Object oriented programming using JAVA

UNIT I

Procedural language

(program code talks)

[code acting on data] → (what is happening)

1. Separate data from functions that operate on them.
2. It is difficult to implement any changes
3. Debugging is difficult
4. Not suitable for defining abstract types
5. Less flexible
6. uses top-down approach
7. communication done through Inter-function communication.
8. Less reusable
9. procedure oriented
10. Not suitable for larger programs and applications

~~example~~: pascal, fortran, COBOL, BASIC

OOP

[data controlling accessing to code] → (who is being affected)

1. Encapsulate data and methods in a class.
2. It is easier to manage and implement changes.
3. Debugging is easier.
4. Suitable for defining abstract types.
5. Highly flexible.
6. Uses bottom-up approach
7. object-communication is possible.
8. More reusable
9. Object oriented
10. Suitable for larger programs and applications.

~~example~~: C++, C#, BETA, Chapel, JAVA, Small talk

Creation of JAVA

↳ This language was initially called "oak" but was renamed "JAVA"

oak → 1991

JAVA → 1995

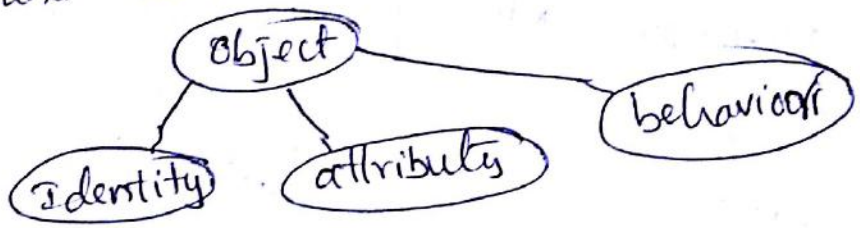
Advantages over C++: JAVA → [Internet version of C++]

Java expands the universe of objects that can move about freely in cyber space.

object-oriented programming (OOP) concepts:

1. object
2. class
3. Encapsulation
4. Abstraction
5. Inheritance
6. polymorphism
7. Dynamic Binding

oop is modeled around the real world which contains full of objects. Every object has a well-defined identity, attributes and behaviour.



class: collection of objects with similar properties. The binding up of data and methods into a single unit called class

Syntax:

```
class classname
{
    data members;
    methods();
}
```

Object: Any entity that has state and behavior is known as an object. example chair, pen, table, keyboard, bike, ... It can be physical (or) logical.

object is an instance of a class.

syntax:

```
classname variable = new classname();
```

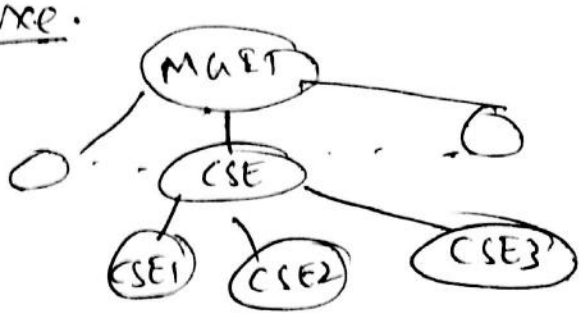
Encapsulation: Binding (or wrapping) up of data and the methods that can manipulate that data into a single unit (class) is called Encapsulation.

Ex: capsule [different medicines].
A Java class is the example of encapsulation.

Abstraction: showing only the essential features and hiding the details is known as abstraction.

Ex: phone call.
In Java, we use abstract class and interface to achieve abstraction.

Inheritance: Inheriting [Acquiring] the properties of one class (parent/base class) into other class [derived/sub class] is called Inheritance.



polymorphism: It means many forms. Expressing a single thing (task) in multiple forms is called polymorphism.
In Java, we use method overloading and method overriding to achieve polymorphism.

Dynamic Binding: Linking up of the method call to method definition is called binding. If it is done at compile time, it is called compile time binding or static binding. If it is done at run time, it is called run-time binding or dynamic binding.

Java Buzzwords

1. Simple
2. Secure
3. Object-oriented
4. Architecture-neutral
5. portable
6. compiled & Interpreted
7. High performance
8. Multi-threaded
9. Distributed
10. Dynamic
11. Robust.

(1) **simple**: In Java, there are a small no of clearly defined ways to accomplish a given task.

→ The knowledge of C-language makes Java easy to learn/understand

→ The knowledge of C++ helps to understand OOP concept better.

(2) **Secure**: Java achieves security by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer. Java is more secure because of bytecode and inbuilt firewalls which restrict malicious content to access the local system.

(3) **object-oriented**:

Java is an OOP language which follows OOP concepts.

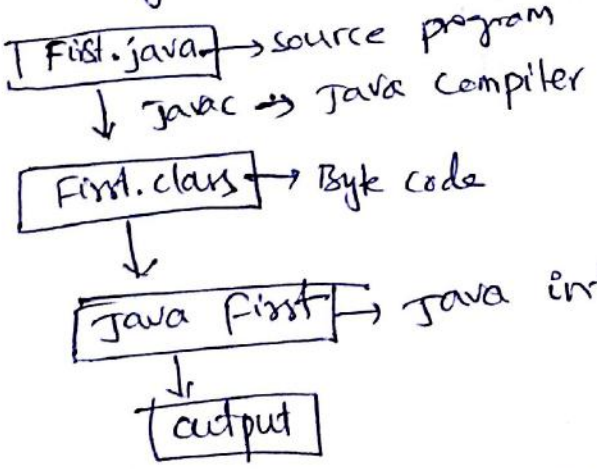
(4) **Architecture-Neutral** [platform Independent]

A Java program can be run on any platform i.e different cpus, different operating systems and different system resources. so, it is called platform independent & Architectural neutral.

This is possible because of byte code can be executed by JDK's [Java Development kit]. so, having a JDK on the system makes Java program to run. JDK's are platform Independent.

portable:
The architecture neutral of Java program makes it easy to execute the program anywhere. It follows "write once, run any where". ~~Option~~. Java also allows tranferring of data via Internet easily.

compiled & Interpretted:
Java is both a compiled & interpretted language. When a Java file is compiled, it results in a class file, that consists of byte code. This class file is interpretted to get the output. An optimised set of instructions that can be understood only by the Java interpretter is called byte code.



Multithreaded: It allows the programmer to write programs that do many things simultaneously to meet the real-world requirements.

Distributed: Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. Java ~~has a~~ contains a package called Remote Method Invocation (RMI) which brings an unparalleled level of abstraction to client/server programming.

Dynamic : Java supports dynamic binding (Dynamic binding dispatch)

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run-time.

Robust : It is said to robust language becuse of

① memory management
② Exception management.

Dynamic
Memory management is done using new operator which is difficult in traditional programming environments. Deallocation is completly automatic, Java provides garbage collection for unused objects.

Exceptions like divide-by-zero or File not found can be handled by a Java program efficiently and easily compare to traditional programming.

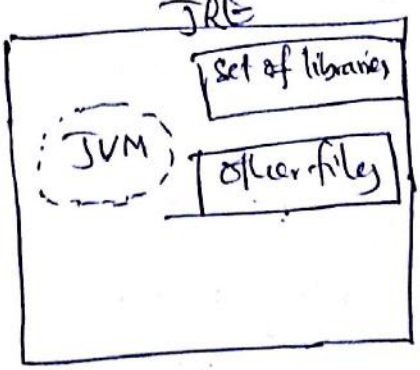
JVM, JRE and JDK :

JVM [Java virtual machine] : It is an abstract machine. It is a specification that provides runtime environment in which Java bytecode can be executed. JVMs are available for many h/w and s/w platforms. JVM, JRE and JDK are platform dependent because of each OS differs but Java is platform independent.

JVM performs the following tasks:

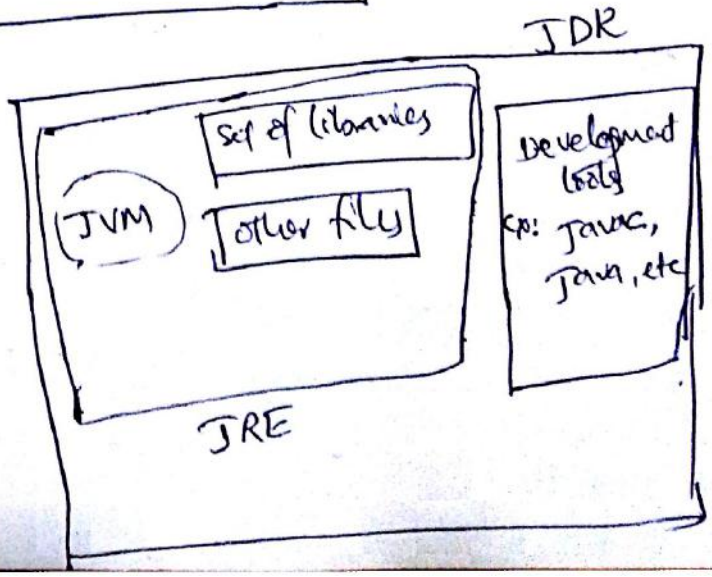
- → Loads code
- → verifies code
- → Executes code
- → Provides run-time environment.

