Constructor

*) is a block (similar to method) having same name as that of class.

*) doesnot have any Return types

*) modifiers are public, ~~Private~~ Private, Protected & Default.

*) It executes automatically when we create an object.

*) Used to initialise the object ✓

*) ~~It is used to~~

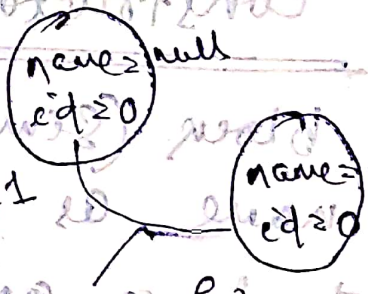
```
class test
{
   public test()
   {
      =
   }
P.S.VM ( )
{
   test t= new test();
}  t.test  X
```

object. initialization.

→ By using reference variable
→ " " method
→ " " constructor.

eg: If we not initialize object.

```
class Emp
{ String name;
  int id;
  :
P.S.V.M( )
{ Emp e1 = new emp();
  Emp e2 = new emp();
  :
  Emp en = " ";
}
```



Using Reference variable.

```
class Emp
{ String name;
  int id;
  :
PSVM()
{ Emp e1 = new emp();
  e1.name = "abc";
  e1.id = 100;
  Emp e2 = new emp();
  e2.name = "def";
  e2.id = 101;
  :
}
```

If we create 1000 object then 2000 line will be added (not a suitable approach).

Using constructor

```
class Emp
{ String name;
  int id;
  :
  Emp (String name, int id)
  { this.name = name;
    this.id = id;
  }
PSVM ( )
{ Emp e1 = new Emp ("abc", 101);
  Emp e2 = new Emp ("def", 102);
}
  :
```

Scanned with OKEN Scanner

types

Default Constructor (No-argument)	No-argument	Parameterized
→ Compiler will create.	→ User will create	

e.g.:- class test

```
{
    Test()
    {
        super();
    }
    psvm()
    {
        test t = new test();
    }
}
```

→ Compiler will create

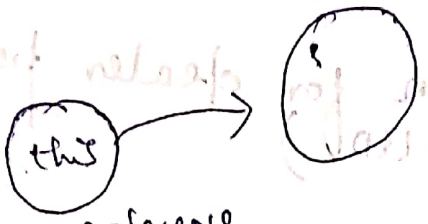
→ User will create

*) compiler will create a default constructor other user will not create any constructor.

Inner class (nested class)

This keyword.

*) this keyword is the Reference variable that refers to the current object.



*) class test
{ int i; → instance variable
void input (int x) { → local variable
i = x; i;
this.i = i;
Refer to current Reference variable or instance variable
*) without this it will print 0.
}
void show()
{ sop(i);
}
}
class xyz
{ psvm()
{ test t = new test();
t.input(10);
t.show();
}}

*) This keyword can be used to refer current class instance variable.

*) This keyword can be used to invoke current class method (implicitly)

*) This keyword can be used to invoke current class constructor.

*) It can be used to pass as an argument in the method call.

*) used to pass as an argument in the constructor or call current class method

```
class test
{
  int i;
  void display()
  {
    sop("Hello");
  }
  void show()
  {
    display();
  }
psvm()
{
  test ob = new test();
  ob.show();
}
}
```

⇒ implicitly compiler will add this keyword to the call the instance method. i.e. this.display();

current class constructor:

```
class test
{
  test()
  {
    sop("no arguments");
    this(10);
  }
  test(int a)
  {
    sop("parameterized");
    this();
  }
psvm()
{
  test ob = new test();
}
}
```

call the default constructor.

Argument in the method call

```
class test
{
  void m1(test ob)
  {
    S.o.p("I am in m1");
  }
  void m2()
  {
    m1(this);
  }
  p s v m ( )
  {
    test t = new test();
    t.m2();
  }
}
```

As an Argument.

Used to pass as an argument in the constructor call.

```
class test
{
  test(test ob)
  {
  }
  :
}
p s v m()
{
  test t = new test(this);
}
```

Command line

*) Command line arguments are arguments which are supplied to the main class at the time of execution.

