object oriented programing system/structure - oop

oop is a programing paradigm/methodology

1. object oriented paradigm
2. procedual paradigm
3. functional paradigm
4. logical paradigm
5. structual paradigm

6 main pillars of oops

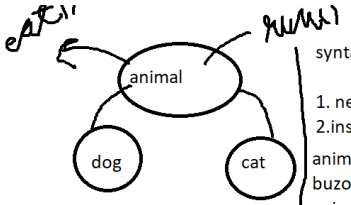
1. class
2. objects & methods
3. inheritance
4. polymorphism
5. abstraction
6. encapsulation

classes and objects :

1. class is the collection of objects.
2. class is not a real world entity. it is just a template or blueprint or prototype.
3. class do not occupy memory.

syntax:
access-modifier class classname
{
methods
constructors
fields
blocks
method class
}

methods :
1. A set of codes which perform a particular task.
advantages:
1. code reusability
code optimization.
syntax:
accessmodifier return type method name(list of parameters)
{.....
.....
}



syntax:

1. new keyword
2.instance methods

animal buzo;
buzo=new animal();
animal buz0=new animal();

objects :
1. object is an instance of class.
2. object is real world entity.
3. objects occupies memory.
consist of:;
1. identity ---- name(buzo)
2. state/attributes- color, bread,age
3. behaviour- eat, run ,bark

```
class Animal
{
public void bark()
{
sopln("do barks");
}
public void sleep()
{
sopln("cat always sleep");
}
psvm()
{
Aminal dog=new Animal();
dog.bark();
Animal Cat= new Animal();
cat.sleep();
}
```

```
class Am{
int a;
double b;
}
class Pm{
psvm()
{
Am obj=new Am();
Am obj2 = new Am();
obj.a=10;
obj.b=1.2;
obj1.a=20;
obj1.b=2.22;
sopln("a="+obj.a+"and"+obj1.a);
sopln("b="+obj.b+"and"+obj1.b);
}
```

constructors:

1. It is a block/ special method having same name as class name.
2. it does not have any return type not even void.
3. always specified in public visibility mode.
4. it linked with the object whenever the object is created the constructor is called automatially.
5. calling is not required.

Default constructor
(no-argument constructor)
-compiler

```
class test
{
test()
{
}
psvm()
{
test obj=new test();
}
}
```

No-arg constructor:
(user- define constructor)

```
class test
{
test()
{
sopln("welcome");
}
psvm()
{
test obj=new test();
}
}
```

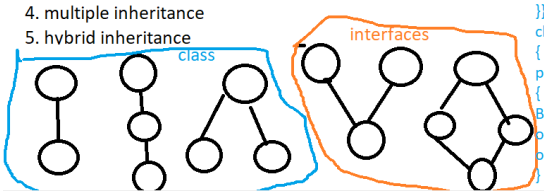
parametrised constructor

```
class test
{
test(string name)
{
sopln(name);
}
psvm()
{
test obj=new test("sakshi");
}
```

inheritance:
it is inheriting the properties of parent class into clild class.

inheritance is the process by which one object acquires all the properties and behavious of a parent object.

types of inheritance :
1. single inheritance
2. multilevel inheritance
3.hierarchical inheritance
4. multiple inheritance
5. hybrid inheritance



single

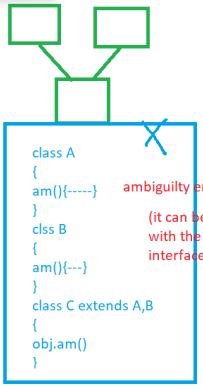
```
class A
{
public void am()
{
sopln("welcome");
}}
class B extends A
{
pubic void pm()
{sopln("home");
}}
class c
{
psvm()
{
B obj=new B();
obj.am();
obj.pm();
}
```

multilevel

```
class A
{
public vois am()
{-------}}
class B extends A
{
public pm()
{-----}}
class c extends B
{
public void gm()
{----}}
class d
{
psvm()
{c obj=new c();
obj.am();
obj.pm();
obj.gm();}}
```

hierarchical

```
class A
{
pubic void am(){----}
}
class B extends A
{
public void pm(){----}
}
class C extends A
{
public void gm(){----}
}
class d
{
psvm()
{ B obj=new B();
obj.pm();
obj.am();
C obj2=new C();
obj2.gm();obj2.am();}
```



Relationships

```
class A
{
------
}
class B
{
-------
}
```

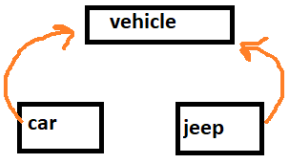
types of relationshhip

IS-A relationship (inheritance) (tight coupling)