JRE [Java Runtime Environment] : It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

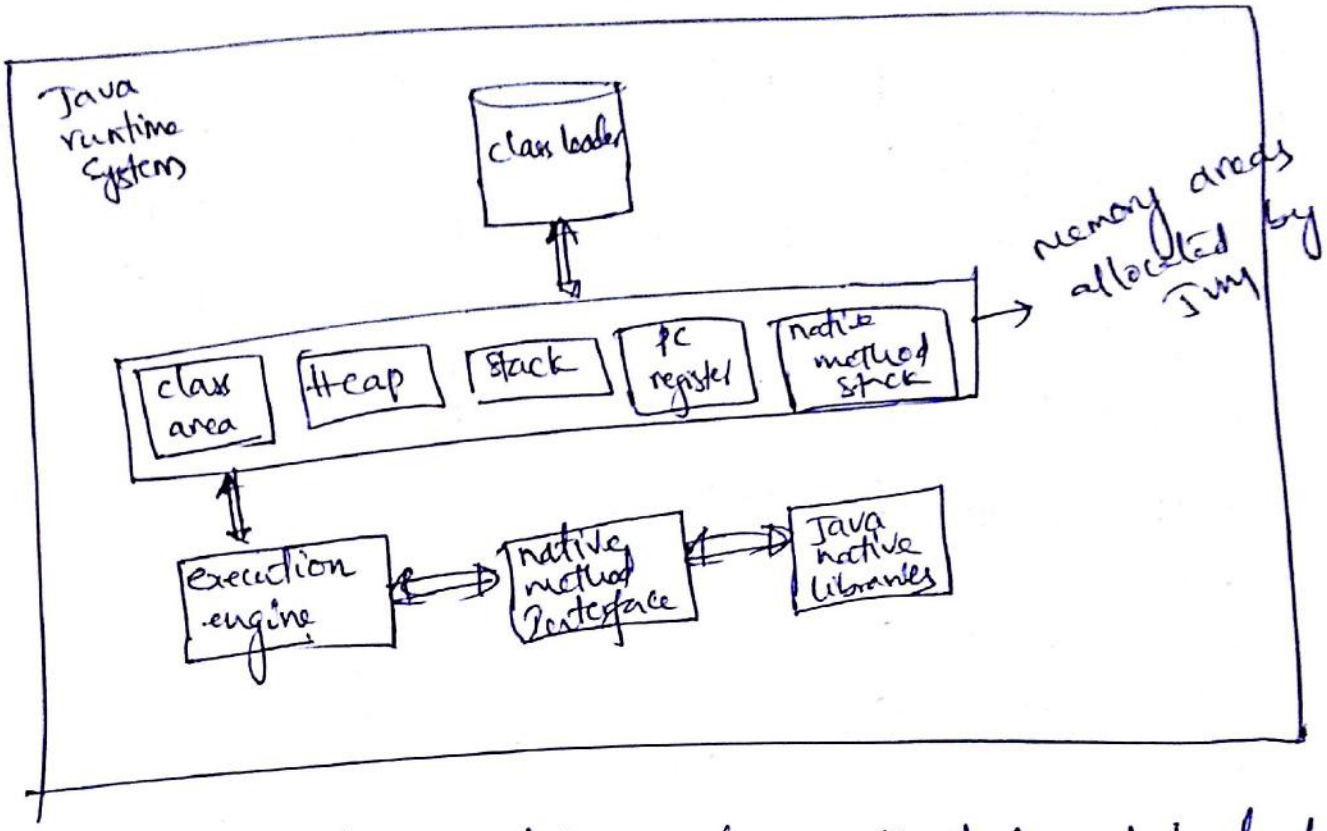


JDK

It contains JRE + development Tools



Internal Architecture of JVM



class loader: It is a subsystem of JVM that is used to load class files.

class area: It stores pre-class structures such as the runtime constant pool, field and method data, the code for methods.

Heap: It is the runtime data area in which objects are allocated.

stack: Java stack stores frames, local variables and partial results.

program counter register: It contains the address of the JVM instruction currently being executed.

Native method stack: It contains all the native methods.

Execution engine: It contains

1. A virtual processor
2. Interpreter: Read byte code stream then execute the instructions
3. Just-In-Time (JIT) compiler: JIT compiles parts of the byte code that have similar needed for compilation to improve the performance.

Simple program [To print 'Helloworld']

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

Filename: Helloworld.java

To compile: { Javac Helloworld.java } → compiler
{ Generates Helloworld.class } [Byte code] class file.

To run: { Java Helloworld } gives output.
↳ Interpreter

/* . . . . . } multiline comment.
. . . . . */

// . . . . . } singleline comment.

execution begins at
↳ public static void main (String args[])

public: It is an access specifier, which allows the programmer to control the visibility of class members. when a class member is preceded by public, then it may be accessed by code outside the class in which it is declared. main() must be declared as public, since it must be called by code outside of its class when the program is started.

static: It allows main() to be called by the Java interpreter without having to instantiate a particular instance of the class.

void: to tell main() doesnot return a value.

main(): It is the method called when a Java application begins.

String args[]: It declares a parameter named args, which is an array of instances of the class String. Objects of String store character strings.

Write a java program for addition of two number.

```
class add
{
    public static void main( String args[])
    {
        int a = 10;
        int b = 20;
        int c;
            c = a+b;
        System.out.println (" result is : " + c);
    }
}
```

Data types

① byte
② short
③ int
④ long
} Integers

⑤ float
⑥ double
} floating-point numbers

⑦ char
⑧ Boolean

[positive and negative]

Integers: [byte, short, int, long] → All are signed. Java does not support unsigned [only positive numbers]

Data type	size (in bits)	Range
byte	8	-2^7 to 2^7-1 (-128 to 127)
~~int~~ short	16	-2^{15} to $2^{15}-1$
int	32	-2^{31} to $2^{31}-1$
long	64	-2^{63} to $2^{63}-1$
float	32	1.4 e-045 to 3.4 e+038
double	64	4.9 e-324 to 1.8 e+308
char	16	0 to 65536

characters: Java uses unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found on all human languages. [Greek, Latin, Arabic, ...]

→ There are no negative chars.

Boolean: It is for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators, such as a < b and also the type required for the conditions in 'if' and 'for'.

A variable can be defined as a boolean type.

Ex: `boolean d = true`

variables

The variable is the basic unit of storage in a java program. A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaration:

~~type~~ datatype identifier [= value] ;

ex: int a, b, c;
int d = 3, e, f = 5;
double pi = 3.14159;
char x = 'x';

Type conversion and Type casting

when one type of data is assigned to another type of variable, an automatic type conversion will take place if

(i) the two types are compatible
(ii) The destination type is larger than the source type.

```
int x = 10; y = 20;
float z;
z = x + y;
```

Converting (or) storing a higher data type value into a lower data type variable cause errors (or) loss of data. Then, that value has to be explicitly type casted.

ex:

```
byte b;
int c = 4;
b = (byte) c;
```

```
float a = 4.567;
int b;
b = (int) a;
```

Type promotion rules

① All byte and short values are promoted to int.

ex:
```
byte b = 50;
b = b * 2;   // Error! cannot assign an int to a byte.
```

while evaluating 50*2, the value is automatically promoted to int. Now result is in int type. so. we cannot assign

```
byte b = 50;
b = (byte) (b*2);   ✓ which gives '100'.
```

② The resultant data-type of an expr is always the highest data-type of an operand in the expr. i.e

int + float → float

Example:
```
class promote
{
  public static void main (String args[])
  {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = 0.1234;

    double result;
    result = (f * b) + (i / c) - (d * s);
    System.out.println("result = " + result);
  }
}
```

(f*b) → promoted to float
(i/c) → promoted int;
(d*s) → promoted to double.
finally result is in double.

operators

① Arithmetic : +, -, *, /, %, ++, --, +=, -=, *=, /=, %=,

modulus (%) → can be applied to floating point numbers.

int x = 42; float f = 42.25;

x % 10 = 2; f % 10 = 2.25;

② Relational : <, >, <=, >=, ==, !=

The relational operators determine the relationship that one operand has to the other. The result produced by a relational operator is a boolean value.

③ Logical : &&, ||, !

④ Bitwise : &, |, ^, ~, <<, >>, >>>

Bitwise unary NOT

Bitwise exor

shift right zero fill.

⑤ Conditional : ?:

⑥ Assignment : =

Right shift with zero fill (>>>) : It fills the leftmost bits with 0's instead of the previous high bit values. It is called as unsigned shift operator and works for 32 & 64 bits.

