IS-A relationship

" ⓐ type object has properties of 'a' type object (or)

" ⓑ object is-a type of 'a' object

Eg:- 1) marker pen is a type of pen (or) can is a type of tata.
2) car is a type of vehicle
3) son is-a type of ster (or) Father is a type of son.
4) Lion is a type of Animal

Two ways

① extends - inheritance
② implements -> implementation (or) realization

Encapsulation

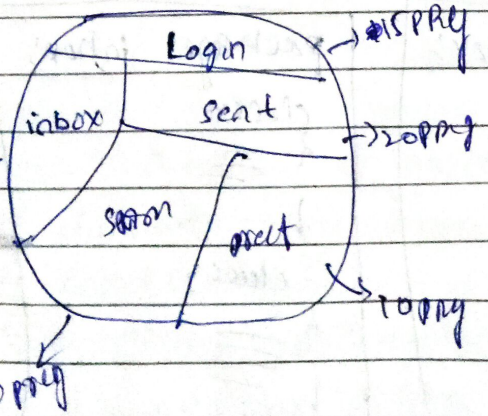
-> binding & protecting members

Eg :- class Demo 'java package
{
int x=10;
void test()
{
}
}

Java package

* collection of java programs (or) group of java program
* organizing & modularity.
e.g. Gmail App

* login package contains 15 prg
* inbox ——— 30 prg



22/03/19

Declaring package

Package packagename

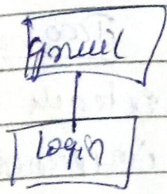
Rules:

1. package declaration stmt must be first stmt of source file.
2. one src file can have only one declaration

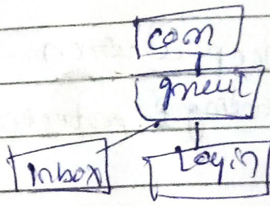
Syntax e.g:

1. package login; → package name
   (package → keyword)

2. package gmail.login; → child/parent
   (gmail → parent package)



3. package com.gmail.login;
   package com.gmail.inbox



eg 1)

```
package login;
class P
{
}
class Q
{
}
```

P.java

class P & Q belong to package login
(or)
P & Q are members of package login

class X & Y are member of package inbox

```
package inbox;
class X
{
}
class Y
{
}
```

X.java

```
package login;
class P
{
}
class Q
{
}
```

P.Java

```
package inbox;
class T
{
}
class U
{
}
```

T.Java

```
package inbox;
class X
{
}
class Y
{
}
```

X.Java

22/03/19

Class X, Y, T, U members of package inbox.

No. of package is - 2 — no. of inbox class - 4

total no. of class - 6 — no of login class - 2

```
package login;
class P
{
}
class Q
{
}
```

login.P p1 = new login.P();

in general syntax

package name.classname

fully qualified classname

differentiate the two different package in the same class name

↑ if import is done after this no this qualification needs.

```
package inbox;
class X
{
}
class Y
{
}
class P
{
}
```

Import statement

⇒ used to import class & it's members to current package

Syntax

import package.classname;

Rules:

1) import stmt must be after package declaration stmt

2) multiple import stmt

```
package login;
class P
{
}
class Q
{
}
```

P.java

```
package inbox;
import login.P
class X
{
}
class U
{
}
```

X.java

2 types of import

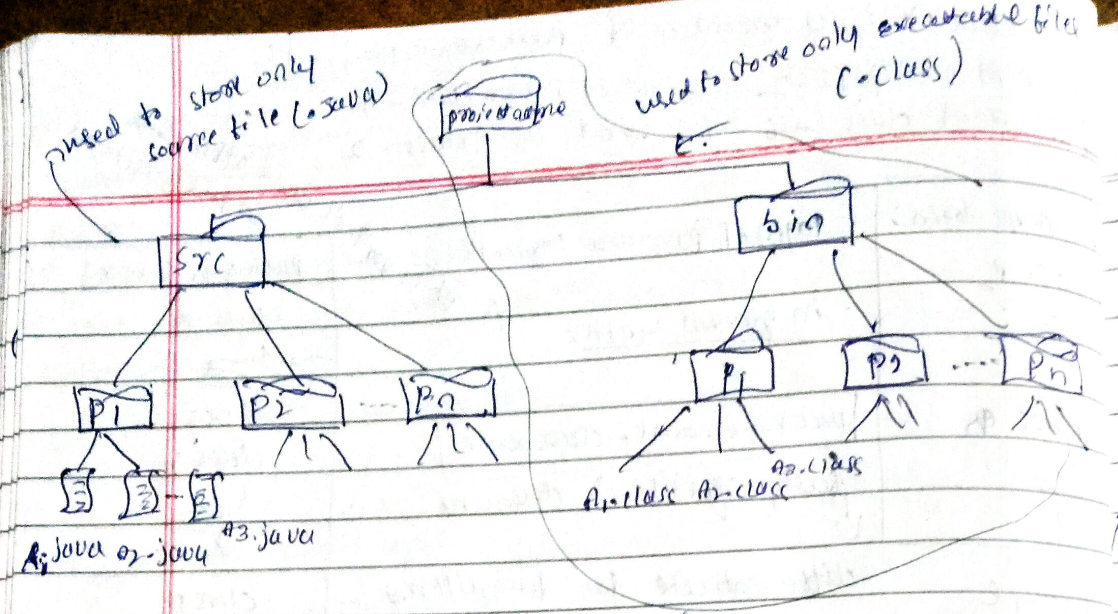
1) import statement → import static & non static

2) static import statement

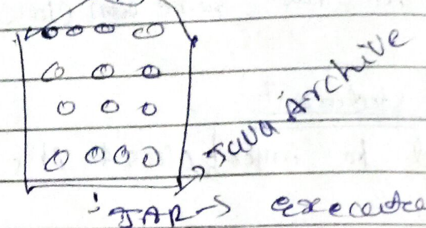
Syntax :- import static packagename.classname.*;

↳ import all static members of class

supported from jdk 1.5v



Create folder
↓
window
↓
perspective
↓
open perspective
↓
Java
file
↓
new
↓
Java
↓
project name
src → right click
↓ new
→ create project
name - com.jspiders.p2
↓
click right
↓
new
↓
class



28/03/19

Prg1:-

```
package com.jspiders.p1;
import java.util.Scanner; → to user input
public class main2
{
  PSVM (String [] args)
  {
    S.O.P. ("main program started");
    String name;
    int age;
    Scanner scn = new Scanner (System.in);
    System.out.println ("enter your name");
    name = scn.next();
    S.O.P ("enter your age");
    age = scn.nextInt(); → upper case
    S.O.P ("values entered are");
    S.O.P (name);
    S.O.P (age);
}}
```

Prg1:- package com
public class Dem
{
private int
int a2 =
protected int
public int
this class can only
private → void f1()
{
S.O.P (
S.O.P (
S.O.P (
S.O.P (
}
}

private value is within this class only

Prg2:- package
De
{
void
}
Dem
S.
S.
S.
}
}

Prg3:- package
import
public
{
void
{
D
}

prg1:- package com.jspiders.p2;
public class Demo1
{
private int a1 = 12; → private

int a2 = 34;
protected int a3 = 56;
public int a4 = 65;

this class can only access if prg2 can't access
void f1()
{
s.o.p("a1 value: "+a1);
s.o.p("a2 value: "+a2);
s.o.p("a3 value: "+a3);
s.o.p("a4 value: "+a4);
}
}

(private value is accessible inside class only)

→ default or protected.

prg2:- package com.jspiders.p2;
public class Demo2
{
void f2()
{
}
Demo1 d1 = new demo1();
s.o.p("a2 value: "+d1.a2);
s.o.p("a3 value: "+d1.a3);
s.o.p("a4 value: "+d1.a4);
}
}

prg3:- package com.jspider.p3;
import com.jspider.p2.demo1;
public class sample1
{
void f3()
{
~~Demo1 d1 = new Demo1();~~
s.o.p("a4 value: "+d1.a4);
}

Java package access specifier
1) private → only inside class
2) default → inside package
3) protector → only inside package
4) public → access anywhere

Both are same package p2
- com.jspiders.p2
  - Demo1
  - Demo2
- com.jspiders.p3;
  - sample

Default & protected values will be access package level if means prg1 & prg2 can access these values.

