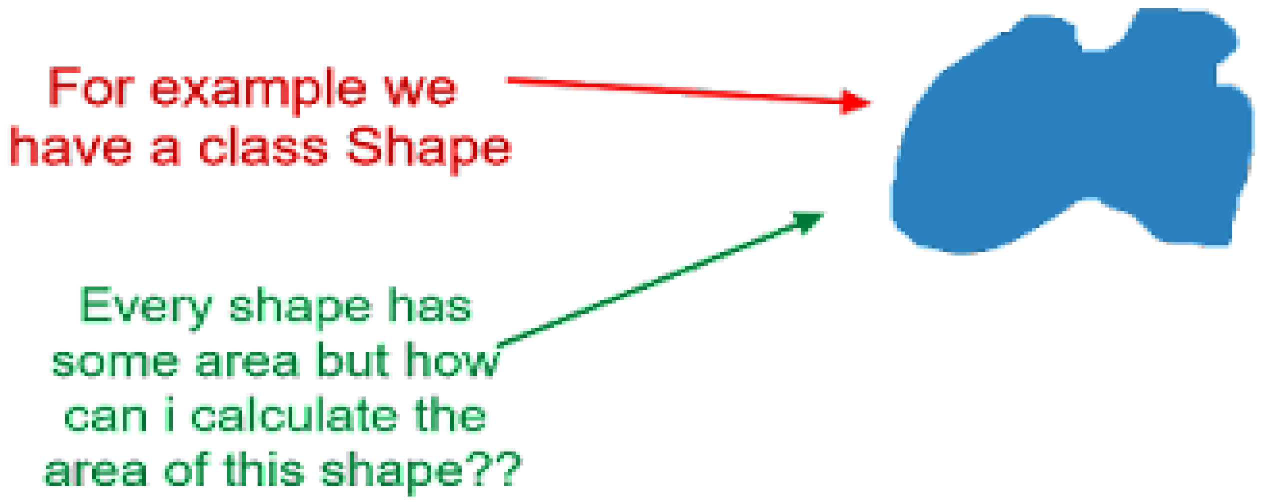
```
class Parent{// method overriding demo

    public void m1()
    {System.out.println("Parent Class 
method");}
}
class Child extends Parent{
    public void m1()
    {System.out.println("Child Class 
method");}
}
public class Demo {
    public static void main(String... arg)
    {
        Parent parent=new Parent();
        parent.m1();//op=> Parent Class method
        Child child=new Child();
        child.m1();//op=>Child Class method
        Parent parent1=new Child();
        parent1.m1();//=>op=>Child Class 
method
    }
}
```

See Here

Abstract Classes

Abstract classes are one of the ways to apply Abstraction in Java. Another way to apply Abstraction in Java is Interfaces. A class can be "abstract" in nature or "concrete".Concrete classes are complete classes and we can create the objects of concrete classes and use them. Abstract classes are incomplete classes and we can not create objects of abstract classes. When one or more than one method of the class is abstract than that class must also be declared as abstract. Any class extending the abstract class either must implement the abstract methods in its superclass or it should be declared as abstract also.

Abstract Classes


For example we have a class Shape
Every shape has some area but how can i calculate the area of this shape??

In the diagram above, The class "Shape" is an abstract class (incomplete knowledge of behavior)
so we must declare it as an abstract class.

Any class extending the abstract class either must implement the abstract methods in its superclass or it should be declared as abstract also.

```java
abstract class A{   //Abstract class having abstract
//method
    abstract void m1();//Abstract method
    void m2()           // Concrete method
    {
        System.out.println("I am in Super class A");
    }
}
class B extends A{ //Concrete class

    @Override
    void m1() {
        System.out.println("I am in Sub class B");
    }//The type B must implement the inherited
    //abstract method A.m1()

}
public class Demo {
    public static void main(String... arg)
    {
        //  A a=new A(); not possible,
        //we can not create an object of any Abstract class
        B b=new B();
        b.m2();
        b.m1();
    }
}
```

Abstract Classes

An abstract class may have static fields and static methods. You can use these static members with a class reference (for example, ***AbstractClassName.staticMethodName()***) as you would with any other class. An abstract class can not be declared as final.