Eg:

-8: 11111111 11111111 11111111 11111000

(Right shift) >> : 11111111 11111111 11111111 11111110

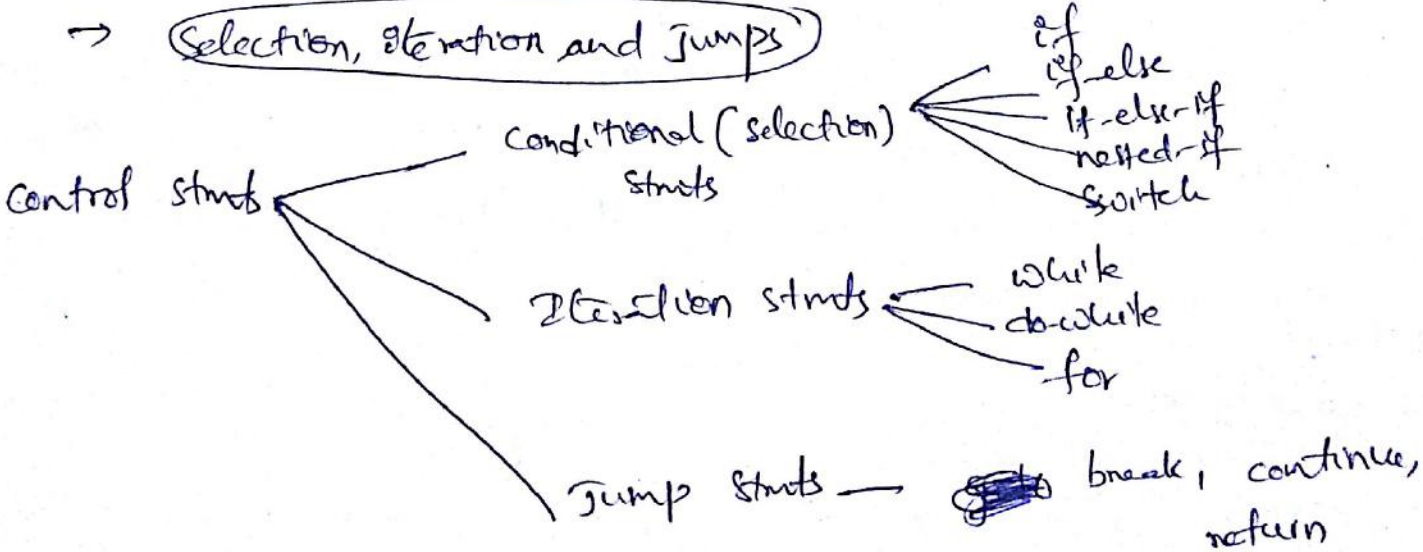
>>> : 01111111 11111111 11111111 11111110

Operator precedence

Highest
( ) [ ]
++ -- ~ !
* / %
>> >>> <<
> >= < <=
== !=
&
^
|
&&
||
? :
= op=
Lowest

Control statements: It causes the flow of execution to advance and branch based on the condition.

→ (Selection, Iteration and Jumps)



classes and objects

class defines a new data type. It is used to create objects of that type. A class is a *template* for an object, and object is an *instance of* a class.

General form:

```
class classname
{
    type instance_variable1;
    type instance_variable2;
    . . .
    type methodname1 (parameter-list)
    {
        ==
    }
    type methodname2 (parameter-list)
    {
        ==
    }
}
```

The data, or variables, defined within a class are called *instance variables*. The code is contained within methods. Collectively, the methods and variables defined within a class are called *members* of the class.

Variables defined within a class are called instance variables because each instance of the class (object) contains its own copy of those variables. The data for one object is separate and unique from the data for another.

Example

Template

```
class Box
{
    double w;   // Instance variables.
    double h;
    double d;
}
```

Creating an object

```
Box b1 = new Box();
```

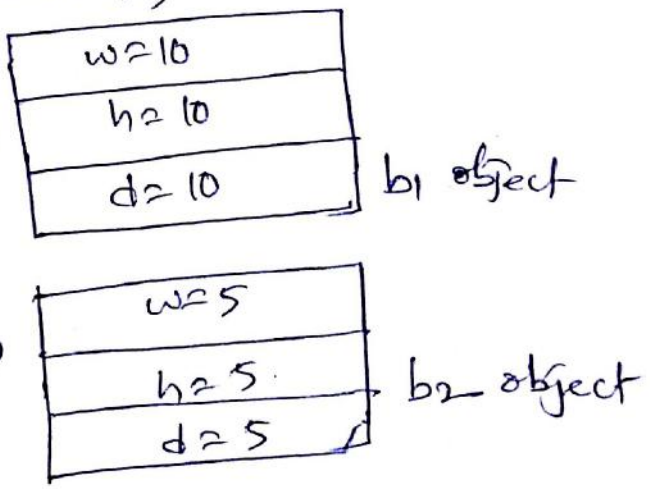
```
Box b1;
b1 = new Box();
```

b1 is an instance of class Box. It contains its own copy of each instance variable defined by the class. To access these variables, we have to use dot (.) operator.

Ex: `b1.w = 100;`

Example

```
class Box
{
    double w, h, d;
};
class BoxDemo
{
 public static void main (String args[])
 {
    Box b1 = new Box();
    Box b2 = new Box();
    b1.w = 10;
    b1.h = 10;
    b1.d = 10;
    b2.w = 5;
    b2.h = 5;
    b2.d = 5;
    double v1, v2;
    v1 = b1.w * b1.h * b1.d;
    v2 = b2.w * b2.h * b2.d;
   System.out.println("volume1 = " + v1);
   System.out.println("volume2 = " + v2);
 }
}
```



Introducing methods

class → { instance variables + methods }

General form

```
type name (parameter-list)
{
    // body of method
}
```

Example:

```
class Box
{
    double w, h, d;
    void volume()
    {
        System.out.println("volume is: " + w*h*d);
    }
}

class BoxDemo1
{
    public static void main(String args[])
    {
        Box b1 = new Box();
        Box b2 = new Box();
        b1.w = 10; b1.h = 20; b1.d = 15;
        b2.w = 5; b2.h = 10; b2.d = 12;
        b1.volume();
        b2.volume();
    }
}
```

→ The call to b1.volume() displays the volume of the box defined by object b1 i.e. "volume is : 3000".

→ The call to b2.volume() displays the volume of the box defined by object b2 i.e. "volume is : 600".

→ the methods can be in any form like no args-no return values, with args-no return values, with args-with return values, no args-with return values

// methods

```
class Box {
  double w,h,d;
  void read (double w1, double h1, double d1)
  {
      w=w1;
      h=h1;
      d=d1;
  }
  double volume ()
  {
      return w*h*d;
  }
}

class BoxDemo2
{
  public static void main (String args[])
  {
      Box b1 = new Box();
      Box b2 = new Box();
      double vol1, vol2;
      b1.read (10, 20, 15);
      b2.read (5, 10, 12);
      vol1 = b1.volume();
      vol2 = b2.volume();
      System.out.println("volume1 = " + vol1);
      System.out.println("volume2 = " + vol2);
  }
}
```

Note: instance variables (w, h, d) and parameters in the read() [w1, h1, d1] are different.

Constructors

Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor. A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntatically similar to a method. once defined, it is automatically called immediately after the object is created, before the 'new' operator completes. It l.

Constructors have no return type, not even void because the implicit return type of a class constructor is the class type itself.

example

```
class Box
{
    double w,h,d;
    Box(double w1, double h1, double d1)
    {
        w=w1;
        h=h1;
        d=d1;
    }
    double volume()
    {
        return w*h*d;
    }
}
class BoxDemo3
{
    public static void main(String args[])
    {
        Box b1= new Box(10,20,15);
        Box b2= new Box(5,10,12);
        double vol1, vol2;
        b1.volume(); b2.volume();
        System.out.println("volume1 is:"+vol1);
    } } System.out.println("volume2 is="+vol2);
```

// parameterized constructor for Box.

In the above program, two objects b_1 and b_2 are created with the values (10,20,15) and (5,10,12). are initialized to (0,0,0). This is done automatically at the time of creation of an object.

Default constructor

```
class Test
{
    int x;
    void show()
    {
        system.out.println(x);
    }
}
class TestDemo
{
    public static void main(String args[])
    {
        Test t1 = new Test();
        t1.show();
    }
}
```

In the above program, constructor is not written by the user. The program gets executed without errors. because whenever the user does not write a constructor, the compiler by default provides a constructor without parameters, called as default constructor.

So, the output of the above program is '0'.

Note: The default constructor will be given by the compiler which will initialize all the data members of the class to '0'

Constructor overloading [polymorphism]

→ If a class has more than one constructor, it is called constructor overloading. The compiler would invoke the appropriate constructor depending on number of parameters.