Here public only can access it

protected is also similar to default, but in inheritance both differs

access specifiers)

java provides 4 access specifiers to define access ability of the members. they are

1) private 2) default 3) protected 4) public

1) any member declared with a ~~to the~~ private keyword has a restriction upto class level

* private members can't be access outside the class
* private access specifier can be use in variable declaration, constructor declaration, function declaration.
* ~~any member without~~

2) default

* any member without access specifier will have default access level restricted upto package level
* any class belonging to the same package can access the default access members
* default access can be used in variable or method or constructor declaration

3) * any member declared with a protected keyword has a restriction upto the package level, it can be access from outside package by inheriting the protected members

4) * member declared with public access specifier doesn't have restriction it can be accessed from any package.

Bafna Gold
Date: 20/03/19

Encapsulation :

Encapsulation is a process of binding ~~to the~~ the members to the class; it also defines protecting members of the class.

↑ class is an encapsulation of member variables and member function.

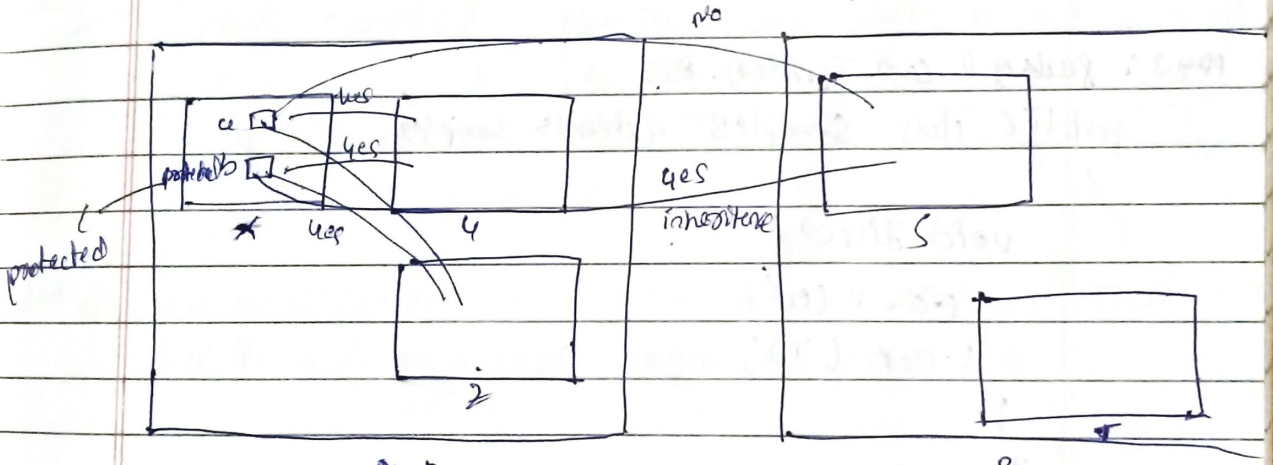
& we can't define any variable (or) function outside the class (BCZ of Encapsulation will not occurs so)

∨ the members of the class protected by using access specifiers.

* the package is a collection of java definition type which encapsulation the no. of classes.

* the packages are encapsulated in jars files

* By default java supports encapsulation. without Encapsulation we can't define any program.



input p1.x;

class S extends X

class T extend X

{ }

in class T extends S

input p1.x;

class S extend X

{ }

Prg1:- package com.jspiders.p1;
public class Ran1
{
protected int x=12;
public int y=34;
}

Note:- import com.jspiders.p1.Ran1
we use import for importing or inheriting the members of other package to sample class of P3 package

Prg2: package com.jspiders.p3;
import com.jspiders.p1.Ran1;
public class sample2 extends Ran1
{
void test()
{
s.o.p(y);
s.o.p(x);
}
}

Prg3: package com.jspiders.p3;
public class sample3 extends sample2
{
void disp(){
s.o.p(a);
s.o.p(x);
}
}

Rules.

1) * defining the class inside another class is known as nested class or inner class

* the inner class can have any of the 4 access specifier but outer class can either public or default access specifier

2) * in a source file multiple class definition are allowed and, any one outer class can be public.

* the source file name should be class name having public access.

3) * constructor ... to the ... restricte... con't b...
Example.
Prg:- packa... class ... {

constructor of sample4 class construct object is only created that class only but we can't create object in other class main sample5 here

rule 4) ...

rule 5) ...

3) If constructor one members of class we can define any 4 access to the constructor if constructor is defined as private restricts the objects created in other class. itself, object can't be created in other classes.

Example:

PP1:- package com.spiders.s.p3;
class sample4
{
sample4 s1 = new sample4();
private sample4() → constructor ⓐ then object can be created within sample4 class
ⓑ other class can't be access or object can't no created for private keyword
}
void test()
{
sample4 s1 = new sample4();
}
}

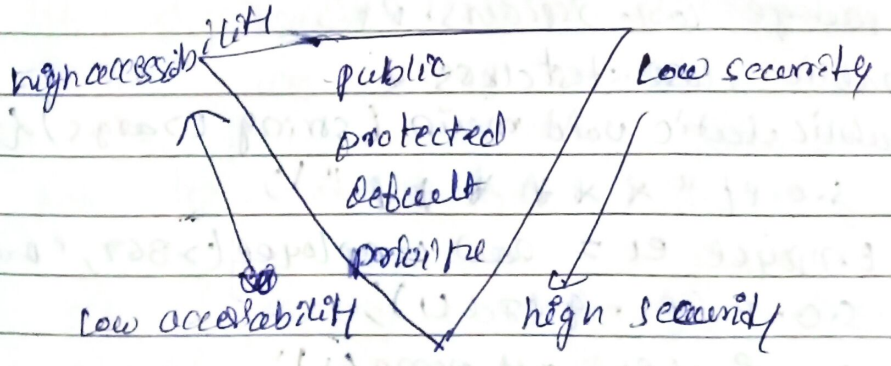
Constructor of sample4 class construct object it only executed that class only but we can't create object in other class mean sample5 here

class sample5
{
}

class sample6
{
}

it restrict object creation to the class itself, object outside can can't be created in other class

rule ❹ 4) the access specifier ——— has a level of access ability and represented below according security



rule 5) the default constructor provided by the compiler will have same access of its the class

private access specifiers can be access by other class by using getters setters.

Prg 1 → package com.jspiders.p1;

```
public class Employee
{
    private int Id;
    private string name;
public Employee (int id, string name)
{
    this.id=Id;
    this.name = name;
}
```

→ private Id and name accessing by using getters and setters.

```
// getter and setters
public int getId(){
    return id;
}
public string get name(){
    return name;
}
public void setname (string name){
    this.name = name;
}
}
```

prg 2 →

```
package com.jspiders.p1;
public class testclass {
public static void main (string []args){
    s.o.p(" * * * * * * ");
    Empoyee e1 = new Employee(2367, "Dorach");
    s.o.p (e1.getId());
    s.o.p (e1.get name());
    e1.setname("Ramesh Kumar");
    s.o.p(e1.getname());
    s.o.p(" * * * * * * * ");
}
}
```

method overloading

methods → specif

(1) method declara

2) method defini

same operation / fu

① climb floo

(a) stair ca

(b) lift

(c) escalato

(d) later

② J spiders

① walk

② bike

③ bus

④ metro

Prg :- package

public

{

//methu

void te

{

s.o.p

s.o.p

}

void

{

s.

s.