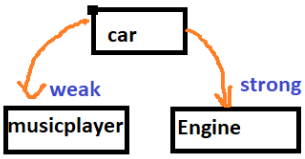
HAS-A relationship (Association)

- aggegation (weak)
- composition (strong)

advamtages :
1. code resuability
2. cost cutting
reduce redudancy



car is a vehicle
jeep is a vehicle



cas has a engine
car has a music player

polymorphism :
one name and multiple forms.

water: soild,liq,gas
shapes:circle,triangle,rectangle

types:
1. compile time polymorphism(compiler)
static polymorphism
method overloading(achieve)
2. run time polymorphism (JVM)
Dynamic polymorphism
method overridding(achieve)

method overloading	method overriding
1. same name of methods	1. same name
2. same class	2. different class
3. different arguments	3. same arguments
- no of arguments	-no of arg
-sequence of arguments	-seq of arg
- types of arguments	-types of arg

method overloading

```
class test{
public void show()
{sopln("hello");}
public void show(int a)
{sopln(a);}
public void show(double a)
{sopln(a);}
public void show(int a,double b)
{sopln(a+b);}
public void show(double b,int a)
{sopln(a+b);}
psvm()
{
test obj=new test();
obj.show();
obj.show(12);
obj.show(1.2);
obj.show(12,13);
obj.show(1.2,2.3);}}
```

method overriding

```
class test{
public void show(string a, int b)
{
sopln(a);sopln(b);
}
class xyz extends test
{
public void show(string a, int b)
{
sopln(a); sopln(b);
}
psvm()
{
test obj= new test();
obj.show("sakshi",10)
xyz obj2= new xyz()
obj2.show("goyal",12);
}
}
```

oops
class
objects & methods

inheritance
has-a relationship
is-a relationship
polymorphism
-method overloading
-method overriding

code resuability

abstraction
data hiding
encapsulation
tightly coupled classes

security

Abstraction and Encapsulation

abstraction	encapsulation
1. Abstraction is detail hiding (implementation hiding)	1. encapsulation is data hiding (information hiding)
2. Data abstraction deals with exposing the interface to the user and hiding the details of implementation	2. encapsulation groups together data and methods that act upon the data.

binding the data in a single unit is called encapsulation.

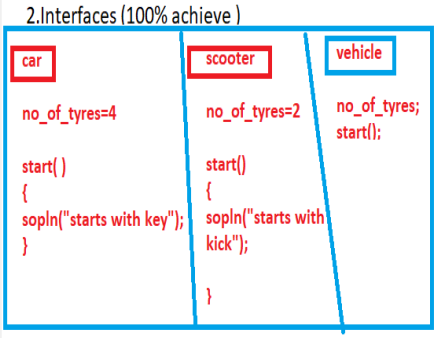
```
class Employee
{
private int empId;
public void SetempId(int eid)
{
empId= eid;
}
public void getempid()
{
return empid;
}
class Company {
psvm()
{
Employee ob=new Employee();
ob.setempid(12);
ob.getempid();
}}
```

Abstraction: (Detail Hidding)

Abstraction is hiding implementation & just highlighting the setup services that we are off

Abstraction can be achieved in two ways:

1. Abstract class(0-100% achieve)
2. Interfaces (100% achieve)



key points :

1. A method without body (no implementation) is known as abstract method.

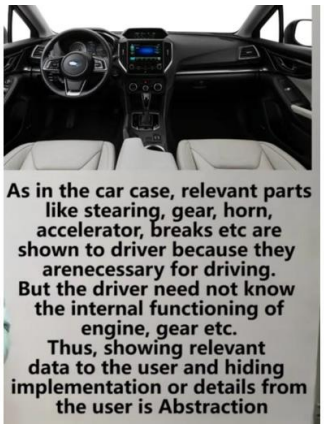
2. A method must always be declared in an abstract class, or we can say that if a class has an abstract method, it should be declared abstract as well.
if any class is abstract class it's not necessary that the method should be abstract.

3. if a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

4. Abstract methods in an abstract class are meant to be overridden in derived concrete classes otherwise compile-time error will be thrown.
that means it follows method overridng concept.