→ If two constructors have same no of parameters then the data type of the parameters will be considered.

→ If the no of parameters is same and also data types of parameters are compatible, then the order of parameters will be considered.

Example:

```
class Box
{
    double l, w, h, d;
    Box ()
    {
        w=10; h=10; d=10;
    }
    Box (int x, int y, int z)
    {
        w=x; h=y; d=z;
    }
    double volume ()
    {
        return w*h*d;
    }
}
class BoxDemo4
{
    public static void main (String args[])
    {
        Box b1 = new Box ();
        Box b2 = new Box (5, 5, 5);
        double vol1 = b1.volume ();
        double vol2 = b2.volume ();
```

```
        System.out.println("volume1 is:" + vol1);
        System.out.println("volume2 is:" + vol2);
    }
}
```

In the above program, class Box has two constructors, one without parameters and one with three parameters. Java compiler provides a default constructor only when the user has not written any constructor in the class with or without parameters.

So, for object b_1, first constructor is called and initializes $w=10, h=10, d=10$. So vol1 is 1000.

for object b_2, second constructor is called and initializes $w=5, h=5, d=5$. So vol2 is 125

'this' keyword :

```
class Box
{
    double w, h, d;          // instance variables of class

    Box()
    {
        w=10; h=10; d=10;
    }
    Box(double w, double h, double d)   // parameters
    {
        w=w; h=h; d=d;
    }
    double volume()
    {
        return w*h*d;
    }
}
```

parameters and instance variables are same

```
class BoxDemo5
{
  public static void main (String args[])
  {
     Box b1 = new Box();
     Box b2 = new Box(1,2,3);
     Box b3 = new Box (4,5,6);

     double vol1, vol2, vol3;
       vol1 = b1.volume();
       vol2 = b2.volume();
       vol3 = b3.volume();
       System.out.println("vol1 +" "+vol2+" "+ vol3);
  }
}
```

In the above program, the parameterized constructor is using the same names for local parameters as the instance variables of the class Box. So, the statements

```
w=w;
h=h;
d=d;
```

causes an ambiguity (or error). To avoid this so called ambiguity 'hiding of instance variables by the local parameters', we use 'this' keyword.

→ "this" points to current object.

Rewrite the parameterized constructor as

```
Box(double w, double h, double d)
{
   this.w = w;
   this.h = h;
   this.d = d;
}
```

Now, when the stmt Box b_2 = new Box (1,2,3) is getting executed, then 'this' keyword in the constructor is pointing to the object b_2 (current object). So, for object b_2, w=1, h=2, d=3

Similarly, Box b_3 = new Box (4,5,6), 'this' keyword in the constructor is pointing to the object b_3 (current object).

method overloading / Form 2 of polymorphism

```
class Test
{
    void display()
    {
        System.out.println("Hi");
    }
    void display(int x)
    {
        System.out.println("x = " + x);
    }
}
class methodOverDemo
{
    public static void main(String args[])
    {
        Test ob1 = new Test();
        ob1.display();
        ob1.display(100);
    }
}
```

Here, class Test consists of two methods with the same name 'display'. The compiler will differentiate these two methods depending on the parameters list.

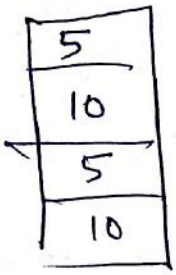
ob1.display() would invoke the 'display' method without parameters and the output will be 'Hi'

ob1.display(100) would invoke the 'display' method with one parameter and the output will be '100'.

Passing Objects in a constructor

```
class A
{
    int x;
    A()
    {
        x = 5;
    }
    A(int a)
    {
        x = a;
    }
    A(A ob1)
    {
        x = ob1.x;
    }
    void display()
    {
        System.out.println("x = " + x);
    }
}

class ConstDemo
{
    public static void main(String args[])
    {
        A ob1 = new A();
        A ob2 = new A(10);
        A ob3 = new A(ob1);
        A ob4 = new A(ob2);
        ob1.display();
        ob2.display();
        ob3.display();
        ob4.display();
    }
}
```



Here, when the object ob3 is created, the object ob1 is passed as an argument. This argument is stored on the formal parameter ob in the third construct.

The stmt "x = ob.x" implicitly ob3.x = ob1.x i.e the object ob3 is initialized with the same values of the object ob1. similarly, the creation of ob4 initializes ob4 with the values of the object ob2.

Returning of objects

```
class A
{
    int x;
    A () { x=5;}
    A ( int a) { x = a;}
    A fun ( )
    {
        A ob = new A (15);
        return ob;
    }
}
class Returnobject
{
    public static void main ( String args[])
    {
        A ob1= new A ();
        A ob2;
        ob2 = ob1.fun();
        System.out.println( ob1.x + "  "+ob2.x);
    }
}
```

Here, the method 'fun' of class A is creating an object 'ob' and is returning that object. The method 'fun' returns an object of the type class 'A', so the return type of that method should be class 'A'.

'fun' is called by using an object ob1 which invokes the method. the returned object ob of type class 'A' must be stored in the same type because the returned value, the object is to be stored in a reference variable of the same type i.e

ob2 = ob1.fun();

Command-Line Arguments

A command line argument is the information that directly follows the program's name on the command line when it is executed. These values are stored as String in the String array passed to main() in Java.

Example

```
class CommandLine
{
    public static void main ( String args[])
    {
        for( int i=0; i<args.length; i++)
            System.out.println(" args[" + i + "] : " + args[i]);
    }
}
```

→ javac CommandLine.java
→ java CommandLine this is a test 100 -1

o/p: args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1

```
class cmda
{
    public static void main ( String args[ ])
    {
        string s = a[0];
        System.out.println (s);
        int i = Integer.parseInt ( a[1]);
        System.out.println ( i+s);
        double d= Double.parseDouble (a[2]);
        System.out.println ( d);
    }
}
```

→ To convert a value into an Integer type, we use the static Integer.parseInt (value); parseInt() method is a static method of class Integer. Similarly parseDouble() method is a static method of class Double.

call by value & call by reference

① call by value

```
class A
{ int x;
  A (int l)
  { x=l;
  }
  void change (int z)
  { z=100;
  }
}
class CallByValue
{
p. s. v. m ( string args[])
{
 A ob=new A(10);
 System.out.println("Before:"+ob.x);
  ob.change (ob.x);
  Sy. o. pln ( After: " + ob.x);
}
}
```

② Call by reference

```
class B
{ int x;
  B ( int l)
  { x=l;
  }
  void change ( B ob)
  { ob.x=100;
  }
}
class CallByRef
{
p. s. v. m ( String args[])
{
  B ob1= new B (15);
  S. o. pln ( " Before= " + ob1.x);
  ob1. change (ob1);
  S. o. pln ("After = " + ob1.x);
}
}
```

Recursion

```
class Factorial
{ int fact (int n)
  { if (n==1)
    return 1;
    else
    return n*fact (n-1);
  }
}
```

```
class FactDemo
{
  p. s. v. m ( String args[])
  {
    Factorial f= new Factorial();
    S. o. pln ( " f. fact (5));
  }
}
```

Access control

Access specifier determines how a member can be accessed and modified. Java supports a rich set of access specifiers. They are

① private ② protected ③ public ④ No access modifier (default)

→ when a member of a class is modified by the public specifier, then that members can be accessed by any other code. when a member of a class is specified as private, then that member can only be accessed by other members of its class.

→ default access specifier is public.

→ Private data cannot be accessed outside the class in which it is defined i.e it cannot be accessed directly even through objects. only the methods of the same class can access the private data directly. Mostly, the data members are private and member functions are public.

→ protected is used in inheritance to give access to the child class.

→ public modifier allows the access of the members outside the class.

→ The method main() is defaut public, so that it can be called from other environment (terminal).

example

```
class Test
{ int a;    // Default access is public
  public int b;   // public access
  private int c;  // private access.
  void read( int i)
    {  c = i;
    }
  int display ()
    {  return c;
  } }
```

```
class TestDemo
{ public static void main(String args[])
  {
      Test ob = new Test();
      ob.a = 10;  // a & b may be accessed directly.
      ob.b = 20;
      // ob.c = 100  ⇒ gives an error. [cannot be accessed directly]
      ob.read(100);
      System.out.println("a = " + ob.a + "b = " + ob.b);
      System.out.println("c = " + ob.display());
  }
}
```

Note: To access the private variable 'c', we need to use the methods read() and display() of class Test.

"Static" keyword:

Generally to access the data members of a class we need an object. To access without an object (or) to access before an object is created, we use the modifier "static" at declaration. 'static' can be applied to both variables and methods.