}

Do

{

}

by using getters

method overloading → method name one same differ in argument list

methods → operation / task

27/03/19

1) method declaration
2) method definition

Real operation / functionality having multiple options

① climb floor	③ payment → PTE real time example
a) stair case	a) COD
b) lift	b) debit card
c) escalator	c) credit card
d) ladder	d) paytm
② Transportation	
① walk	
② bike	
③ bus	
④ metro	

method overloading

constructor overloading in different class same name

diff class & same method.

```
Pgm:- package com.jspiders.pi;
      public class Demo1
      {
        //method overloading
        void test (int arg1)
        {
          S.O.P ("running test(int) method...");
          S.O.P ("arg1 value: " +arg1);
        }
        void test (double arg1)
        {
          S.O.P ("running test (double) method...");
          S.O.P ("arg1 value: " +arg1);
        }
        void test (int arg1, double arg2)
        {
          S.O.P ("running test (int, double) method...");
          S.O.P ("arg1 value: " +arg1);
          S.O.P ("arg2 value: " +arg2);
        }
```

Prog2:- package com.spiders.p1;
Public class mainclass1
{
public static void main (String[] args){
S.O.P ("main method started");
Demo1 d1= new demo1();
d1. test (12);
d1. test (4.5);
d1. test (12, 4.5);
}
}

Prog1:- package com. spiders.p1;
public class Demo2 {
void test ()
{ S.O.P("running test() method ...");
}

class Demo3 extend Demo2
{ void test (int arg)
{ S.O.P("running test (int)method");
}
}

prog2:- package com.spiders.p1;
public class main class1
{
PSVM (String[] args)
{
S.O.P("main method started");
Demo3 d1= new demo3();
d1. test()
d1. test (32);
S.O.P ("main method ended");
}

in a class defining more than ~~defining~~ (2) multiple method with the same name ~~different~~ argument list it known as method over loading.

* the argum
argument
Heade
* the over lo
type
* the meth
parti
* while
function ac
go for
me
* in her
in
* while
return real
should
* only
* ~~is~~ -A
per
* met
par
* while
where
met

prog:-

* the argument list should be differ. either in argument type (or) argument length we can overload both static method or non static method.
* the over loaded method are involved based on the argument type
* the method over loading is used to achive compile time porlimorfism.
* while developing application if we come across functionality with multiple option then we should go for method overloading

*

method over riding

* inheriting a method from superclass changing implementation in a subclass according to subclass specification. is known as method over riding.
* while over riding the method in sub class should be retain the same method declaration and it should provide different implementation.
* only non static method can be overridded.
* IS-A relationship (or) inheritence is compulsory to perform method overriding.
* method over riding is used to achive run time porlimorfism
* while implementing a project if we come across a method where the implementation as to be check then we go for method over riding.

Eg:-

```
package com.ispiders.pl;
public class demo1
{
    void test()
{   s.o.p("running test() method....");
}
}
```

array (or) argument

```
class Demo2 extends Demo1
{void test (){
  S.o.p ("test() overridden in Demo2 class");
  }
}
```

pgm:- package com.jspiders.p1;

```
public class mainclass
{ p s v m (String[] args) {
  S.o.p ("********");
  Demo2 d1 = new Demo2();
  d1.test(); ---> in this o/p implemented of it will come not demo1 test method
  S.op ("*****");
  }
}
```

* While overriding the sub class as a permission to increase access level But doesn't have a permission to decrease the access level.
* static methods of the super class can't be overriding reason-> static members are not inherited to sub class
* if super class is having ~~fixed~~ final non static method it will be inherited sub class, but it can't be overridden.
* the ~~private~~ private methods of the super class can't be overridden since it doesn't have access in the sub class.

29/05/19

```
package com.jspiders.p1;
public class Demo1
{
    int x = 12;
}
class Demo2 extends Demo1
{
    int x = 34;
    void display()
```

{int x=5;
S.o.p (
S.o.p (
S.o.p (
}
}

this ->
super ->

this()
* call another class
* should be
* must be construct
* only one
* can call construct
eg:- this()
* explicit

pgm:- pack
publi
{
Dr
{
S.o.p(
}
}
class
{
{
}
}

```
{
int x=56;          // default when local variable are inherited
S.O.P("local x value: "+x);      // default when local variables are not inherited
S.O.P("current class x value: "+this.x);     // it accesses the members
S.O.P("super class x value: "+super.x);      // variables of super class
}
}
```

→ store as one more program → go to back page

this → access current class non-static member

super → access super class non-static member

this()	super()
* call another statement of current class.	* call constructor of super class from sub class
* should be must in constructor body	* should be used in constructor body
* must be first statement of constructor	* must be first statement of constructor
* only one this() is allowed	* only one super() is allowed
* can call no arg (or) parameterized constructor	* can call no arg (or) parameterized constructor eg: super(), super(10) ↳ constructor
eg: this(), this(10).	
* explicit call	* implicit (or) explicit call

implicit call

We can call the constructor by another two calling statement (i) this() (ii) super()

prg:-

```
package com.jspiders.p1;
public class Demo3
{
Demo3()
{ [        ] → default super() written by compiler
S.O.P("running Demo3() constructor");
}
}
class Demo4 extends Demo3
{
Demo4()
{ [        ] → default super() written by compiler.
S.O.P("running Demo4() constructor");
}
}
```

class Demo5 extends Demo4
{
Demo5() —) Defaultly written by compiler
{ super();
S.O.P ("Running Demo5() constructor");
}
}

Prg2:- package com.jspiders.P1;
public class mainclass1
{
P S V m (String [] args)
{
S.O.P ("main method started");
Demo5 d1 = new Demo5();
S.O.P ("main method ended");
}
}

explicit call

Prg1:- package com.jspiders.P1;
public class Demo3
{
Demo3 (int arg)
{
S.O.P ("Running Demo3() constructor");
S.O.P ("arg value: " + arg);
}
}

Class Demo4 extends Demo3
{
Demo4()
{
super (25); —> explicit calling
S.O.P ("Running Demo4() Constructor");
}
}

prg :- package
public m
{
ps vm (
{
S.O.P
Demo4
S.O.P
}
}

* when sub class
class constructor
—> * the sub
super class ei
* it super
then sub class
* it super cl
sub class s
* A phenomenon
to initalize
* the constructor
constructor happed in
class c

compiler

prg :-

```
package com.sridevi.p1;
public main class1
{
ps vm ( String[] args )
{s.o.p ("main method started");

Demo4 d1 = new demo4();
s.o.p ("main method ended");
}
}
```

* when sub class inherits properties from super class ~~the sub class constructor should make a call to the sup~~
  * the sub class constructor should make a call to the super class either implicitly or explicitly.
  * if super class contains no arguments constructor then sub class constructor will make implicit calls.
  * if super class contains parameterized constructor then sub class should make explicit call
* A phenomenon of running more than one constructor to initialize an object is known as constructor chaining
* the constructor chaining happen within current class
  constructors happened (ii) it can happen how sub class and super class constructor

Abstract method

02/04/19

concrete method (or) complete method	Abstract method (or) incomplete method
1) method declaration	* has only method declaration, no defn / body
2) method defn / body	e.g. abstract void test();
eg:: void test() { ≡ }	* Abstract method can be defined in a) Abstract class b) ~~interfaces~~ interface
3) concrete methods can defined in abstract class or concrete class	class name must be abstract keyword if we want to use abstract

Note:- new operator can't ~~create~~ create object of abstract class

Prg 1:- package com.jspiders.p1;

```
(abstract) public class Demo1      <- in abstract class
{                                   (keyword)
    int x = 12;
    static int y = 3;
    Demo1()
    {
        S.O.P ("running constructor");
    }
    void f1()
    {
        S.O.P ("running f1() method");
    }
    static void f2()
    {
        S.O.P ("running f2() method");
    }
    abstract void f3(); -> (abstract method don't have body)
}
```

Note :- cannot be construct an object in abstract class, can't create object in main class & can't access nonstatic.