*) We should provide the input at command prompt.

e.g. compile → javac filename.java
                    → Command line
run → java filename
                    → Command line

*) Read the data from command line.

```
class cmdline
{
    p.s.v.m.(String args[]).   → Used to store values passed from cmd line.
    {
        for( i=0; i<args.length; i++)
        {
            s.o.p (args[i]);
    } }
```

javac cmdline.java ↵
java cmdline x y z ↵

Find the sum of two nos.

```
class Abc
{
    p s v m (String args[])
    {
        int a, b, c;
        a=Integer.parseInt(args[0]);
        b=      "       "  (args[1]);
        c= a+b;
    } } s.o.p("Sum= "+c)
```

~~Super keyword~~

~~1.) Supper Key~~

Polymorphysim in java.

~~(Method over~~

Polymorphysim

- Compile time (static)
  - → Method overloading
  - → Handle by compiler
  - → Same name
  - → Same class
  - → Different Arguments
    - → No. of Arguments
    - → Seq. of Arguments
    - → Types of Arguments
- Runtime (Dynamic)
  - → Method overriding
  - → Handle by JVM
  - → Same name
  - → Diff. class
  - → Same Arguments
    - → No. of Arguments
    - → Seq. of "
    - → Types of "
  - → Inheritance (IS-A) Relationship

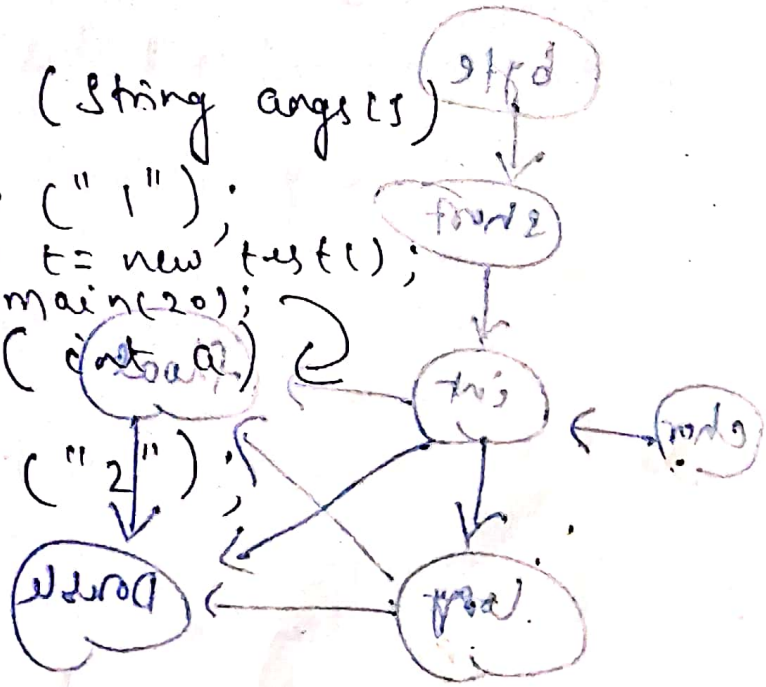
Method overloading.

```
class test
{
    void show()
    {
        s.o.p("1");
    }
    void show(int a)
    {
        s.o.p("2");
    }
    p s v m(String args[])
    {
        test t = new test();
        t.show();
        t.show(3);
    }
}
```

Q. Can we overload java main() method

```
class test
{   void show(int a)
    { s.o.p("1");
    }
    void show(String a)
    { s.o.p("2");
    }
p.s.v.m(String args[])
    { test t= new test();
      t.show(10);
    }
p.s.v.m(int a)
    { s.o.p("3");
    }
}
```

```
class test
{
    p.s.v.m(String args[])
    {
        s.o.p("1");
    }   test t= new test();
        t.main(20);
    p.s.v.m(int a)
    {
        s.o.p("2");
    }
}
```