Static methods can:

(a) access only static methods and static variables.
(b) cannot refer to 'this' or 'super' in any way.
(c) → static variables don't have a seperate copy for objects like instance variables [seperate copy for each object].
→ Each object can use the static variable and behave as ⇒ global variables.
→ static variables cannot be initialised using objects so they are initialized either directly (or) using a static block.

→ when we run a program

(A) static variables will be initialized } (if any)
(B) static block will be executed
(C) main() method will be invoked

→ static methods called outside the class in which they are defined should be invoked by using the class name.

Ex: Math.sqrt();

Examples

eg1)

```
class Test
{
    int x;
  static void read()
   {  x = 10;
   }
  static void show()
  {
     System.out.println("x="+x);
  }
 public static void main(String args[])
  {
        read();   // without object & without classname
        show();
    }
}
```

eg2)

```
class Test1
{
    int x;
    static void read()
    {  x=20;
    }
    static void display()
    {
      System.out.println("x="+x);
    }
}
```

```
class Test1Demo
{
 p.s.v.m(String args[])
  {
     Test1.read();
     Test1.display();   // with the classnam
  }
}
```

// Static Keyword

prog3.

```
class Test2
{
    static int a = 3;
    static int b;
    static void meth (int x){
    {
        System.out.println ("x = "+ x);
        system.out.println ("a = "+a);
        system.out.println ("b = "+b);
    }
    static
    { system.out.println ("static block initialized.");
        b = a*4;
    }
    public static void main (String args[])
    {
        meth(20);
    }
}
```

→ As soon as the Test2 class is loaded, all the static stmts are run. First 'a' is set to '3', then the static block executes the printing msg and b is initialized to '12' [i.e a*4] then main() method is called, which calls meth(), passing 20 to x. It gives

```
static block initialized.
x = 20
a = 3
b = 12
```

Note: It is illegal to refer any instance variables inside of a static method.

Nested and Inner classes:

A class defined within another class is called as nested class.
A nested class has access to the members [including private] of the class in which it is nested. But the enclosing class does not have access to the members of the nested class.

There are two types of nested classes: static and non-static

If it is static nested class, it must access the members of its enclosing class through an object i.e it cannot refer to the members of its enclosing (outer) class directly.

If it is non static nested class then it is also called as inner class. It has access to all of the variables and methods of its outer class and may refer to them directly.

Example: The scope of a inner class is always within the outer class.

```
class Outer
{
    int x = 100;
    class Inner
    {
        void display()
        {
            System.out.println("x=" + x);
        }
    }
    void test()
    {
        Inner in = new Inner();
        in.display();
    }
}
class InnerClassDemo
{
    public static void main(String args[])
    {
        Outer ob = new Outer();
        ob.test();
    }
}
```

If we want to access inner class members through outer class object then

① create an object of outer class.

② create an object of inner class in outer class method and then call inner class members using its object.

③ call the method in which inner class object is created using outer class object.

creation of an object of inner class using outer class object

① creating object of outer class

```
outer out = new outer();
```

② creating object of inner class

```
outer.inner in = out.new inner()
```

- outer class name
- inner class name
- inner class object
- outer class object
- inner class name

Example

```
class Outer
{
    int x=100;
    void show()
    {
        S.O.P(x);
        S.O.P("hello");
    }
    class Inner
    {
        int y=200;
        void display()
        {
            S.O.P(x);
            S.O.P(y);
        }
    }
}

class InnerDemo
{
    P.S.V.main(String args[])
    {
        Outer out = new Outer();
        out.show();
        Outer.Inner in = out.new Inner();
        in.display();
    }
}
```

o/p

```
100
hello
100
200
```

Arrays

→ An array is a group of contiguous (or) related data items that share a common name.

→ All the elements of an array share the same name with a different subscript (or) index.

→ Index starts from zero. In Java arrays are objects. An array class is predefined in the language itself.

Creating an array:

1. Declare the array
2. create memory locations
3. put values into the memory locations.

① Declaration:

Syntax: `type arrayname[]` (or)

`type [] arrayname`

Ex: `int number[]; float [] avg;`

→ we donot enter size of the arrays in the declaration.

② Creation:

Syntax: `arrayname = new type [size];`

Ex: `number = new int[5];`

`avg = new float[10];`

`int number[] = new int[5];` declaration + creation.

③ Initialization

`arrayname [subscript] = value;` ⇒ `number[0] = 1, number[1] = 2`

(or)

`type arrayname [] = { list of values };`

Unlike C language, Java protects arrays from overruns and underruns. Trying to access an array beyond its boundaries will generate an error message.

```
int a[] = {1,2,3,4,5};
int b[];
b = a;
```
} valid

Array length : In Java, all arrays store the allocated size in a variable named length.

Ex: `int asize = a.length.`

Two-Dimensional arrays

Syntax: `type arrayname[][] = new type [rowsize][columnsize];`

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

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

```
int a[][] = {1,2,3,4}
```

(a)

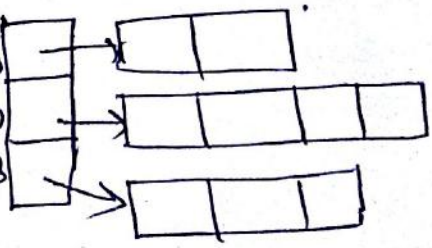
```
int a[][] = {{1,2},{3,4}};
```

(b)

```
int a[][] = {{1,2},
             {3,4}
            };
```

Variable size arrays

```
int x[][] = new int [3][];
x[0] = new int[2];
x[1] = new int [4];
x[2] = new int [3];
```



Strings

→ Java implements strings as objects of type **String** [string class]

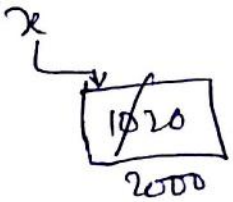
→ String & String Buffer are the classes from java.lang package with which we can manipulate strings.

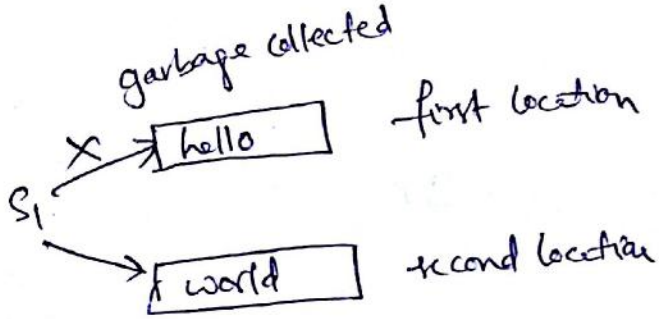
→ Every string you create is actually an object of type String class ~~class~~ and these objects can store string constants ("group of characters")

→ Objects of type string are immutable: once a string object is created, its contents cannot be altered.

Ex:

```
int x = 10;
S.O.P(x); // 10
  x = 20;
S.O.P(x); // 20
String S1 = "hello"
  S.O.P(S1); // hello
String S1 = "world"
  S.O.P(S1); // world
```





→ whenever a string value is changed the new value is stored in another location and old location is garbage collected which is called immutable nature.

Creation of string object with String class

① String S = "CSE";

② String S = new String ("CSE");

③ String S1 = new String (S);

Here, one object of string is created and assign string constant 'CSE' to object of string class i.e 'S'.
(string literal)

Java defines one operator for string objects: + [concatenation of string]

Ex: String s = "Hello" + "world";

s.o.p(s); ⇒ Helloworld.

→ String class cannot be extended because it is defined as final.

The string constructors

① → To create an empty String, call the default constructor.

Ex: String s = new String();

It will create an instance of String with no characters in it.

② To create a string initialized by an array of characters, use the constructor

String (char chars[]);

Ex: char ch[] = { 'a', 'b', 'c' };

String s = new String(ch);

It initializes 's' with the string constant 'abc'

③ To specify a subrange of a character array as an initializer use

String (char chars[], int startIndex, int numchars);

Ex: char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f', };

String s = new String (chars, 2, 3);

This initializes 's' with the characters cde.

④ To construct a String object that contains the same character sequence as another String use

String (String strObj);

(8) The string class provides constructors that initialize a string when given a byte array.

Syntax: String (byte asciichars[]);

String (byte asciichars[], int startIndex, int numchars);

String Literals

String s = "abc";

s.o.p (s.length()); // → 3

s.o.p ("abc".length()); // → 3

String Concatenation

String age = "19"; (or) int age = 19;

String s = "He is " + age + "years old";

s.o.p(s) ⇒ He is 19 years old.

String s = "four: " + 2 + 2;

s.o.p(s) ⇒ four: 22

String handling methods

(1) The length of a string is the number of characters that it contains.

int length();

String S_1 = "CSE";

int l_1 = S_1.length(); ⇒ $l_1 = 3$

char ch = {'a', 'b', 'c', 'd'};

String S_2 = new String (ch);

s.o.pln(S_2.length) ⇒ 4

② charAt(): It returns the character at the position specified by the variable index.