) 02/04/19

declaration,

l test ();

) be defined

s

interface

abstract
want to

ct class

d don't have

an't create

prg2 :- package com.isiders.p1;

~~main~~ public class main class {

{

psvm (string [] args)

{

~~Demo1.f2 ()~~

s.o.p ("y value :" demo1.y);

Demo1.f2 ()

}

}

we can't create object of nonstatic member

Bafna Gold
Date: Page:

* here concrete class contain abstract but function should be also contain abstract.

* abstract can have constructor, static, nonstatic

To access non-static members in abstract class we use inheritance

prg1 :- package com.isiders.p1;

abstract public class Demo2 — this is abstract class not a interface class

{

int x;

Demo2 (int arg1) { ← constructor

x = arg1;

}

void test ()

{

s.o.p ("running test () method");

} in here Demo2 we can't inherited it you want inherited all should abstract method declaration in subclass

Because pure c → all method is concrete so that is to simply inherited to Demo3 class if any abstract method declared

class Demo3 extends Demo2 → inheriting class Demo2

{

Demo3 (int arg1) {

super (arg1);

}

}

here no abstract keyword it's normal class.

Prog2: package com.spiders.p1;
Public class main class 2
{
P S V m (String [] args)
{
S.O.P ("** * ** **");
Demo3 d1 = new demo3 (25);
S.O.P (d1.x);
d1.test()
S.O.P ("* * * * *");
}
}

Note to achive non static members in abstract class first abstract class inherited to normal class (sub class) then we created object in main class to achive non static member

method1: inherited abstract method from by creating abstract sub class IMP

include abstract key can't be used in static class it's used only non static member

Prg1:- package com.spiders.p1;
abstract public class demo4
{
void m1()
{
S.O.P ("running m1() method");
}
abstract void m2();
}

this program we can't access the properties

there have abstract method so the class should be contain abstract otherwise we could not inherite

abstract class demo5 extends demo4
{
}

note :- To access the properties we use inheritance, To access abstract method using inheritance, class should be abstract.

if you want to declare use in normal class you should declare abstract method of class in normal sub class

Prg2:- package com.spiders.p2
public class main class 3

prg1: package
abstract
void
S.O.
}
class
{
note
1
Prg2
abs
clas

pkg1: package com.jspiders.p1
abstract public class Demo4
{
void m1()
{
s.o.p("running m1() method");
}
abstract void m2();
}

Here we use normal class so the abstract method void m2() declared in a normal subclass

class Demo5 extends Demo4
{
void m2()
{
s.o.p("m2() method defined in Demo5 class");
}
}

note: ① declare abstract key word otherwise declare the abstract function body in class which you extract the class
Eg:- in Demo5 we declare the body of abstract method of void m2() function in Demo5

Pkg2

package com.jspiders.p1;
public class main class
p s v m (String[] args)
{
s.o.p("*** ***");
Demo5 d1 = new Demo5();
d1.m1();
d1.m2();
s.o.p("***");
}
}

abstract method function not contain body But abstract class contain body and function.

Interface

03/04/19

* interface is a java type definition which need to be declarred using "interface" key word.
* interface can store only abstract method. whereas concrete method are not allowed within an interface

(concrete → normal class methods)

Syntax :- interface InterfaceName

```
{
scope
}
```

* All the methods with in a interface are auto matically public and abstract.
* all the variables with in a interface are automatically public, static and final

Ex:-

```
interface Demo
{
    void test();   — public abstract
    int x = 50;    — public static final
}
```

NOTE :- A class can have IS-A relationship with an Interface with the help of implements key word.

	class	interface
class	extends	implements
interface	X	extends

Note :- we can not create an object of Interface

```
interface Demo
{
    void disp();
    int x=10;
}
```

```
class Runner implements Demo
{
    public void disp()
    {
        S.O.P("Hello");
    }
    PSVM...
    {
        Runner r = new Runner();
        r.disp(); // Hello
        S.O.P(Demo.x); // 10
    }
```

04/19

prg1: package com;
public interface A
{
void m1(); // automatically public & abstract
int x=90; // public, static and final
}

prg2: package com;
public interface B extends A
{
void m2();
}

in normal class, default method is default
In interface, public is default. when you implements the class to interface we public key word

prg3: package com;
public class mainclass implement B
{
public void m1()
{
S.O.P("in m1");
}
public void m2()
{
S.O.P("in m2");
}
P S V M (String[] args)
{
mainclass c = new mainclass();
c.m1();
c.m2();
S.O.P(A.x); (or) S.O.P(m1.x);
}
}

you should declare before abstract method in class

O/P
in m1
in m2
90

multiple inheritance is possible in terms of interface the reason is because interface doesn't contain constructors.

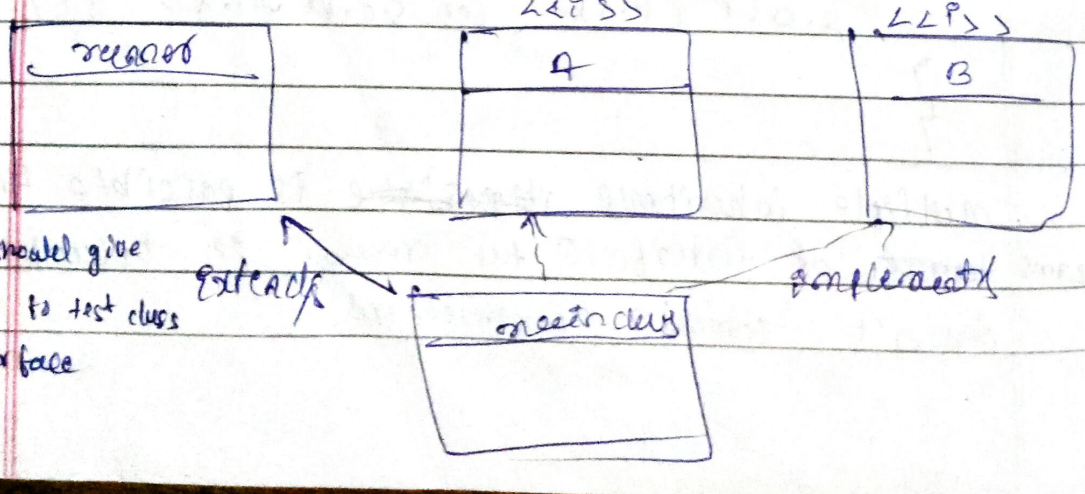
NOTE:- one class can implements can any no of classes

prg1 → package com;
public interface A
{
void m1();
}

prg2 → package com;
public interface B
{
void m2();
}

prg3 → package com;
public class runner implements A, B
{
// one class can ~~extends~~ implements any no of interface
public void m1()
{
S.O.P("method overridden from interface A");
}
public void m2()
{
S.O.P("method overridden from interface B");
}
public s v m (String[] args)
{
runner r = new runner();
r.m1();
r.m2();
}
}

note:- A class can extend one class and implements any no of interfaces at a time.



main class should give first priority to test class and to interface

prg1:- package com;
public interface A
{
}

prg2:- package com;
public interface B
{
}

prg3: package com;
public class runner implements A, B
{

}

prg4:- package com;
public class mainclass extends runner implements A, B
{

04/04/2019

Typecasting (or) type conversion

-> one type is converted to another type

Two type

1) primitive type casting (or) Data type casting
* a datatype is casting to another data type
2 type
i) widening (ii) Narrowing

2) non primitive type casting (or) class type casting
* a classtype is casted to another classtype
(SuperType) (JavaType)