Interfaces

Interfaces

- In Java, we can not only achieve *Abstraction,* with the help Abstract classes but also with the help of Interfaces.
- Interfaces provides a skeleton of a class.
- We can only declare public, static and final variable in an Interface.
- All the methods declared in an Interface are public and abstract.(a method without implementation)

•In Java, we can not only achieve Abstraction, with the help of Abstract classes but also with the help of Interfaces.
•Interfaces provide a skeleton of a class.
•All variables declared in an interface are **public**, **static** and **final**

```
/****Declaring an Interface**********/
 interface MyInterface{
//all variable(will be public static and final)
//all methods(only declartions/no implementations)
}
```

•All the methods declared in an Interface are **public** and **abstract** (a method without implementation).
•The interface provides a protocol of behavior(methods) or how the behavior is implemented is the responsibility of that Individual class.
•Syntactically an interface is declared the same as a class with a keyword ***interface***.
•The keyword ***"implements"*** is used to apply any interfaces in any class.
•Interfaces provide a mechanism to provide multiple inheritance.
•A single interface can be implemented by several classes.

Declaration & Implementation of an Interface :-

→ We can declare an Interface by using Interface keyword, we can implement an Interface by using implements keyword.

Ex:-

```
interface Interf
{
  void m1();   // by default public abstract void m1();
  void m2();
}
abstract class ServiceProvider implements Interf
{
 → public void m1()
   {
   }
}
```

→ If a class implements an interface compulsary we should provide implementation for every method of that interface otherwise we have to declare class as abstract. Violation leads to Compile-time Error.

```java
interface A {
int v1=1;
}
// interfaces can be inherited
interface B extends A
{
int v2=2;
int sum=v2+A.v1;
}
```

Why Interfaces required?

- Interface allows us to implement common behaviour in two different classes.
- Interfaces allows us to apply behaviours beyond classes and class hierarchy.
- Consistent specification among unrelated classes.
- Reduce coupling between software component
- Two or more classes may not achieve same functionality.(No duplicity of class)
- Create richer classes.

An interface can declare three kinds of members:

- Constants //all fields are public static and final
- methods//all methods are public and abstract
- nested classes and interfaces

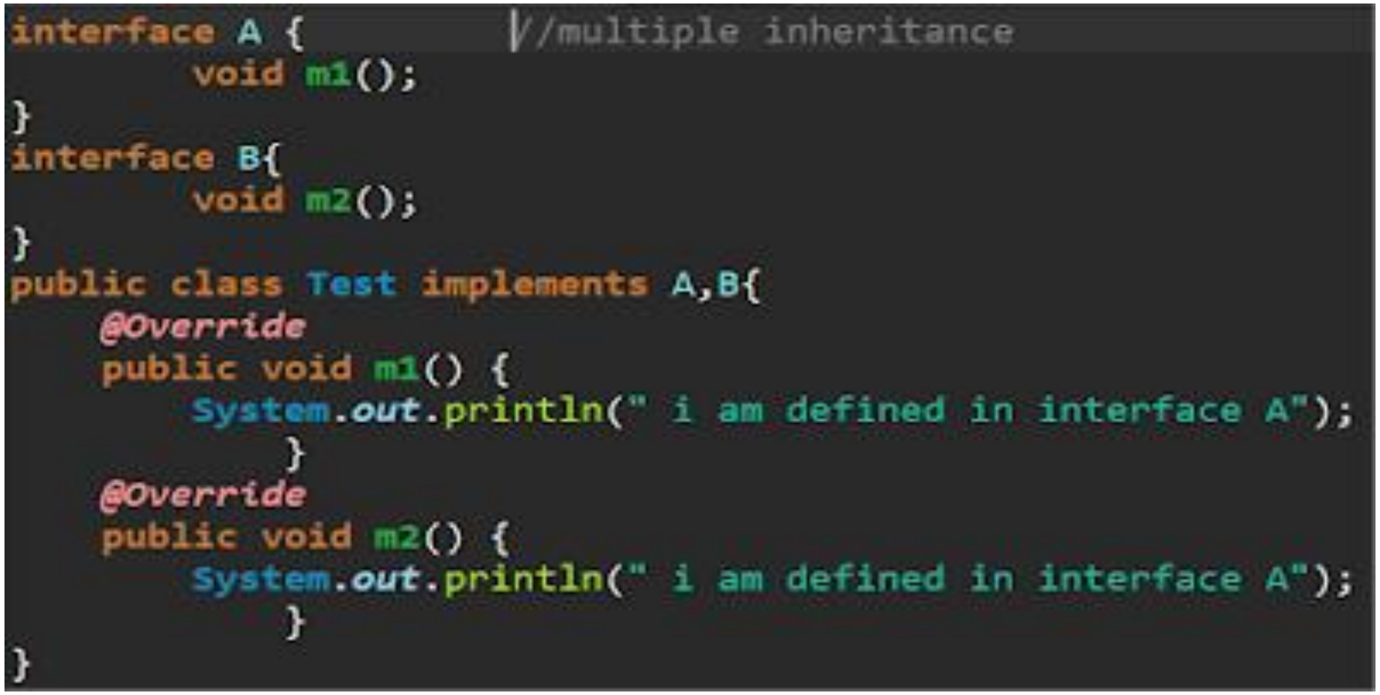
An interface can be preceded by any of these three modifiers

- public(without public modifier interface is only accessible inside its own package only)
- abstract(can not be static,final,native or synchronized)
- strictfp (for strict floating-point evaluation)

Multiple Inheritance

Several other object-oriented programming languages support a kind of Inheritance known as multiple inheritances. When a subclass extends two or more than two super classes, it is known as multiple inheritances. This type of inheritance is not supported by Java. This can be provisionally applied in Java with the help of Interfaces.

For example, interface A and B are implemented by the class in the following example.



```
interface A {            //multiple inheritance
        void m1();
}
interface B{
        void m2();
}
public class Test implements A,B{
    @Override
    public void m1() {
        System.out.println(" i am defined in interface A");
            }
    @Override
    public void m2() {
        System.out.println(" i am defined in interface A");
            }
}
```

Q) which of the following is True?

(1) A class can extend any no. of classes at a time. ✗

(2) A class can implement only one Interface at a time. ✗.

(3) A class can extend a class ~~and~~ or can ~~implemplmt~~ implement an interface but not both simultaneously ✗

(4) An Interface can extend only one interface at a time ✗

(5) An Interface can implement any no. of classes at a time ✗

(6) None of the above ✓

→ As every interface method is bydefault public & abstract the following modifier are not applicable for interface methods.

(1) private
(2) protected
(3) <default>
(4) final
} ✗

(5) static
(6) strictfp
(7) synchronized
(8) native
} ✗

Interface nameing Conflicts :-

① method naming Conflicts :-

Case1 :-

→ If Two interfaces Contains a method with Same Signature & Same return type in the implementation class we can provide implementation for only one method.

Ex:-

```
interface Left              |   interface Right
{                           |   {
   Public void m1();        |     Public void m1();
}                           |   }

Class Test implements Left, Right
{
   Public void m1()
   {
   }
}
```

✓

9

Case 2 :-

→ If Two interfaces Contains a method with Same name but different args then, in the implementation class we have to provide implementation for both methods & these methods are Considered as overloaded methods.

Eg:-

```
interface Left                 |  interface Right
{                              |  {
    public void m1();          |    public void m1(int i);
}                              |  }


class Test implements Left, Right
{
    public void m1()
    {
    }
    public void m1(int i)
    {
    }
}
```

overloaded methods

Packages

Package:-

→ It is an Encapsulation mechanism to group related Classes and interfaces into a Single module. The main purposes of packages are

1. To resolve naming Conflicts.
2. To provide Security to the classes & interfaces. So that out Side person Can't access directly
3. It improves modularity of the application.

→ There is one Universally accepted Convension to name packages i.e to use internet domain name in reverse.

$$\underbrace{\text{Com.icicibank}}_{\text{domain name in reverse}}.\underbrace{\text{loan}}_{\text{module name}}.\underbrace{\text{housingloan}}_{\text{Submodule name}}.\underbrace{\text{Account}}_{\text{Class name.}}$$

①
→ In Any Java program there should be only At most 1 packg
Statement. If we are taking morethan one package Statement we will get Compiletime Error.

Ex:-
```
✓ package pack1;
→ package pack2; ←
  class A
  {
  }
```
C.E:- class, interface or enum expected.

② In Any Java program the first non Comment statement should be package Statement (if it is available).

Ex:-
```
✓ import java.util.*;
→ package pack1;
  class A
  {
  }
```
C.E:- class, interface or enum Expected.

→ The proper Structure of a Java Source file is

order is important ↓

```
package Statement;                          → Atmost one
import Statements;                          → Any number
class/interface/enum declarations           → Any number
```

→ The following are Valid Java programs.

① (empty) ✓
Test.java

② ✓
```
package pak1
```
Test.java

③ ✓
```
import java.lan
```
Test.java

④ ✓
```
package pck1
import java.L
```
Test.java

⑤ ✓ → in this prog only creates .class file.
```
class A
{
}
```
Test.java

An Empty Source file is a Valid Java program.

Thank You