Syntax: `char charAt(int index);` → To extract one character

Ex: char ch;
ch = "abc".charAt(1) ⇒ returns b, `ch = 'b'`

③ getChars(): It is used to extract more than one character at a time.

Syntax:
```
void getChars(int sourceStart, int sourceEnd, char target[],
              int targetStart);
```

Ex: String s1 = "welcome to java programming";
int start = 5;
int end = 10;
char ch[] = new char[end - start];
s1.getChars(start, end, ch, 0);
S.o.p(ch) ⇒ 'metoj'

String comparison

① equals(): To compare two strings for equality use

boolean equals(Object str);

str → It is string object being compared with the invoking string object. It returns true if the strings contain the same characters in the same order, and false otherwise. It is case-sensitive.

→ To ignore case differences, use equalsIgnoreCase

(ii) equalsIgnoreCase()

Syntax
boolean equalsIgnoreCase(String str);

Ex:
```
string s1 = "welcome";
string s2 = "welcome";
string s3 = "thank you";
string s4 = "WELCOME";
S.O.P( s1.equals(s2)); ⇒ true
S.O.P( s1.equals(s3)); ⇒ false
S.O.P( s1.equals(s4)); ⇒ false
S.O.P( s1.equalsIgnoreCase(s4)); ⇒ true.
```

① compareTO() :

For sorting applications, you need to know which is lessthan, or greaterthan (or) equal to the next.

lessthan → It comes before } according to dictionary order
greaterthan → It comes after

Syntax: `int compareTo(String str);` It returns

value	meaning
① lessthan Zero	→ The invoking string is lessthan str.
② Greaterthan Zero	→ the invoking string is greaterthan str.
③ Zero	→ The two strings are equal.

```
String s1 = "world";
String s2 = "Hello";
S.O.P( s1.compareTo(s2)); >0
S.O.P( s2.compareTo(s1)); <0
S.O.P (s1.compareTO(s1)); =0
```

② regionMatches()

boolean regionMatches (int startIndex, String str2, int str2-start index, int numchars);

(3) startsWith() and endsWith() :

Syntax:
```
boolean startsWith(String str);
boolean endsWith(String str);
```

To determine a given string begins with or ends with a specified string.

(4) equals() vs == :

equals() → compares the characters inside a string object.

== operator compares two object references to see whether they refer to the same instance.

Searching Strings

① indexOf() → Searches for the first occurrence of a character or substring

② lastIndexOf(): searches for the last occurrence of a character or substring.

⇒ returns the index at which the character or substring was found, or -1 on failure.

Syntax:
```
int indexOf(char ch);
int lastIndexOf(char ch);
```

ex String s = "welcome to cse";

int a, b;

a = indexOf('c') ⇒ 3

b = lastIndexOf('c') ⇒ 9.

modifying a String

① substring(): It returns a string from given begin index to last index on the existing string.

String substring (int startIndex);
String substring (int startIndex, int endIndex);

Ex: string s_1 = "welcome";
string s_2 = s_1.substring (3); // come
String s_3 = s_1.substring (3,5); co

② concat(): It concatenates given string with existing string.

String s_1 = "welcome";
String s_2 = s_1.concat ("to java"); // welcome to java.

③ replace(): It replaces old character with new character.

④ trim(): It removes white spaces from both ends of existing string.

changing the case of characters (lower ⇄ Upper)

String toLowerCase();
String toUpperCase();

String s_1 = "WELCOME";
String s_2 = toLowerCase();

String s_1 = "welcome";
String s_2 = toUpperCase();

Inheritance:

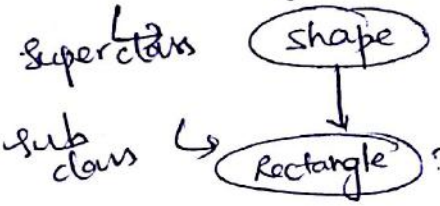
In Java, the code reusability is done in the following way:

code reusability → ① Composition
→ ② Inheritance

① we can create object of one class in another ~~sub~~ class known as composition (or) containment.

② Inheritance (or) "is a" relationship:

This relationship allows "creating of one class from another existing class":



This is create from the class shape.

→ It allows the creation of hierarchical classifications. A class that is inherited is called as superclass and the class that does the inheriting is called a subclass

Example:

Superclass

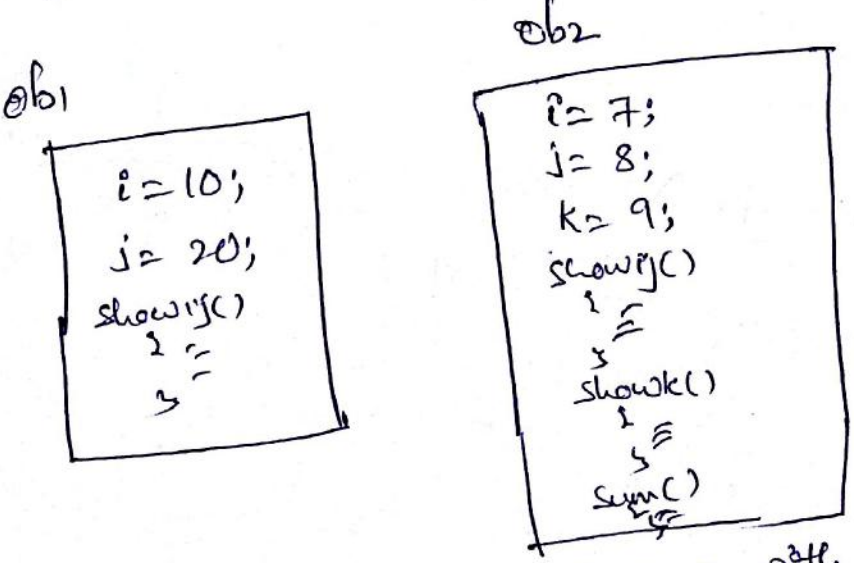
```
class A
{
  int i, j;
  void showij()
  {
    System.out.println("i and j: "+i+" "+j);
  }
}
```

Subclass

```
class B extends A
{
  int k;
  void showk()
  {
    System.out.println("k: "+k);
  }
  void sum()
  {
    System.out.println("i+j+k: "+(i+j+k));
  }
}
```

```
class InheritanceDemo
{
    public static void main ( String args[])
    {
        A   ob1 =  new  A();  // superclass object ob1
        B   ob2 =  new  B();  // subclass object ob2

        ob1.i = 10;
        ob1.j = 20;
    System.out.println(" Content of superclass object ob1 :");
        ob1.showij();
        ob2.i = 7;
        ob2.j = 8;
        ob2.k = 9;
    System.out.println(" content of subclass object ob2 :");
        ob2.showij();
        ob2.showk();
    System.out.println(" sum of i, j and k is :");
        ob2.sum();
    }
}
```



→ extends is a keyword in Java with which we can link two classes. By using sub ~~super~~ class object, we can call superclass variables and methods but superclass object cannot call subclass variables and methods.

member access and inheritance:

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.

Example:

```
class A
{
    int i; // => public
    private int j; // private
    void read(int x, int y)
    {
        i = x;
        j = y;
    }
}
```

Super class

```
class B extends A
{
    int total;
    void sum()
    {
        total = i+j; // error, j is not accessible.
    }
}
```

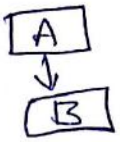
Sub class.

```
class AccessDemo
{
    p.s.v.m(String args[])
    {
        B ob2 = new B();
        A ob1 = new A();
        ob1.read(10,20);
        ob2.read(20,30);
        s.o.p("total:" + ob2.total);
    }
}
```

Note: A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

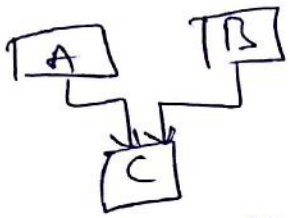
Types of inheritance

① single | simple inheritance :



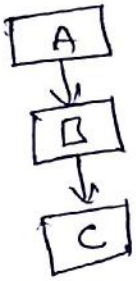
(derivation of subclass from only one superclass)

✗ ② Multiple : Derivation of one class from two (or) more superclass.



Java does not support multiple inheritance directly but it can be implemented using interfaces.

③ multilevel : Derivation of a class from another derived class.

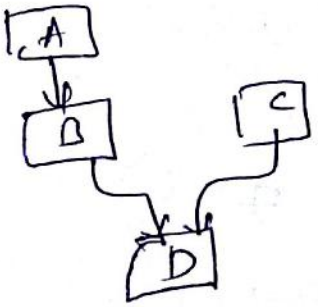


④ Hierarchical : Derivation of several class from single superclass.