2 types
1) upcasting 2) Down casting

Data type casting

(1) int x = 95;
x is a variable of int type
value of int type

(2) Double y = 4.6;
y is a variable of double type
value of double type

typecasting

typecasting

of class

package com;
interface B

main();

interface

interface A{}

interface B{}

implements

<<P>>
B

extends

(2) int x = 4.6; → value of double type
x is a variable of int type

type mismatch

double y = 7; → value of int type
y is a variable of double type

prg1 :-
```
package datatypecasting;
public class mainclass1
{
public static void main(String[] args)
{
S.O.P("main method started");
int n = (int) 59.99; → narrowing (explicit)
// double type is casted to int type
double y = (double) 45; → widening (auto implicit
// int type is casted to double type (or explicit)
S.O.P("x value: "+x);
S.O.P("y value: "+y);
S.O.P("main method ended");
}
}
```

prg1 :-
```
package datatypecasting;
public class mainclass2
{
public static void main(String[] args)
{
S.O.P("-----");
int x1 = 123;
double y1 = 45.12;

int x2 = (int) y1; → narrowing memory before smaller
double y2 = (double) x1; → widening memory become extra.
```

prg2 :-

S.O.P
S.O.P
S.O.P
S.O.P
S.O.P
}
}

prg1 :-

Char

→ narroc
→ widen

```
S.O.P("x1 value : "+x1);
S.O.P("y1 value : "+y1);

S.O.P("x2 value : "+x2);
S.O.P("y2 value : "+y2);
S.O.P("----------");
}
}
```

prg1 :-

```
package datatype casting;
public class Demo1
{
 void square (int arg1)
 {
   S.O.P("Square of "+arg1+" is "+(arg1*arg1));
 }
}
```

prg2 :-

```
package datatype casting;
public class mainclass3
public s vm (String[] args)
 {
  S.O.P("*********");
  Demo1 d1 = new Demo1();
   d1.square(5);
   d1.square((int)2.6);
    S.O.P("*********");
 }
}
```

widening will have default take

Datatype casting

narrowing

Char byte short int long float double

widening

→ narrowing can be done only explicit
→ widening is done by both explicit & implicit

ex:-
int x = (int) 59.44; // explict narrowing.
// double type is casted to int type

double d = 45; // auto widening → default compiler will take has double

prg1:- package dedatypecasting;
public class mainclass {
}
P S V M (String [] args)
{
S.O.P(" * * * * * * ");
char c1 = 'a'; → converted to ASCII value.
char c2 = 'z';
char c3 = 'A';
char c4 = '*';

int n1 = c1; | auto widening
int n2 = c2; |
int n3 = c3; |
int n4 = c4; |

S.O.P(" ASCII value of "+c1+" is "+n1);
S.O.P(" ASCII value of "+c2+" is "+n2);
S.O.P(" ASCII value of "+c3+" is "+n3);
S.O.P(" ASCII value of "+c4+" is "+n4);
S.O.P(" * * * * * * * ");
}
}

* casting one type of information to another type is known as type casting
* there are 2 types of typecasting
1) primitive type casting 2) non primitive type casting
* converting one datatype to another data type is known as primitive type casting it is also known as datatype casting

* primitive type
1) widening
2) casting lower
"wid
* widening can
* if compiler
"explicit type
* if code
"explicit t
* casting higher
"narrowing"
* narrowing
* whenever

class typecas

Demo1 d1 = an
↓
d1 is a variable
of Demo1 type

Demo2 d2 =
↓
d2 is a variable
of Demo2 type

Demo1 d
↓
d1 is a variable
of Demo1

Demo2 d2
↓
d2 is a variable
of Demo2

* primitive type casting classified into 2 type.
widening => narrowing

* casting lower data type to higher data type it known as "widening"

* widening can be perform implicitly or explicitly.
* if compiler perform type casting then it is known as "implicit type casting"
* if code perform type casting then it is known as "explicit typecasting"
* casting higher data type to lower data type it known as "narrowing"
* narrowing should be perform explicit
* whenever we perform narrowing then will be loss of data.

class typecasting

Demo1 d1 = new Demo1();
↓ d1 is a variable of Demo1 type
↳ object of Demo1 type

Demo2 d2 = new Demo2();
↓ d2 is a variable of Demo2 type
↳ object of Demo2 type

type widening.

Demo1 d1 = (Demo1) new Demo2();
↓ d1 is a variable of Demo1 type.
↳ object of Demo2 type

Demo2 d2 = (Demo2) new Demo1();
↓ d2 is a variable of Demo2 type
↳ object of Demo1 type

type narrowing

If you want explicit class typecasting you should write like this otherwise compiler will throw error

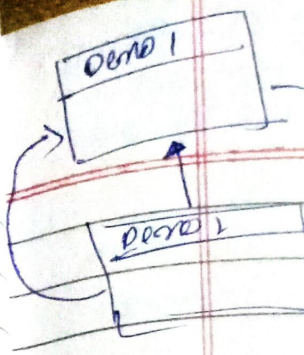
5/4/19

prog 1:- ~~Demo~~ package classtypecasting;
public class Demo1
{
int x = 12;
void m1()
{
S.O.P("running m1() method");
}
}

prog 2:- package classtypecasting;
public class Demo2 extends Demo1
{
int y = 34;
void m2()
{
S.O.P("running m2() method");
}
}

prog 3:- package classtypecasting;
public class mainclass1
{
P S V m(String [] args)
{
S.O.P("running main method");
Demo1 d1 = (Demo1) new Demo2(); // d1 is not the Demo1 type
Demo2 d2 = (Demo2) new Demo2(); // Demo2 type is casted to
d1 is a ref var of Demo2 type, object. Demo1 type Demo1. type
~~Demo1~~ U Demo1 type casted to Demo2 type

// Rule; 1. class must have is-a relationship
2. class or object must have properties of the other classes
to which it should be ~~casted~~ casted
S.O.P("*****");
}
}

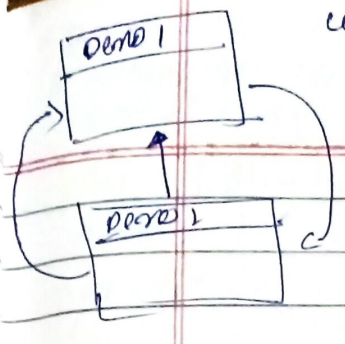


* casting a cl
* to perform
1. classes
2. class mul
there are
1) upcasting
2) downcas
* casting
up
* casting
* casting
* casting
downcasti
* only up

prog upcast

Demo1
U

5/4/19



upcasting Demo2 (cast) Demo1
Subclass type → super class type

Date: Page:

Downcasting Demo1 (cast) → Demo2
superclass type → subclass type

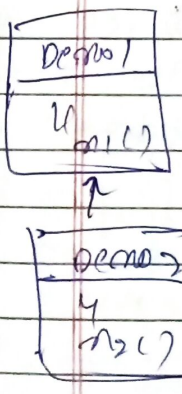
* casting a class type to another class type is known as class type casting
* To perform class type casting we have to fulfill below rules
1. Classes must have IS-A relation ship
2. Class must have properties of the class to which it's casted

there are two types of class type casting
1) upcasting
2) downcasting

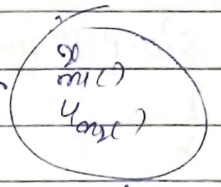
* casting sub class type to super class type is known as upcasting. when sub class type is upcasted it's behaviour like super class type.
* casting can be done either Amplicity or explicity.
* casting super class type to sub class type is known as down casting. down casting should be done explicitly
* only upcasted object can be down casted.

upcasting: * Subcasted to superclass
* Implicitly / Explicitly

Demo1 d1 = (Demo1) new Demo2();



d1 (Demo1 type reference variable) casted to Demo1 type



Demo2 type object

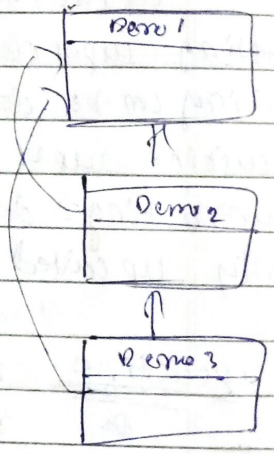
of Demo2 type can Demo1 type, type is casted to Demo1.type

other classes

prog1:- same behind prog1

prog2:- ~~~~~~~~~~

prog3:- package classtypecasting
public class mainclass1
{
psvm(String[] args)
{
S.O.P("~~~~~~~~~~");
Demo1 d1 = (Demo1) new Demo2();
// d1 is ref var Demo1 type, object Demo2 type
// Demo2 type is casted to Demo1 type
S.O.P(d1.x);
d1.m1();
S.O.P("~~~~~~~~~~");
}
}



prog1:- same behind prog
prog2:- ~~~~~~~~~~
prog3:- package classtypecasting;
public class Demo3 extends Demo2
{
int z=76;
void m3()
{
S.O.P("running m3() method....");
}
}

prog4:- package classtypecasting;
public class mainclass1
{
psvm(String[] args)
{
S.O.P("~~~~~~~~~~");

Demo2 d1 = (Demo2) new Demo3(); → intuitive type casting Demo1 & Demo2 properties have also accessed

// Demo3 type casted to Demo2 type

S.O.P(d1.x);

S.O.P(d1.y);

d1.m1();

d1.m2();

we can remove this by implicit cast (upcasting)

Demo1 d2 = (Demo1) new Demo3(); → in this type casting only accessing the Demo1 properties.