9) Automatic Promotion

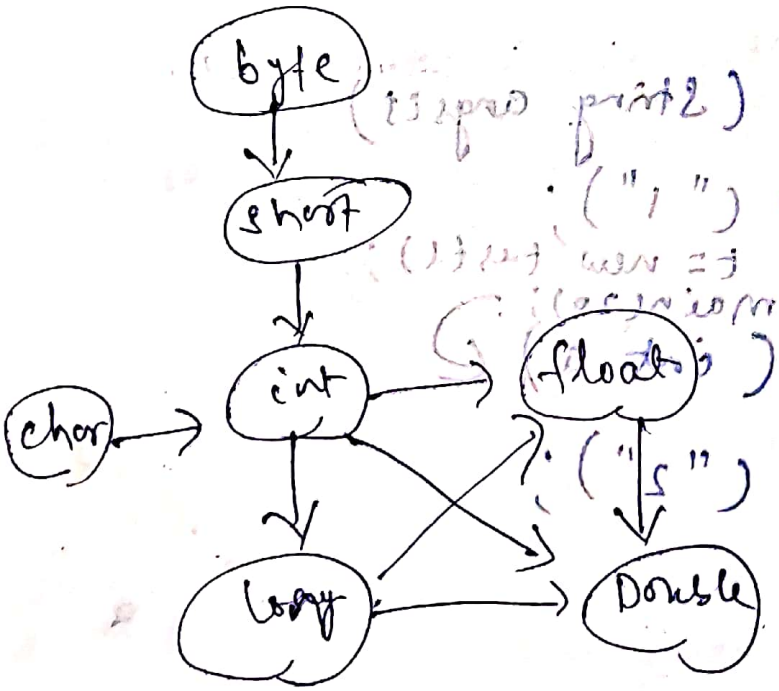
- One type is promoted to another type implicitly if no matching data type is found.

e.g.

```
class test
{
void show(int a)
 { S.o.p ("int method");
 }
void show (String a)
 { S.o.p("String method");
 }
P.S.v.m (String args[])
{ test t= new test();
  t.show (20);
  t.show ('a');
}
```

→ character.

no match found.
Automatically converted to integer.



Method overriding.

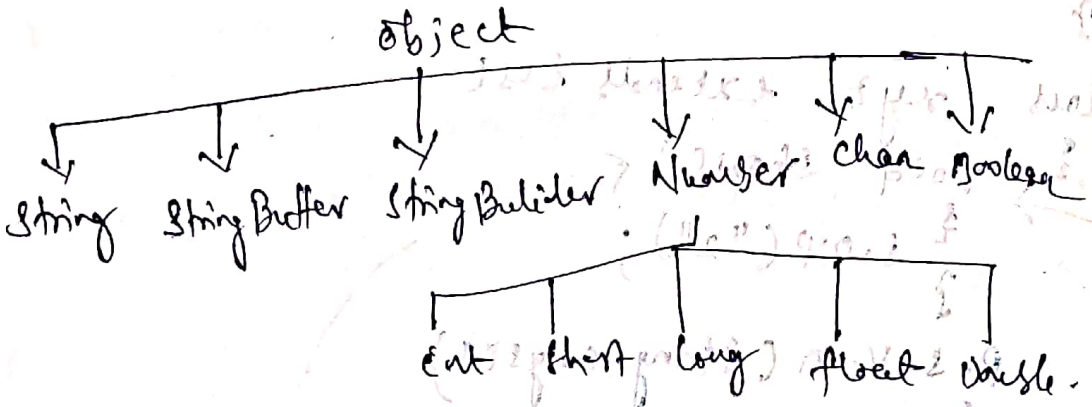
```
class test
{
    void show()
    {
        s.o.p("1");
    }
}
class xyz extends test
{
    void show()
    {
        s.o.p("2");
    }
    p.s.v.m(String args[])
    {
        test t = new test();
        t.show();   ---> 1.
        xyz x = new xyz();
        x.show();   ---> 2
    }
}
```

Q) Do overriding method must have same Return type? (Co-variant Return type)

```
class test
{
    (void) show()
    {
        s.o.p("1");
    }
}
```
object

```
class xyz extends test
{
    (void) show()
    {
        s.o.p("2");
    }
}
```
String

*) If Parent class Return type is parent datatype then we can override it in child class with child data type.

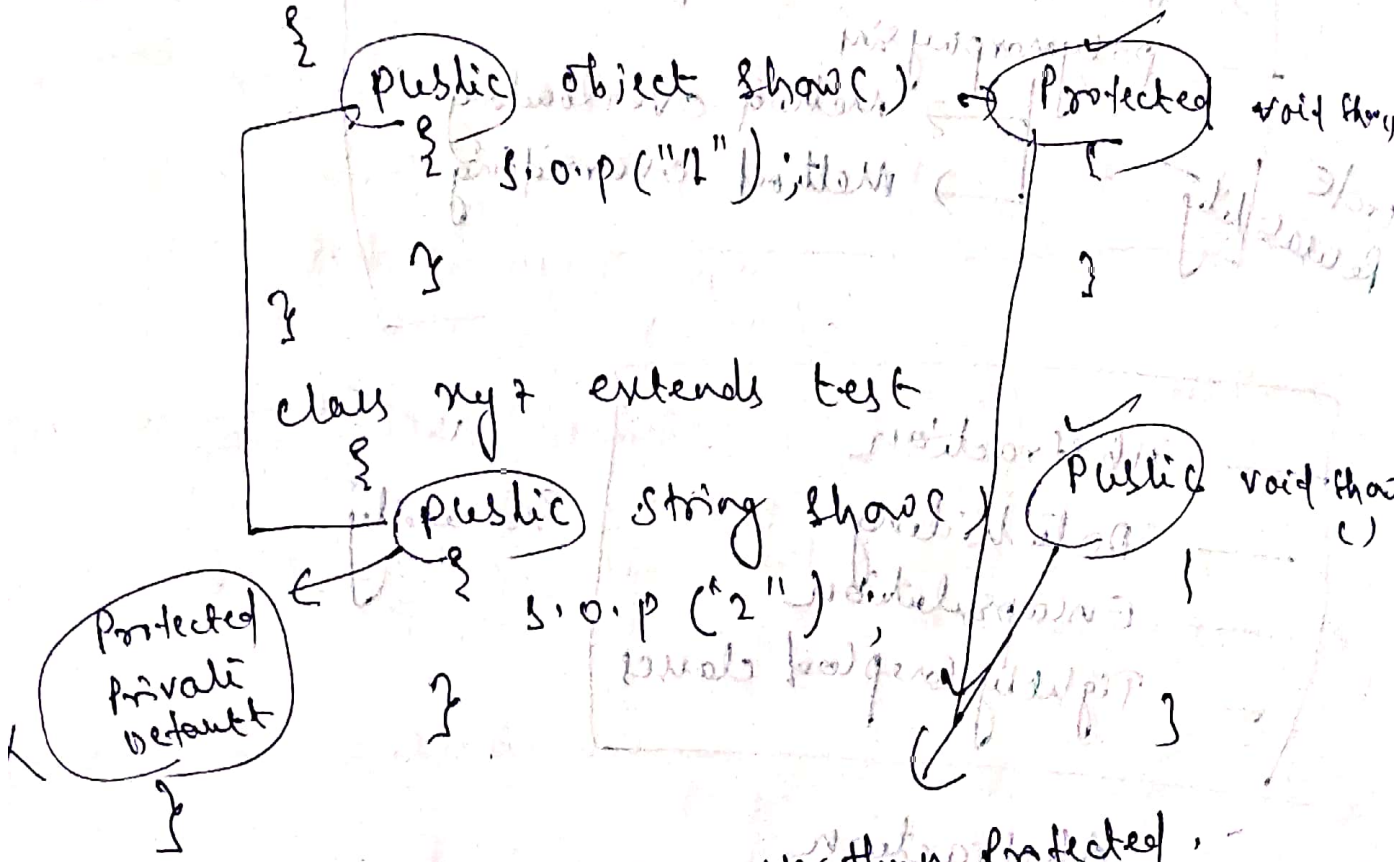



```
class test
{
    Object show()
    {
        S.O.P("1");
        return null;
    }
}
class xyz extends test
{
    String show()
    {
        S.O.P("2");
        return null;
    }
    P.S.V.M()
    {
        test t= new test();
        t.show();
        xyz obj= new xyz();
        obj.show();
    }
}
```

Q2) Accesses Modifiere in overriding

*) parent class Access modifier should be same or less than child class accesses modifier.

e.g.



Q3)