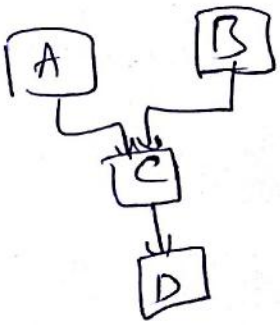
→ super

→ subclasses.

✗ ⑤ Hybrid : Derivation of a class involving more than one form of inheritance.



(or)



~~Method overriding~~ Constructors

↳ Constructors are called in order of derivation from superclass to subclass. Whenever, a child class object is created, child class constructor is invoked. The first stmt in the child class constructor (either default or parameterised) will always be a call to the default constructor of the immediate parent (if it exists).

```
class A
{
    A()
    {
        System.out.println("Inside A's Constructor.");
    }
}
class B extends A
{
    B()
    {
        System.out.println("Inside B's constructor.");
    }
}
class C extends B
{
    C()
    {
        System.out.println("Inside C's constructor.");
    }
}
class CallingConsDemo
{
    public static void main(String args[])
    {
        C ob = new C();
    }
}
```

eg: Inside A's constructor.
Inside B's constructor.
Inside C's constructor.

Example2:

```
class S
{
    S()
    {
        System.out.println("Superclass default constructor");
    }
    S(int l)
    {
        System.out.println("l=" + l);
    }
}

class D extends S
{
    D()
    {
        // calls S();
        System.out.println("Derived class default cons");
    }
    D(int x)
    {
        // calls S()
        System.out.println("x=" + x);
    }
}

class InheritDemo
{
    public static void main(String args[])
    {
        D ob1 = new D();
        D ob2 = new D(20);
    }
}
```

o/p:

Superclass default constructor
Derived class "
Superclass default constructor
x = 20

"super" Keyword: [method overriding]

It is one reference that refers to superclass (s) Superclass object.

(a) Accessing super class datamembers from its subclass

Syntax: super. Superclass datamembername;

(b) Accessing super class methods

Syntax: super. Superclass methodname();

(c) Accessing superclass constructors (parameterized) from its subclass:

super (parameter list); — This stmt should always be the first stmt in the derived class.

Example

```
class Test
{
    int x=100;
    public void display();
    {
        s.o.p ("Hello1");
    }
}
```

```
class Demo extends Test
{
    int x=200;
    public void show()
    {
        s.o.p( super.x);
        super.display();
    }
    public void display()
    {
        s.o.p ("Hello2");
    }
}
```

```
class SuperDemo
{
    p.s.v.m (String args[])
    {
        Demo d1 = new Demo();
        d1.display();
        d1.show();
        s.o.p( d1.x);
        s.o.p( super.x) // error.
        super.display() // error.
    }
}
```

Example2:

```
class S
{
    S()
    {
      System.out.println("default superclass constructor");
    }
     S(int l)
     {
       System.out.println("l=" + l);
     }
}
class D extends S
{
        D()
        {
          //Calls S();
          System.out.println("default derived class cons");
        }
        D(int x)
        {
          super(50);
          System.out.println("x=" + x);
        }
}
class InheritDemo
{
  public static void main(String args[])
  {
        D ob1 = new D();
        D ob2 = new D(100);
  }
}
```

o/p : Default superclass constructor.

~~default derived~~

default derived class cons

l = 50

x = 100

'super'

→ To access the private data in a derived class constructor "super" is used.

→ when the data member names in the child class & super class are same, if we try to access it always considers the child class data member which is called "hiding of superclass instance variable by derived class instance variables"

→ To access the superclass instance variable in the derived class, we use the keyword "super".

Method overriding [Run-time polymorphism]

It occurs when the superclass and subclass are having same methods (same method signature) and subclass hides the superclass methods. This is a form of polymorphism.

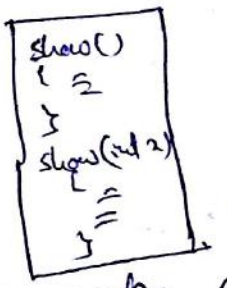
To access superclass ~~met~~ method in child class; we use

super.methodname()

```
class A
{
public void show()
    {
        System.out.println("In Superclass");
    }
}
class B extends A
{
    public void show()
    {
        System.out.println("In subclass");
    }
}
class MethOverideDemo
{
    public static void main(String args[])
    {
        A ob1 = new A();
        ob1.show();
        B ob2 = new B();
        ob2.show();
    }
}
```

Example : Method overloading (Method overriding in Inheritance)

```
class A
{
    void show()
    {
        System.out.println("In Superclass");
    }
}
class B extends A
{
    void show(int x)
    {
        System.out.println("x = "+x);
    }
}
class Methodover
{
    public static void main(String args[])
    {
        B ob2 = new B();
        ob2.show();
        ob2.show(5);
    }
}
```



Here, class B gets a copy of show() without parameters, as B is the subclass of A and B defines show(int x) with one integer parameter. So, B contains two methods with the same name 'show' and differing in no of parameters. So, this is a form of method overloading.

"final" keyword

The keyword 'final' can be applied to variables, methods & classes.

'final' applied to a variable becomes a constant i.e cannot changed.

'final' method cannot be overridden.

'final' class cannot be inherited.

A superclass ref variable can refer to subclass object, when it refers it is the type of the ref variable but not the type of object that determines which members can be accessed. It can refer only to the members of its own class but not to members defined by subclass.

→ // A superclass reference variable can refer to a subclass object.

```
class Box
{ double h,w,d;
  Box()
  { h=1; w=1; d=1;
  }
  Box(double a, double b, double c)
  { h=a; w=b; d=c;
  }
  void volume()
  { System.out.println("volume=" + (h*w*d));
  }
}
class BoxColor extends Box
{ String color;
  BoxColor(double p, double q, double r, String s)
  { h=p; w=q, d=r;
    color = s;
  }
}
class SuperRefDemo
{ public static void main(String args[])
  { Box b1 = new Box();
    b1.volume();
    BoxColor b2 = new BoxColor(1,2,3,"Red");
    b2.volume();
    Box b3;
    b3 = b2;
    b3.volume();
    System.out.println("color=" + b3.color);
    //System.out.println("color2" + b3.color); // error.
  }
}
```

Dynamic method Dispatch (Run-time polymorphism)

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run-time, rather than compile-time. It is known as run-time polymorphism.

When an overridden method is called through a superclass ref, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. This determination is made at runtime, so it called as dynamic binding.

Ex:

```
class A
{
    void call()
    {
        S.o.p("Inside A's callsmethod");
    }
}
class B extends A
{
    void call()
    {
        S.o.p("Inside B's call() method");
    }
}
class C extends A
{
    void call()
    {
        S.o.p("Inside C's call() method");
    }
}
class DMDDemo
{
    p.s.v.m(String args[])
    {
        A a = new A();   a.call();
        B b = new B();   b.call();
        C c = new C();   c.call();

        A r;
        r = a; // r refer to an A's object.
        r.call(); // calls A's call() method
        r = b; // r refers to B's object
        r.call(); // calls B's call() method
        r = c; // refer to C's object
        r.call(); // calls C's call() method
    }
}
```

The object class:

There is one special class, 'object', in Java. All the other classes are subclasses of 'object'. i.e object is a superclass of all other classes. This means that a reference variable of type object can refer to an object of any other class.

	Methods	Purpose
①	object clone()	→ creates a new object that is same as the object being cloned.
②	boolean equals(object object)	→ Determines whether one object is equal to another.
③	void finalize()	→ called before an unused object is recycled.
④	class getclass()	→ obtains the class of an object at run-time
⑤	int hashCode()	→ returns the hash code associated with invoking object
⑥	void notify()	→ resumes execution of a thread waiting on the invoking object.
⑦	void notifyAll()	→ resumes execution of all threads waiting.
⑧	string to string()	– returns a string that describes the object.
⑨	void wait() void wait(long millisecs) void wait(long millisecs, int nanosecs)	} waits on another thread of execution.

Abstract classes: sometimes, superclass gives only a generalized form that will be shared by all of its subclasses, leaving it to subclasses to fill in the details. i.e. superclass determines the nature of the methods that the subclass must implement.

→ Methods without body are called abstract methods.

→ Method Name should be preceed by the keyword 'abstract'

Syntax: abstract <return-type> <method-name> (parameter-list);

Any class that contains one (or) more abstract methods must be declared abstract. To declare a class abstract, use the 'abstract' keyword in front of the class keyword.

Syntax:

```
abstract class <class-name>
{
    abstract <return-type> methodname(<params>)
    {
    =
    }
    =
}
```

↔ There can be no objects of an abstract class. i.e abstract classes cannot be directly instantiated with the new operator.

→ Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared as abstract. The abstract class can have both abstract and concrete methods. ~~The Abstract~~ It cannot have static variables and main(). It can have constructors.