5. Abstract class can not be instantiated means we can not create an object of Abstract class.

```
abstract class vehicle
{
abstract  void start();
}

class car extends vehicle
{
 public void start()
{
sopln("car starts with keys");
}
}

class scooter extends vehicle
{
public void start()
{
sopln("scooter starts with
kick");
}

psvm()
{
car c=new car();
c. start();

scooter s= new scooter();
s.start()
}
}
```



Interfaces :

interfaces are similar to Abstract class but having all the methods of abstract type.

interfaces are the blueprint of the class. it specify what a class must do and not how.

1.it is used to achieve abstraction.
2. it supports multiple inheritance.
3. It can be used to achieve loose coupling.

syntax:

```
Interface interfacename
{
methods // abstract public
fields  //public static final
}
```

note: if you don't write public abstract in front of method name then compiler will write it by itself

if you don't write public static final in front of field name compiler will write it by itself.

we can use concrete method in java but . . their access modifier should be default — 8th version

we can also create static methods inside the interface. but its access modifier should be public. else by default compiler will write public.

now we can create private methods also in interfaces. — 9th version

we can not create the object of an interface.

interfaces also supports method overriding concept.

```
interface I1
{
void show()
}
```

```
interface I1
{
public abstract void
show();
}
```

```
interface I1
{
void show();
void display()
{
-------
}
}
```

```
interface I1
{
void show();
default void display()
{
-------
}}
```

```
interface I1
{
void show();
static void display()
{
-----------
}
}
```

```
interface I1
{
void show();
protected void display()
{
-------
}
}
```

```
interface I1
{
void show();
private void display()
{
------
}
}
```

```
interface I1
{
int a=10;
public static final b=20;
void show();
}
```

```
interface I1
{
void show();
}
class test implements I1
{
public void show()
{
system.out.println("hello");
}
psvm()
{
//I1 i=new I1();(not allowed)
test t=new test();
t.show();
}
}
```

inheritance supports multiple inheritance

```
interface I1
{
void show();
}
interface I2
{
void display();
}
class test implements I1,I2
{
public void show(){
sopln("hello");
}
public void display()
{
sopln("hii");
}
psvm(){
test t=new test();
t.show();
t.display();
}}
```

this keyword

this keyword is the reference variable that refers to the current object.

program without using this

```
class test
{
int i;
void setvalues(int i)
{
i=i;
}
void show()
{
sopln(i);
}
class xyz{
psvm{
test t=new test();
t.setvalues(10);
t.show();
}        output:
}        0
```

program by using this keyword

```
class test
{
int i;
void setvalues(int i)
{
this.i=i;
}
void show()
{
sopln(i);
}}
class xyz{
psvm(){
test t=new test();
t.setvalues();  output:
t.show();       10
}
}
```

here this refers to the current class instance variable.
that means if you give same name to the local and instance variable in that case also you can provide the same value to the instance variable with the help of this keyword.

6 uses of this keyword

1. this keyword can be used to refer current class instance variable.

2.this keyword can be used to invoke current class method(implicity).

3. this() can be used to invoke current class constructor.

4. this can be used to pass an argument in the method call.

5. this can be used to pass an argument in the constructor call.

6. this can be used to return the current class instance from the method.

1. this keyword is used to invoke current class instance variable

program without using this

```
class test
{
int i;
void setvalues(int i)
{
i=i;
}
void show()
{
sopln(i);
}
class xyz{
psvm{
test t=new test();
t.setvalues(10);
t.show();
}       output:
}       0
```

program by using this keyword

```
class test
{
int i;
void setvalues(int i)
{
this.i=i;
}
void show()
{
sopln(i);
}}
class xyz{
psvm(){
test t=new test();
t.setvalues();  output:
t.show();       10
}
}
```

here this refers to the current class instance variable.
that means if you give same name to the local and instance variable in that case also you can provide the same value to the instance variable with the help of this keyword.

2. this keyword can be used to invoke current class method.

```
//program without this
class test
{
void display()
{
sopln("hello");
}
void show()
{
display();
}
psvm()
{
test t=new test();
t.show();
}   output:
}   hello
```

if you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

```
class test
{
void display()
{
sopln("hello");
}
void show()
{
this.display();
}
psvm()
{
test t=new test();
t.show();
}  output:
   hello
```

3. this keyword can be used to invoke current class constructor

```
//without this
class test
{
test()
{
sopln("no arg
constructor");
}
test(int a)
{
sopln("parametrised
constructor");
}
psvm(){
test t= new test();

test t1=new test(10);
}
}
```

output:
no arg constructor
parametrised constructor

if you use this inside any constructor it will call default constructor inside the constructor.

```
class test
{
test()
{
sopln("no arg constructor");
}
test(int a)
{
this();
sopln("parametrised
constructor");
}
psvm(){

test t1=new test(10);
}
}
```