// Demo3 type casted to Demo1 type

S.O.P(d2.x);

d2.m1();

S.O.P("");

}

}

prg1:- same as privious prg

prg2:- same as privious prg

prg3:- same as privious prg

prg4:- package classtypecasting;

public class main class2

{

PSVM(String[] args)

{

S.O.P("");

Demo3 d1 = new Demo3(); → in this type casting accessing all the

Demo2 d2 = d1; // up casting properties Demo1, Demo2, Demo3.

// Demo3 type casted to Demo2 type

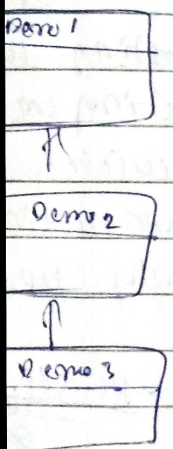
Demo1 d3 = d1;

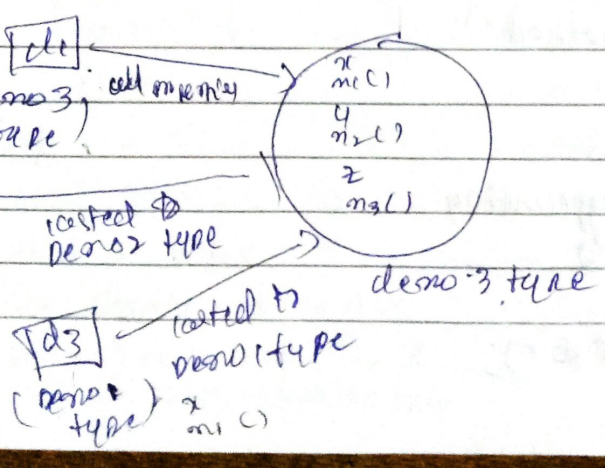
// Demo3 type casted to Demo1 type

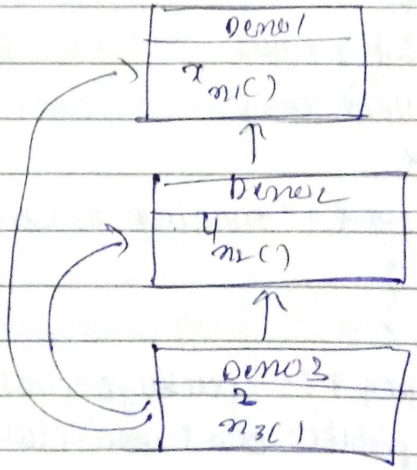
S.O.P("");

}

}

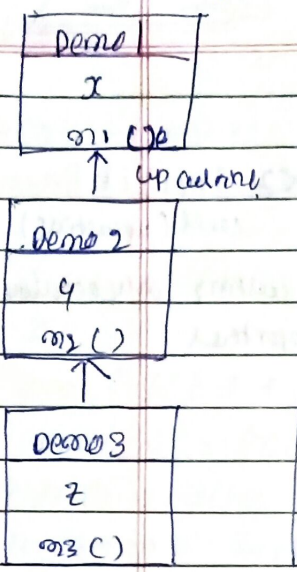






Down casting

Casting super class type to subclass type.



prg1:-

```
package classtypecasting;
public class Demo1
{
int x=10;
void m1()
{
S.o.p("running m1() method");
}
}
```

prg2:-

```
package classtypecasting
public class Demo2 extends Demo1
{
int y=20;
void m2()
{
s.o.p("running m2() method");
}
}
```

prg3:-

```
package classtypecasting
public class Demo3 extends Demo2
{
int z=30;
void m3()
{
s.o.p("running m3() method");
}
}
```

prg4:- package classtypecasting

```
public class maininclass2
{
  S.o.p("***
```

```
Demo1 d1 = new
Demo2 d2 = (Demo
S.o.p("***
}
}
```

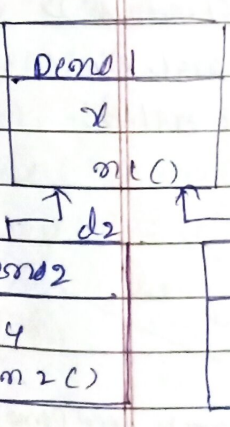
prg1:- same as pr

prg2:- same as

prg3:- same as

prg4:- package cl

```
public class mai
{
p.s.v.m(String[]
{
s.o.p("***
Demo1 d1= new
// access only
Demo2 d2= (De
// access Demo1
Demo3 d3= (De
s.o.p // access
s.o.p("***
}
}
```



case2:-

```
Demo2 ref1
ref1 = new
ref1 = new
Downcasting
ref1 = (Demo
```

27/04/19

```
Demo1 d1 = new Demo2(); // upcasting
Demo2 d2 = (Demo2) d1; // Down casting
S.O.P ("*********");
}
}
```

Note - upcast object can only be down cast

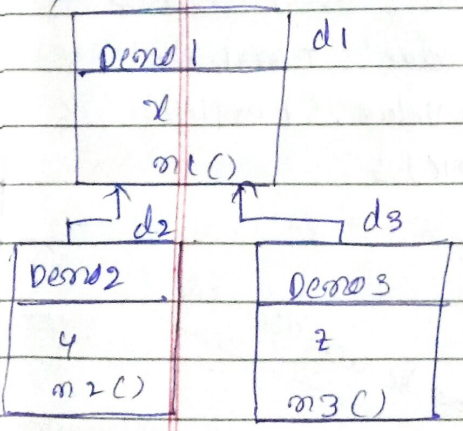
prg1 :- same as previous

prg2 :- same as previous

prg3 :- same as previous

prg4 :-

```
package classtypecasting;
public class mainclass3
{
p.s.v.m (String[] args)
{
S.O.P("*********");
Demo1 d1 = new Demo3(); // up casting
// access only Demo1 properties
Demo2 d2 = (Demo2) d1; // Down cast
// access Demo1 and Demo2 properties
Demo3 d3 = (Demo3) d1; // Down cast
// access Demo1, Demo2 and Demo3 properties
S.O.P("*********");
}
}
```



Case 1 :

```
Demo1 ref1;
ref1 = new Demo1();
ref1 = new Demo2(); upcasting
ref1 = new Demo3(); upcasting.
```

Case 2 :-

```
Demo2 ref1;
ref1 = new Demo2(); ✓
ref1 = new Demo1(); X
```

Downcasting directly not happens

```
ref1 = (Demo2) new Demo3(); X
```

no IS-A relationship

Case 3 :-

```
Demo3 ref1;
ref1 = new Demo3(); ✓
ref1 = new Demo1(); X
ref1 = new Demo2(); X
```

Function Arguements

8 types

1) primitive Arguements
2) non-primitive Arguements

① void m1(int arg)
{ ... }
primitive type

caller function must pass primitive value (int type)
eg :- m1(10);
passing value

② void m2(Demo1 arg)
{ ... }
Java type (class, enum, interface, annotation)

caller function must pass object of Java type (Demo1 type object)
e.g :- m2(new demo1());
passing object ref