Example

```
abstract class A
{
    abstract void call();
    void callme()
    {
        S.o.p("This is a concrete method");
    }
}
class B extends A
{
    void call()
    {
        S.o.p("B's implementation of call()");
    }
}
class AbstractDemo
{
    P.S.V.main(String args[])
    {
        B b = new B();
        b.call();
        b.callme();
    }
}
```

Abstract classes

```
abstract class Figure
{
    double d1, d2;
    Figure ( double a, double b)
    {
        d1 = a; d2 = b;
    }
    abstract double area();
}
class Rectangle extends Figure
{
    Rectangle ( double x, double y)
    {
        super (x, y);
    }
    double area()
    {
        return d1 * d2;
    }
}
class Triangle extends Figure
{
    Triangle (double p, double q)
    {
        super (p, q);
    }
    double area()
    {
        return (d1*d2)/2;
    }
}
class Abstract Demo
{
    p.s.v.m (String args[])
    {
        Rectangle r = new Rectangle (4,5);
        r.area();
        Triangle t = new Triangle (6,7);
        t.area();
    }
}
```

partial implementation of abstract classes

If an abstract class has n- abstract methods, of the child of this abstract class doesnot define all of the n-methods of superclass, then it should be declared as 'abstract'. In such case there should be a grand child class which should define the remaining abstract methods.

Ex:

```
abstract class A
{
   abstract void fun1();
   abstract void fun2();
     void fun3()
     {
        S.O.P("Hi");
     }
}
abstract class B extends A
{
     void fun1()
     {
        S.O.P("CSE");
     }
}
class C extends B
{
     void fun2()
     {
        S.O.P("MCAIT");
     }
}
class partialAbstractDemo
{
     P.S.V.main(String args[])
     {
        C ob = new C();
        ob.fun1();
        ob.fun2();
        ob.fun3();
     }
}
```

Java StringBuffer class :

A string that can be modified (or) changed is known as mutable string. StringBuffer (Synchronized) and StringBuilder (Non-Synchronized) classes are used for creating mutable strings.

Constructors:

StringBuffer() :→ creates an empty stringbuffer with the initial capacity of 16.

StringBuffer(String str) :- creates a stringbuffer with the specified string

StringBuffer(int capacity) : creates an empty string buffer with the specified capacity as length.

methods:

① append(String s) : It is used to append the specified string with this string.

② insert(int index, String s) : It is used to insert the specified string with this string at the specified position (index)

③ replace(int startIndex, endIndex, String str) : It is used to replace the string from specified startIndex and endIndex

④ delete(int startIndex, int endIndex) : It is used to delete the string from specified startIndex and endIndex.

⑤ reverse() : to reverse the string.

⑥ capacity() : It is used to return the current capacity.

Ex: Initial capacity of empty SB is 16.

If 17th character is inserted, it is (16*2)+2 = 34

(Current Capacity * 2) + 2

7. charAt(int index) : to return character at the specified position.

8. length() : to return the length of string. [no of characters]

9. substring(int startIndex) : to return substring from the startIndex.

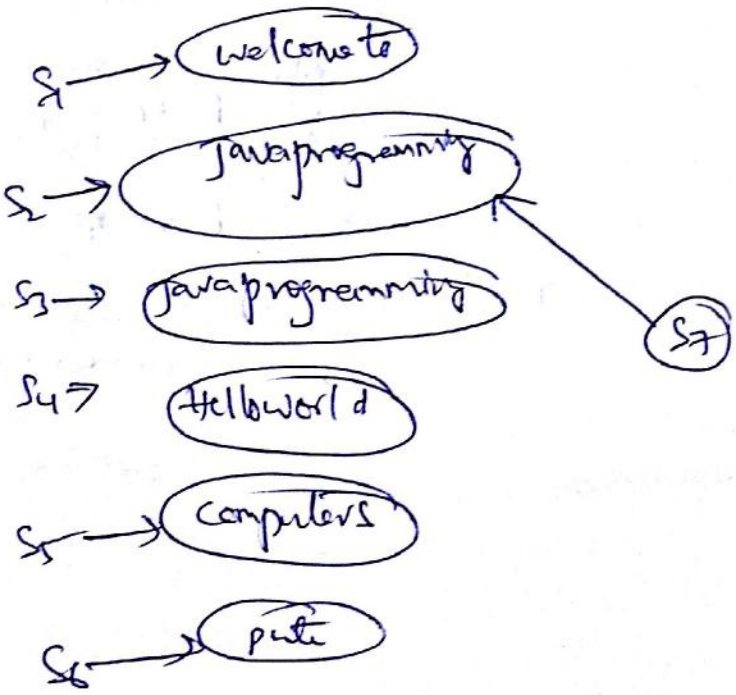
10. substring(int startIndex, int endIndex) :
It is used to return the substring from the specified startIndex and endIndex.

```
// to demonstrate methods
class StringBuffDemo
{
    p.s.v.m(String args[])
    {
        StringBuffer sb = new StringBuffer("Hello");
        sb.append("Java");
        s.o.p(sb); // HelloJava <- sb
        sb.insert(5,"to");
        s.o.p(sb); // HellotoJava <- sb
        sb.replace(5,10,"TOJAVA");
        s.o.p(sb); // HelloTOJAVA <- sb
        sb.delete(1,4);
        s.o.p(sb); // HTOJAVA <- sb
        sb.reverse();
        s.o.p(sb); // AVAJOTH <- sb
        StringBuffer sb1 = new StringBuffer();
        s.o.p(sb1.capacity()); // 16 (default)
        sb1.append("Hello"); s.o.p(sb1.capacity()); // 16
        sb1.append("to Java programming");
        s.o.p(sb1.capacity()); // 34 [(16)+2 = 34]
    }
}
```

// to demonstrate string literals, string constructors.

```
class String1
{
    p.s.v.m(String args[])
    {
        String s1 = "welcome to ";   // literal
        String s2 = new String("Java programming "); // constructor
        String s3 = new String(s1);
        String s4 = "Hello " + "world";
        so.p(s1); s.o.p(s2); s.o.p(s3); s.o.p(s4);
        char ch[] = {'c','o','m','p','u','t','e','r','s'};
        String s5 = new String(ch);
        string s6 = new String(ch, 3, 4);
        String s7 = s2;
        s.o.p(s5); so.p(s6); s.o.p(s7);
    }
}
```

o/p: welcome to
Javaprogramming
welcome to
Helloworld
computers
pute
Javaprogramming



// to demonstrate string handling methods.

```
class string2
{
    p.s.v.m(String args[])
    {
        String s1 = " welcome to ";
        String s2 = " Java programming ");
        String s3 = " WELCOME TO "
        int l1, l2;
        String s4 = s3;
        l1 = s1.length(); //
        l2 = s2.length(); //
        s.o.p (" l1 = " + l1 + " l2 = " + l2);
        char ch = s1.charAt(5);
        s.o.p ( " ch = " + ch);
        int start = 5, end = 9;
        char ch1[] = new char[10];
        s2.getChars ( start, end, ch1, 0);
        s.o.plu( ch1);
        s.o.p ( s1.equals (s2));
        s.o.p ( s2.equals (s3));
        s.o.p( s1.equalsIgnoreCase (s3));
        s.o.p ( s1.compareTo (s2));
        s.o.p ( s2.compareTo (s1));
        s.o.p ( s3.CompareTo (s3));
        s.o.p ( s1 == s2 );
        s.o.p ( s3 == s4);
        Int a, b;
        a = s1.IndexOf ('o');
        b = s1.lastIndexOf ('o');
    }
}
```

s.o.plu("a = " + a);
s.o.plu("b = " + b);

```
// to demonstrate string handling functions
class String3
{
    p.s.v.m(String args[])
    {
        String s1 = " welcome to ";
        String s2 = " Java programming ";
        String s3 = s2.substring(5); // programming
        String s4 = s2.substring(5,8); // pro (not including '8')
        String s5 = s1.concat("JAVA");
        String s6 = s1.trim();
        String s7 = s1.toUppercase();
        String s8 = s7.tolowerCase();
        String s9 = s1.replace('e','E');
    }
}
```

```
// sorting strings
class SortingStrings
{
    p.s.v.m(String args[])
    {
        String temp = null;
        String name[] = new String[100];
        int length=0;
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        S.o.println("Enter lines of text, "stop" to quit");
        for(i=0; i<100; i++)
        {
            name[i] = br.readLine();
            if((name[i].equals("stop"))==0)break;
            length++;
        }
        length = name.length();
        for(int i=0; i<length; i++)
        {
            for(int j=0; j<length-i-1; j++)
            {
                if((name[j].compareTo(name[j+1]))>0)
                {
                    temp = name[j];
                    name[j] = name[j+1];
                    name[j+1] = temp;
                }
            }
        }
        S.o.println("strings in Ascending order");
        for(int i=0; i<length; i++)
            S.o.println(name[i]);
    }
}
```