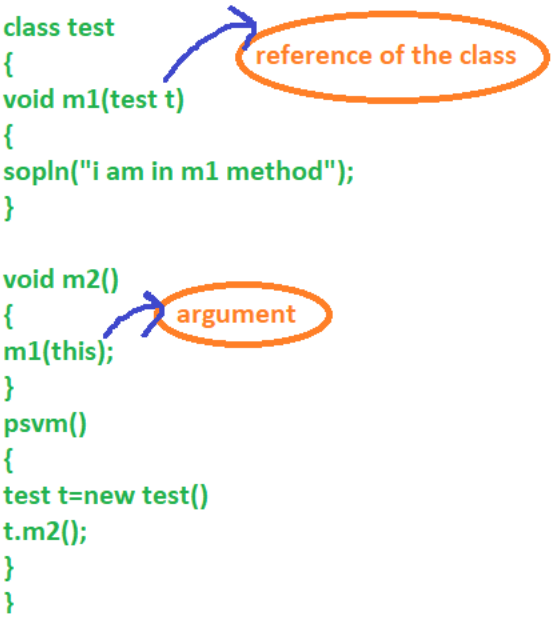
output:
no arg consructor
parametrised constructor

We can also call parametrized constructor inside the default constructor of the current class by writing :

```
Test()

{

this(20);

Sopln(“no arg constructor “);

}
```

We can also achieve constructor chaining with the help of this keyword.

4. this keyword can be used to pass an argument in the method call.




```
class test
{
void m1(test t)
{
sopln("i am in m1 method");
}

void m2()
{
m1(this);
}
psvm()
{
test t=new test()
t.m2();
}
}
```

5. this keyword can be used to pass as an argument in the constructor call

```
class test
{
test(thisdemo td)
{
sopln("test class constructor");
}
}
class thisdemo
{
void m1()
{
test t= new test(this);
}
psvm()
{
thisdemo td=new thisdemo();
t.m1();
}
}
```

6. this keyword can be used to return the current class instance from the method.

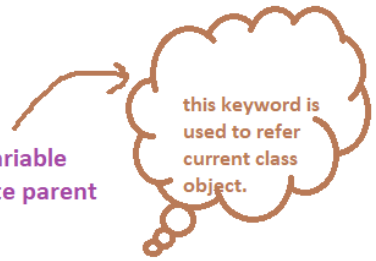
```
class test
{
test m1()
{
return this
}
psvm()
{
test t=new test();
t.m1();
}
```

this program will not show any error.

super keyword:

super keyword is a reference variable which is used to refer immediate parent class object.

super mean parent and we know where the parent will come there will be concept of inheritance.




```
class A
{
int a=10;
}
class B extends A
{
int a=20;
void show(int a)
{
sopln(a);       //30
sopln(this.a);  //20
sopln(super.a);  //10
}
psvm()
{
B obj= new B();
obj.show(30);
}
}
```

uses of super keyword:

1. super keyword can be used to refer immediate parent class instance variable.

2. super keyword can be used to invoke immediate parent class method.

3. super() can be used to invoke immediate parent class constructor.

1. Super keyword used to refer immediate parent class instance variable.

```
class A
{
int a=10;
}
class B extends A
{
int a=20;
void show(int a)
{
sopln(a);       //30
sopln(this.a);  //20
sopln(super.a);  //10
}
psvm()
{
B obj= new B();
obj.show(30);
}
}
```

2. super ca be used to invoke immediate parent class method

```
class A
{
void m1()
{
sopln("i am in class A");
}
}
class B extends A
{
void show()
{
super.m1();
}
psvm()
{
B obj=new B();
obj.show();
}
```

```
class A
{
void m1(){
sopln("i am in class A");
}
}
class B extends A
{
void m1()
{
sopln("i am in class B");
}
void show()
{
m1();//i am in class B
super.m1();//i am in class A
}
psvm(){
B obj=new B();
obbj.show();
}}
```

3. super keyword can be used to invoke immediate parent class constructor

```
class A
{
A()
{
sopln("i am in class A");
}
}
class B extends A
{
b()
{
sopln("i am in class B");
}
 psvm()
{
B obj=new B();
}
}
```

output
i am in class A
i am in class B

```
class A
{
A()
{
sopln("i am in class A");
}
}
class B extends A
{
b()
{ super();
sopln("i am in class B");
}
 psvm()
{
B obj=new B();
}
}
```

output:
i am in class A
i am in class B

in this case without writting super function compiler will create super function by its own.