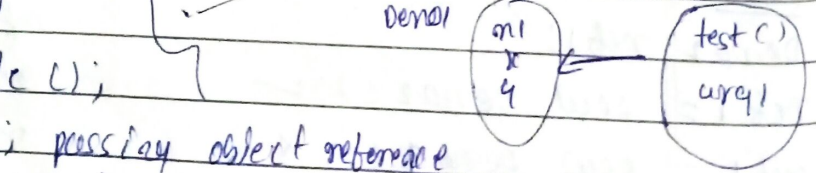
prg1 :- package Isplidersx.p1;
public class Demo1
{
int x = 10;
double y = 34.13;
void m1()
{
S.O.P("running m1() method");
}
}

prg2 :- package isplidersx.p1;
public class Sample1
{
void test(Demo1 arg1)
{ // argument of Demo1 type
// test() method use the properties
// of Demo1 type
S.O.P("running test() method");
S.O.P("x value :" + arg1.x);
S.O.P("y value :" + arg1.y);
arg1.m1();
}
}

prg3 :- package isplidersx.p1;
public class mainclass
{
psvm(String[] args)
{
S.O.P("main class started")
Sample1 s1 = new Sample();
s1.test(new Demo1()); passing object reference
S.O.P("main class ended");
}
}

can be written as

Demo1 arg1 = new Demo1()



prg1 :-
prg2 :-
public c
{
void t
{
S.O.P("
C.O.P("
arg1.
arg1.x
arg1.y
}
}
prg3 :- p
public
{
psvm(
{
S.O.P("
Demo1
Sample1
s1.t
S.O.P(
S.O.P(
}
}

prg1:- same as previous prg

prg2:- package sridhar.p1
public class sample1
{
void test (Demo1 arg1)
{
s.o.p ("x value:" + arg1.x);
s.o.p ("y value:" + arg2.y);
arg1.m1();
arg1.x = 100;
arg1.y = 234.12;
}
}

Prg3:- package sridhar.p1;
public class mainclass1
{
psvm (String[] args)
{
s.o.p ("main class started");
Demo1 d1 = new Demo1();
Sample1 s1 = new sample();
s1.test (d1);
s.o.p ("x value:" + d1.x);
s.o.p ("y value:" + d1.y);
}
}

m2 (Demo1 arg)
↓
Java type (class, enum, interface, annotation)

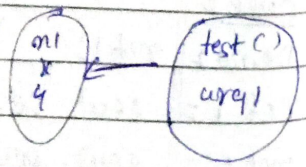
unction must pass object
type (Demo1 type object)
m2 (new demo1());
↓
passing object ref

ckage sridhar.p1;
class Sample1

est (Demo1 arg1)
equivalent of Demo1 type
) method use the properties
emo1 type
nting test() method");
value:" + arg1.x);
value:" + arg1.y);
();

n be written as

Demo1 arg1 = new Demo1();



9|7|19

Function return type

2 - ~~types~~ return type

1) primitive type
2) non primitive type

eg 1) **primitive**

```
int f1 ()
{
   ≡ } operation
}
   return val;
}
```

calling ~~st~~stmt

```
int res = f1 ();
```

eg 2) non primitive

```
Demo1  f2()
{
   ≡ } operation
}
return ref1;   → returning object ref
}
```

callingstmt

```
Demo1 d1 = f2();
```

prg1 :- Demo1 → previous program of function argument
prg2 :- sample1 → previous — " —
prg3 :- mainclass1 (if ull)
prg4 :- sample2

```
package jspiders.p1;
public class sample2
{
 Demo1 disp() <
 {
  s.o.p ("running disp() method.");
  Demo1 ref1 = new Demo1();
  return ref1;
 }
}
```

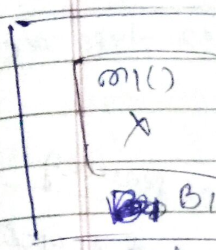
```
s.o.p (ref2.x);
s.o.p (ref2.y);
ref2.m1();
s.o.p ("...
```

prg5 :-

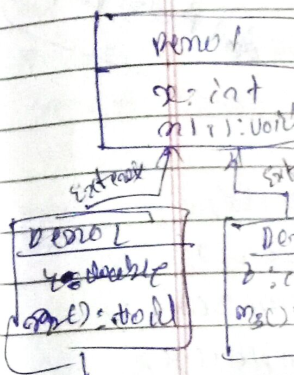
```
package jspiders.p1;
public class mainclass2
{
 psvm (string [] args)
 { s.o.p ("...");
   Sample2 s1 = new sample2();
   Demo1 ref1 = s1.disp()
```

type

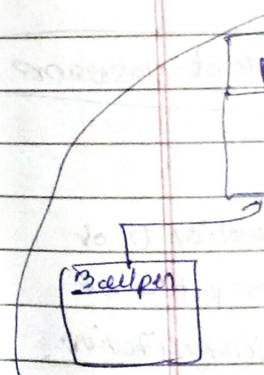
object creation



object c



child class (os
sub class



→ it's out
args and
done it

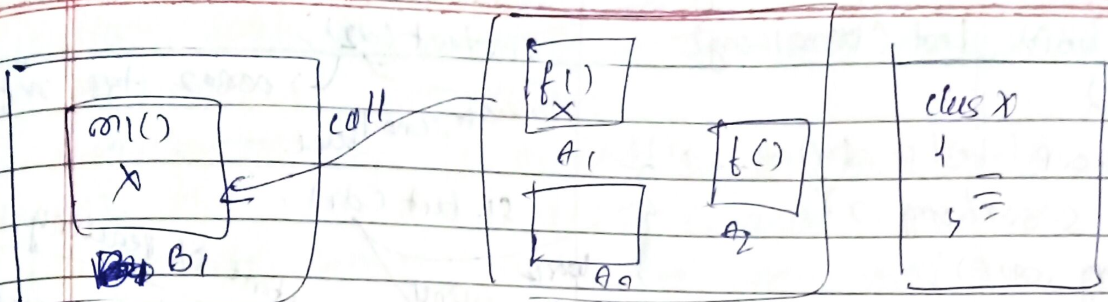
Specification

class
{
void de
{

9/7/19

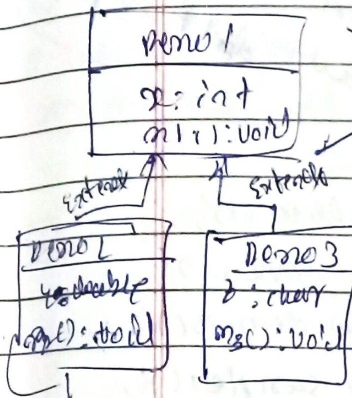
project creation

f1() creates B1 object



object utilization   object creation   Def class

Function argument



child class (or) sub class

Demo1 d1 = new demo1();
Demo2 d2 = new demo2();
Demo3 d3 = new demo3();
sample1 s1 = new sample1();
s1.test(d1); ✓ upcasting
s1.test(d2); ✓
s1.test(d3); ✓

object are created and ref d2 and d3 are also pointing to the demo1() & demo3() so d1=d2=d3

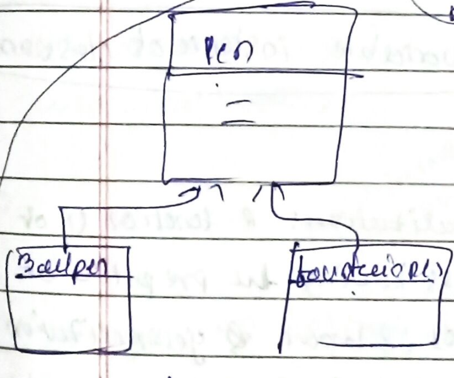
if you send same argument it will occur upcasting

class sample1
{
void test(Demo1 arg)
{
s.o.p("test() started...");
s.o.p(arg.x);
arg.m1()
}

upcasting is default done

demo1 d1 = new demo1();
demo1 d2 = new demo2();
demo2 casted to demo1 type.
demo3 casted to demo1 type

class student
{
void write notes (Pen p1)
{
p1 ....
}



it's not execute but implicit by same as demo1 type args and type cast is different so if type cast is done it will execute → we called this process → "generalization"

Specification

class sample1
{
voide test (Demo2 arg)
{

downcasting
s1.test(d1); X
s1.test(d2); ✓
s1.test(d3); X
↓
no relation

next
same class
same args

generalization program

```
class sample1 demo1 arg = d2;
{
void test (Demo1 arg)
{
s.o.p ("test() started...");
s.o.p (arg.x);
arg.m1();
Demo2 ref2 = (Demo2) arg;
s.o.p (ref1.y);
ref1.m2();
}
```

Generalization specialization

```
class sample1
{
void test (Demo1 arg)
{
s.o.p ("test() started...");
s.o.p (arg.x);
arg.m1();
if (arg instanceof Demo2)
{
Demo2 ref = (Demo2) arg;
sop (ref1.y)
ref1.m2();
}
else if (arg instanceof Demo3)
{
Demo3 ref2 = (Demo3) arg;
s.o.p (ref.z);
ref.m3()
}
}
```

upcasted

upcasted

```
Demo1 d1 = new Demo1();
Demo2 d2 = new Demo2();
Demo3 d3 = new Demo3();
sample1 s1 = new sample1();
s1.test(d2);
```

→ Demo2 type ref casted to Demo1 type.

s1.test(d1);

down casting not work

this start is reading

if we write d2 arg it will execute Demo2 and if we write d1 it will not execute d3 properly.

```
Demo1 d1 = new demo1();
Demo2 d2 = new demo2();
Demo3 d3 = new demo3();
sample s1 = new sample1();
s1.test(d2);
s1.test(d3);
```

Ref variable instance of classname

Generalization: A function () of subclass having the properties of superclass is known as generalization.

in generalization & specialization we have to use extends (or) inheritance

to achieve generalization arguements have

any function object is known

the generalization argument. in any functional type of object developing function of object specialization arguements

prog1:- package

```
public cl
{
int age;
String name;
public person(
{
this.age = a
this.name
}
}
```

Prog3:- package

```
public class
{
int empI
double sa
public Empl
{
super(
this.ea
this.sal
}
```

to achive generalization can be defining the function arguements as base superclass

10/4/19

Bafna Gold
Date: Page:

* any function developed to work more than one type of object it known as "generalization"
* the generalization can be achived by declaring function argument. in a projected if we want implement any functionality which works for more than one type of object then we go for generalization
* developing the functionality which works only one type of object it known as specialization
* specialization can be achive defining the function arguments as subclass type

Function arguments with generalization and specialization

prog1:- package com.spiders.p1;
public class person
{
int age;
string name;
public person(int age, string name)
{
this.age = age;
this.name = name;
}
}

prog2:- package com.spiders.p1;
public class student extends person
{
double stmarks;
int stid;
public student(int age, string name, double stmarks, int stid)
{
super(age, name); // call the constructor of super class from subclass
this.stmarks = stmarks;
this.stid = stid;
}
}

prog3:- package com.spiders.p1;
public class Employee extends person
{
int empId;
double empSalary;
public Employee(int age, string name, int empId, double empSalary)
{
super(age, name);
this.empId = empId;
this.empSalary = empSalary;
}
}

prg 4 :- package com.jspiders.p1;

public class Government service
{
void Aadhar enroll (person arg) → class type argument non primitive
}
s.o.p("Enrolling to aadhar");
s.o.p(arg.name+", Enrolled successfully"); → you can give a arg of student no problem bcz it's class type argument it is also generalization bcz argument type in super class
}
void taxpayment (Employee arg) → specialization its only works when we pass same argument which means sub class
}
double taxAmt = (arg.empsalary * 30)/100;
s.o.p("tax amount Rs" + taxAmt);
} → specialization its only works when we pass same argument which means student argument only then works which means sub class.
void scholarship (student arg)
{
if (arg.stmarks >= 70.00)
}
s.o.p(arg.name+", eligible for scholarship");
}
else
{
s.o.p(arg.name+", not eligible for scholarship");
}
}
}

prg 5 :- package com.jspiders.p1;
public class mainclass1
{
public s v m (String[] args)
{
s.o.p("........");
person p1 = new person (21, "ramesh");
student s1 = new person (17, "suresh", 75.00, 3872);
Employee e1 = new employee (24, "mahesh", 3287, 80000.00);

Government se
gov. Aadhar
gov. Aadhar
gov. Aadhar
gov. scholars
gov. tax pay
s.o.p("
}
}

* an object of it's
1) compilation
* compile time
* to method
compilation
communication
known as
can't be
* method
polymorphism
* in run time
by the
happened at
binding also
* method
* the runtime
1) IS A
2) method
3) typecast
prg 1: pack
public class
{
void
s.o.p(
}
}

Government services gov = new government services();
gov. AadharEnroll(p1); → person class object
gov. AadharEnroll(s1); → student class object
gov. AadharEnroll(e1); → employee class object
gov. scholarship(s1);
gov. tax payment(e1);
s.o.p("...");

} → if you want to display student ID or marks in Adhar enroll you have down cast same as employee class

}

polymorfism (polymorphism)

* an object showing different behaviour at different stages of it's life cycle is known as "polymorfism". 2 types
1) compiler poly 2) runtime polymorfism
* compile time polymorfism method declaration is binded to method defination by the compiler at time of compilation. since the binding is done at the complication time is known as early binding it is also known as static binding because once the binding is done can't re binding.
* method over loading use to achive compile time polimorfism
* in run time polimorfism the method declaration is binded by the jvm during execution. since the binding happen at run time it is known as late binding. the binding can be rebinding hence is known as dynamic binding.
* method over riding is use to achive run time polimorfism
* the runtime polimorfism can be achived by following concept
1) IS A relationship
2) method overriding
3) typecasting

run time

```
pg 1: package com.jspiders.p1;
public class Demo1
{
  void wish() {
    s.o.p("Hi sir...!");
  }
}

class Demo2 extends Demo1
{
  void wish()
  {
    s.o.p("Bye sir.....!");
  }
}
```

prg2: package com.ispiders.p1;
public class mainclass
{
p s v m (String [] args)
{
s.o.p("**** ****");
demo1 d1 = new demo2(); → here we done method overriden so implemented method will be executed
d1.wish();
s.o.p("**** ...");
}
}

o/p → Bye sir

prg1: package com.ispiders.p1;
public class animal
{
void noise()
{
s.o.p("animal makes noise....");
}
}

prg2: package com.jspiders.p1;
public class cat extends animal
{
void noise()
{
s.o.p("meow....meow");
}
}

prg3: package com.jspiders.p1;
public class dog extends animal
{
void noise()
{
s.o.p("Bow....Bow....");
}
}

prg4: package com.ispiders.p4;
public class snake extends animal
{
void noise()
{
s.o.p("Hisss....Hisss....");
}
}

prg5: package com.ispiders.p1;
public class animalsimulator
{
void make noise (animal arg) → super class (generalization)
{ arg.noise();
}
}

prg: package com...
public class ...
p s v m (S...
{
s.o.p("...
cat c1 = ...
Dog d1 = ...
snake s1 ...
animal s...
a1. make...
a1. mak...
a1. mak...
s.o.p ...
}
}

prg1: pack...
abstract ...
{ abstr...
}

package coor.inheritance.p1;
public class mainclass2
{ public static void main(String[] args)
{
S.O.P("main method started");
cat c1 = new cat();
dog d1 = new dog();
snake s1 = new snake();
animalstimulator a1 = new animalstimulator();
a1.makenoise(c1);
a1.makenoise(d1);
a1.makenoise(s1);
S.O.P("main method ended");
}
}

pgm: package coor.inheritance.p1;
abstract public class animal
{ abstract void noise();
}

prog2: snake as private ex
prog3: Cannot
prog4:
prog5:
prog6: