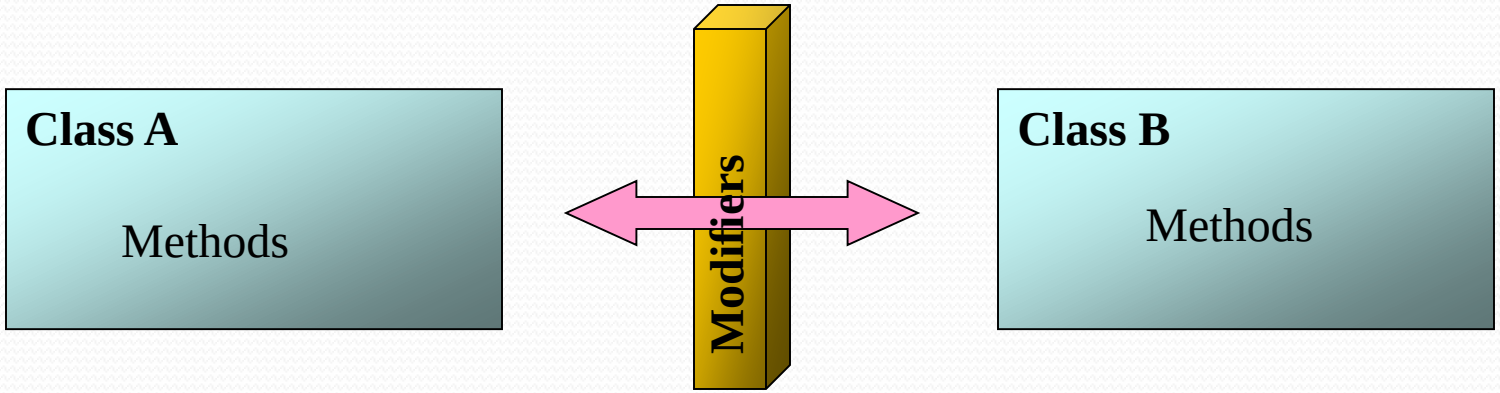
Implementing OOP in Java

Session 4-2

Objectives

- Object Orientation
- Method Overloading
- Encapsulation
- Access Modifiers
- Implement Constructors
- Implement Destructors in
- Explain the working of Garbage Collector
- Static Members
- This pointer

Modifiers - I



Modifiers restrict access to variables by other classes

They are keywords that give additional meaning to variables, code, and classes

Modifiers enhances the encapsulation features of OOP

Modifiers - II

```
class Stack
{
        int[] stak = new int[10];
        int size ;
        int top = -1;
        Stack(int s)
        {
           top = -1;
           size=s;
        }
        void push(int it)
        {
          if(top==this.size-1)
                System.out.println("Stack is full ");
```

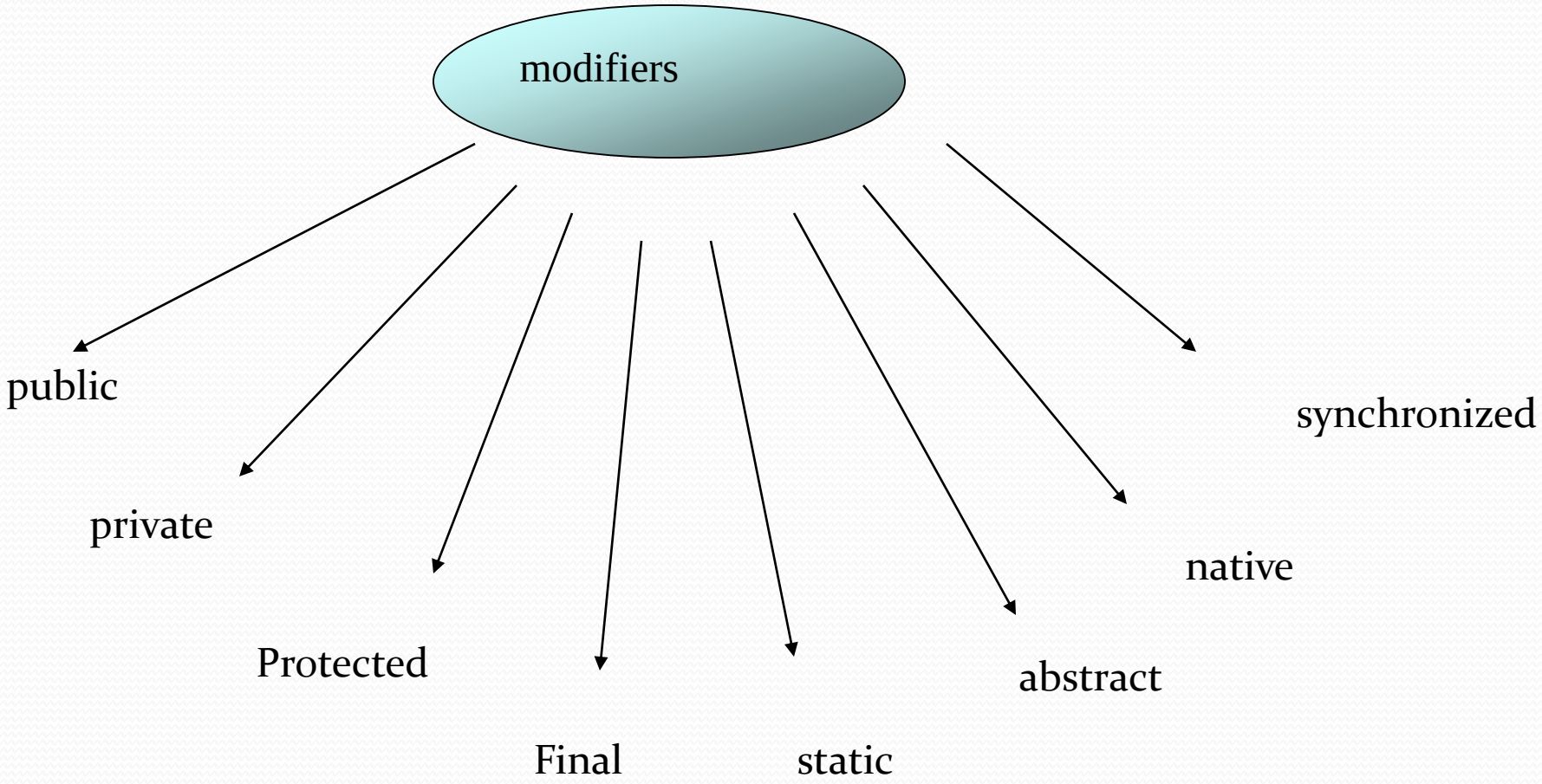
Modifiers - III

```
else
        stack[++top] = it;
        }
        int pop()
        {
`         if(top < 0)
         {
                System.out.println("Stack underflow    ");
                return 0 ;
         }
         else
                return stack[top--];

        }
```

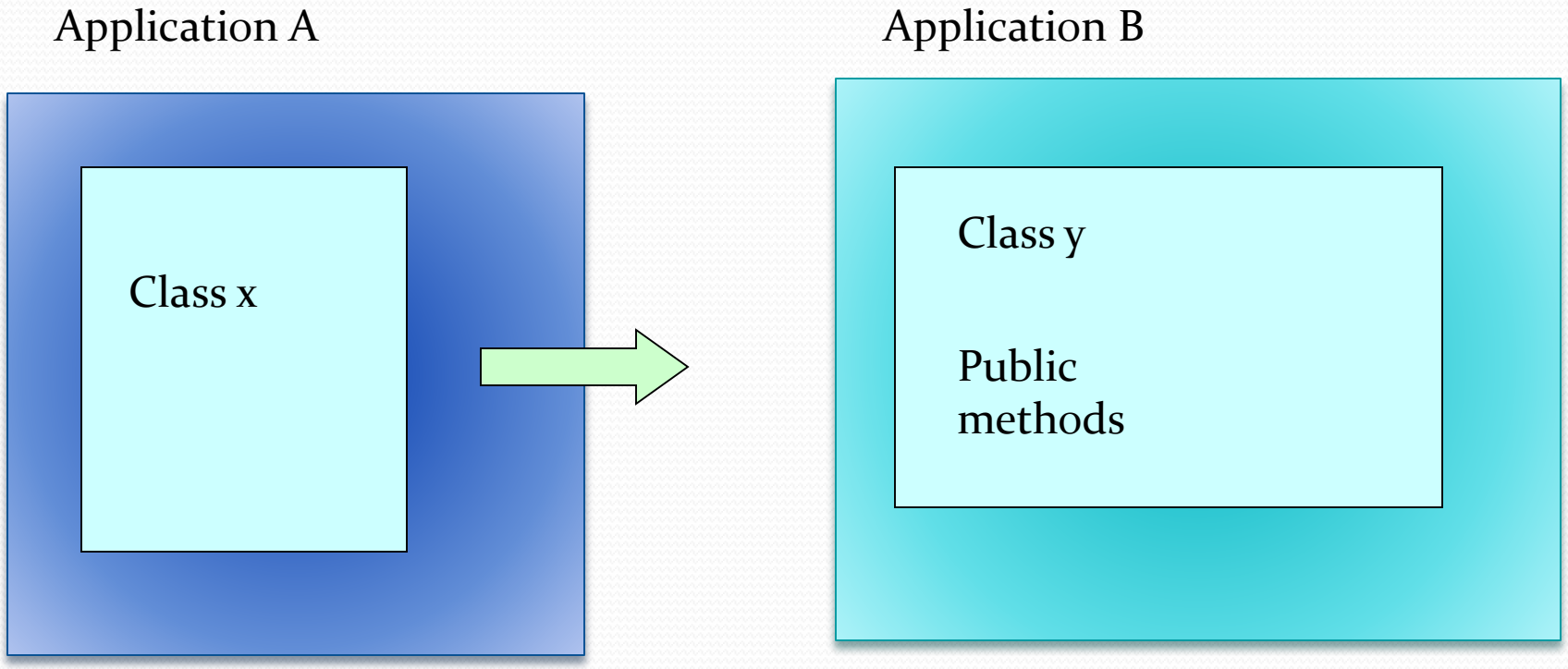
Modifiers - IV

```
class Test {

 Stack S = new Stack();

……..

System.out.println(“The  stack  size  “  +
S.size);

……}
```

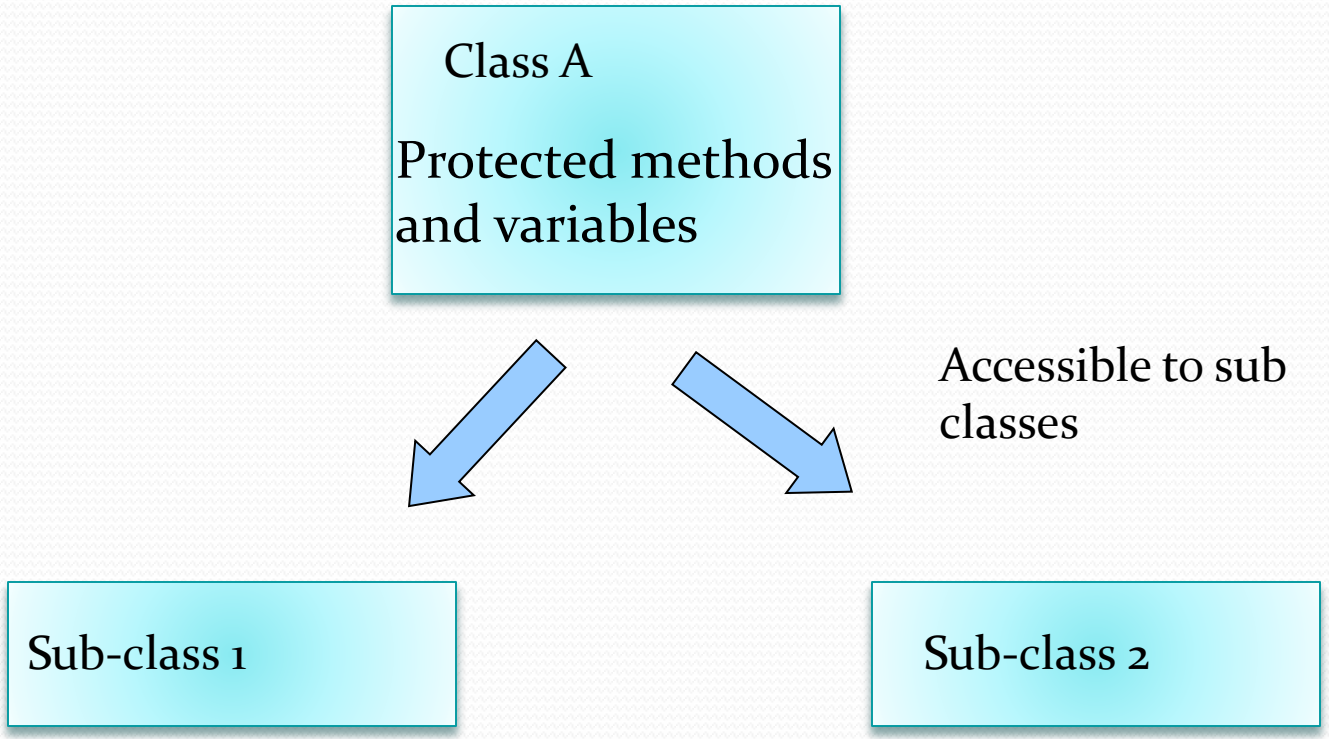
Modifiers - V


modifiers
public
private
Protected
Final
static
abstract
native
synchronized

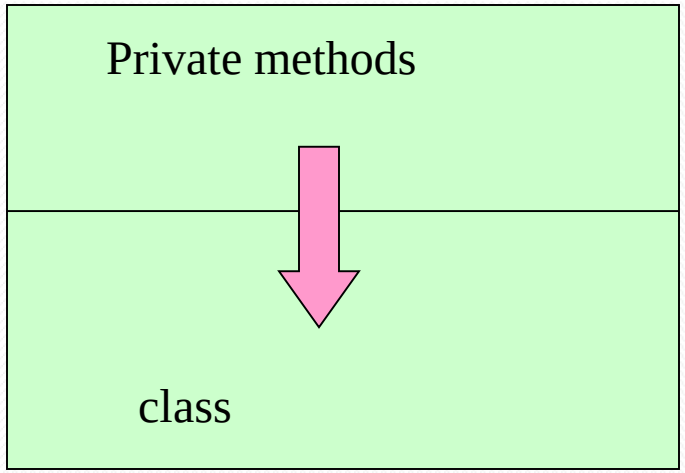
Public



Protected


Class A
Protected methods and variables
Accessible to sub classes
Sub-class 1
Sub-class 2

Private



Methods and data members accessible only to the members of the same class

Class Specification	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package- different class	No	Yes	Yes	Yes
Different package- sub class	No	No	Yes	Yes
Different package- different class	No	No	No	Yes

Static Variables

- Lifetime of static variable is throughout the program life
- If static variables are not explicitly initialized then they are initialized to 0 of appropriate type

Static Data Member

Definition

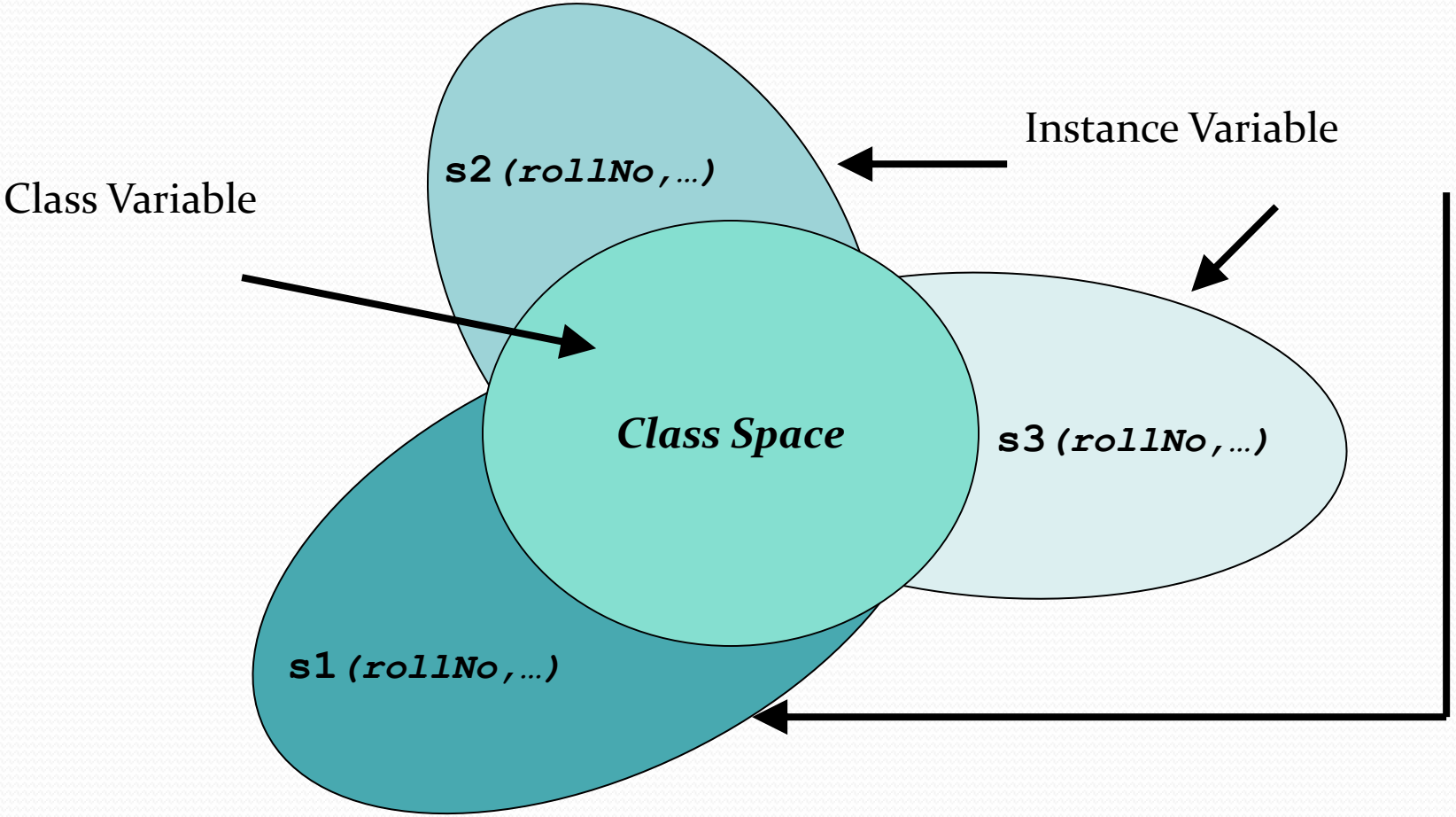
"A variable that is part of a class, yet is not part of an object of that class, is called static data member"

Static Data Member

- They are shared by all instances of the class
- They do not belong to any particular instance of a class

Class vs. Instance Variable

- **`Student s1, s2, s3;`**



Static Data Member (Syntax)

- Keyword static is used to make a data member static

```
class ClassName{
…
static DataType VariableName;
}
```

Initializing Static Data Member

- Static data members should be initialized once at file scope
- They are initialized at the time of definition

Life of Static Data Member

- They are created even when there is no object of a class
- They remain in memory even when all objects of a class are destroyed

Static - I

```
class Cvar
{
 static String name="Aladdin";
 //. .  constructor
 static void showName(){
  . . .
  System.out.println("Static name:" + name);
 }
}
```

Static - II

```
class Staticvar
{
        static int s = 20;
        static void print()
        {
        System.out.println("From the class : " + s);
}
}
class Display
{
        public static void main(String args[])
{
                Staticvar.print();
                System.out.println("From outside the class : " +
            Staticvar.s);
        }
}
```

Uses

- They can be used to store information that is required by all objects, like global variables

Example

- Develop Student class such that one can know the number of student created in a system

Example

```
class Student{
…
public static int noOfStudents=0;
  public Student(){
      noOfStudents++;
  }
…
}
```

Example

```
public static void main(String[] args) {

        System.out.println(Student.noOfStudents);
        Student s = new Student();
        System.out.println(Student.noOfStudents);
        Student s1 = new Student();
        Student s2= new Student();
        Student s3 = new Student();
        Student s4 = new Student();

        System.out.println(Student.noOfStudents);
    }
```

Output:

0

1

5

Problem

- noOfStudents is accessible outside the class
- Bad design as the local data member is kept public

Static Member Function

Definition:

"The function that needs access to the members of a class, yet does not need to be invoked by a particular object, is called static member function"

Static Member Function

- They are used to access static data members
- Access mechanism for static member functions is same as that of static data members
- They cannot access any non-static members

Example

```
class Student{
  private static int noOfStudents;
  int rollNo;
public static int getTotalStudent(){
      return noOfStudents;
  }
}
Class Display{
public static void main(String [] args){
  int i = Student.getTotalStudents();
}
}
```

Accessing non static data members

```
static int getTotalStudents(){
  return rollNo;
}
……
public static void main(String [] args){
  int i = Student.getTotalStudents();
  /*Error: */
}
```

Final

'**Final**' modifier when used with :

Variable	Indicates that once a value is assigned, it cannot be changed
Method	Indicates that method body cannot be overridden
Class	Indicates that this class cannot be inherited

Native

The '*native*' modifier indicates that a method body has been written in a language other than Java, like C or C++

```
native void codeSomeWhere()
    {
    // C / C++ Code
    }
```

Modifiers Snap Shot

Modifier	Method	Variable	Class
Public	Yes	Yes	Yes
Private	Yes	Yes	Yes(Nested Classes)
Protected	Yes	Yes	Yes(Nested Classes)
Abstract	Yes	No	Yes
Final	Yes	Yes	Yes
Native	Yes	No	No

this Pointer

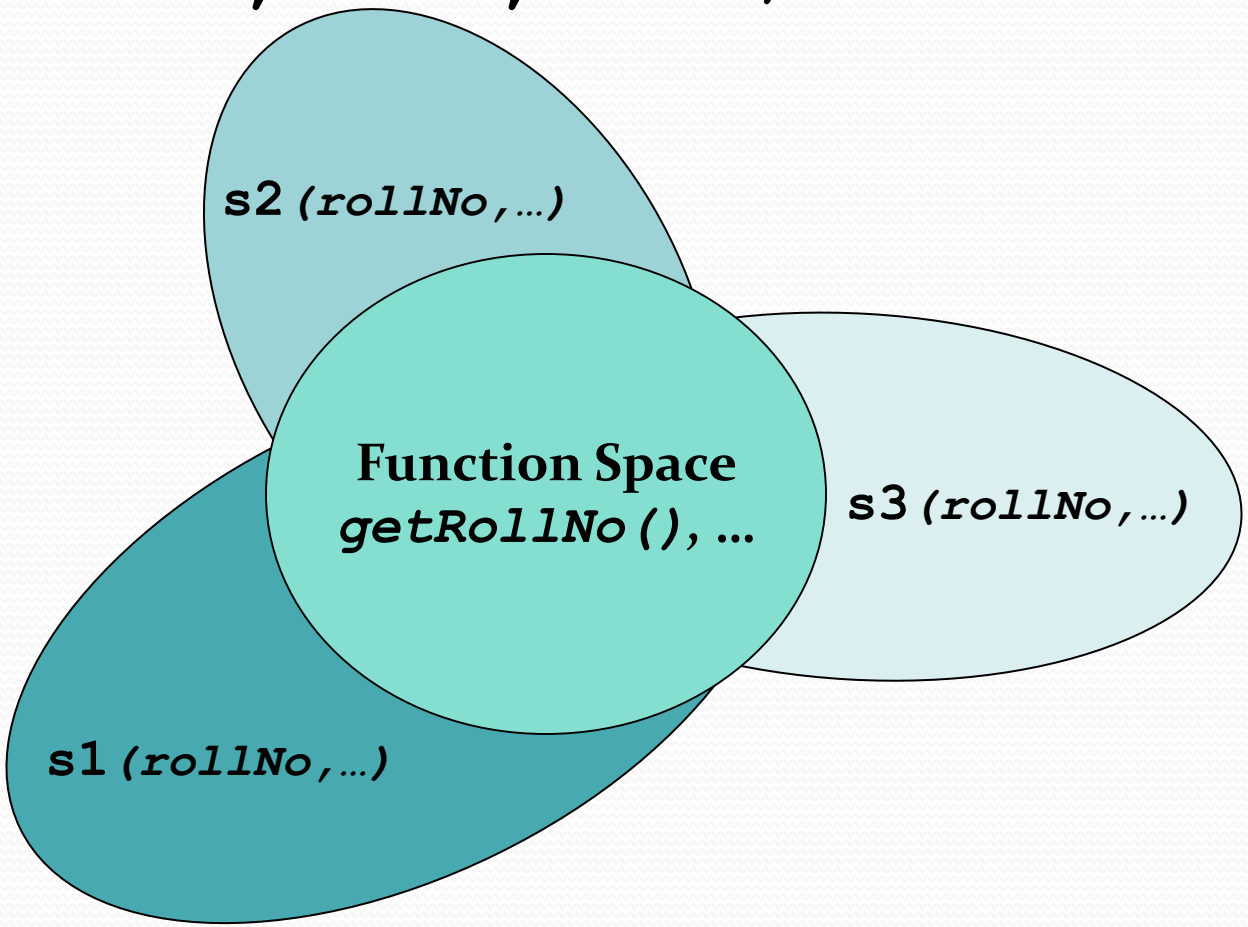
```
class Student{
  private int rollNo;
  ……
public int getRollNo(){…}
public void setRollNo(int aRollNo)
{…}
}
```

this Pointer

- The compiler reserves space for the functions defined in the class
- Space for data is not allocated (*since no object is yet created*)

this Pointer

- `Student s1, s2, s3;`

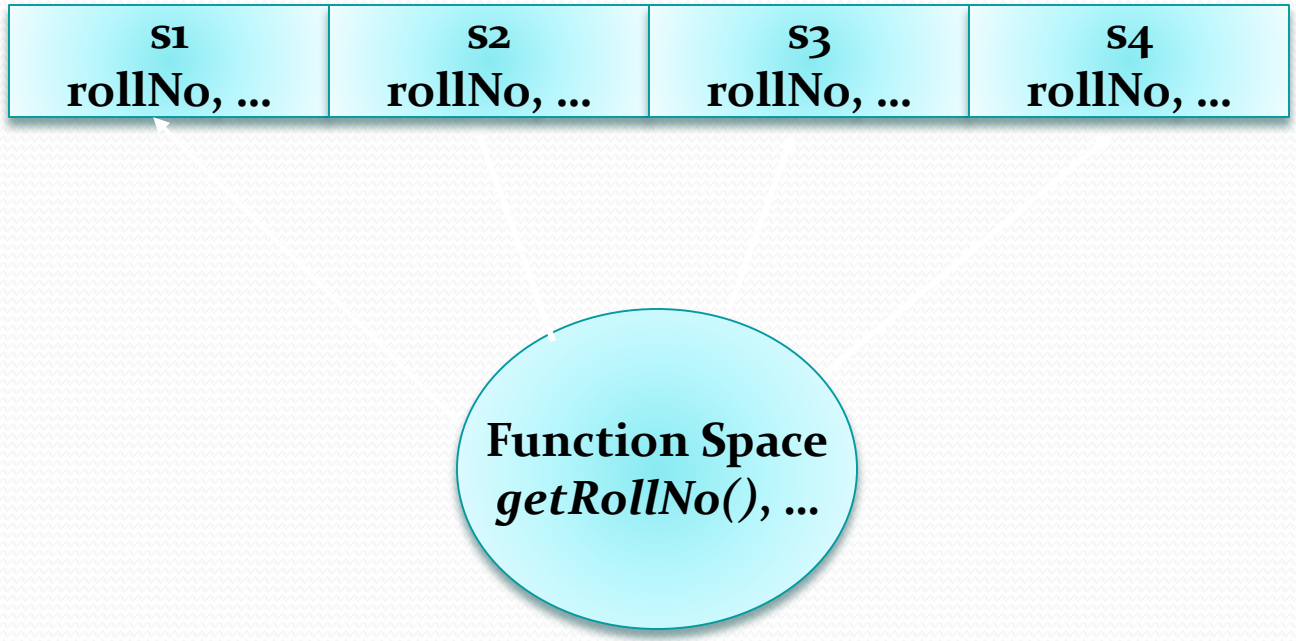


this Pointer

- Function space is common for every object
- Whenever a new object is created:
  - Memory is reserved for variables only
  - Previously defined functions are used over and over again

`this` Pointer

- Memory layout for objects created:



- How does the functions know on which object to act?

this Pointer

- Address of each object is passed to the calling function
- This address is deferenced by the functions and hence they act on correct objects

s1 rollNo, ...	s2 rollNo, ...	s3 rollNo, ...	s4 rollNo, ...
address	***address***	***address***	***address***

•The variable containing the "self-address" is called this pointer

Passing *this* Pointer

- Whenever a function is called the *this* pointer is passed as a parameter to that function
- Function with *n* parameters is actually called with *n+1* parameters

Example

```
void setName(String a)
```

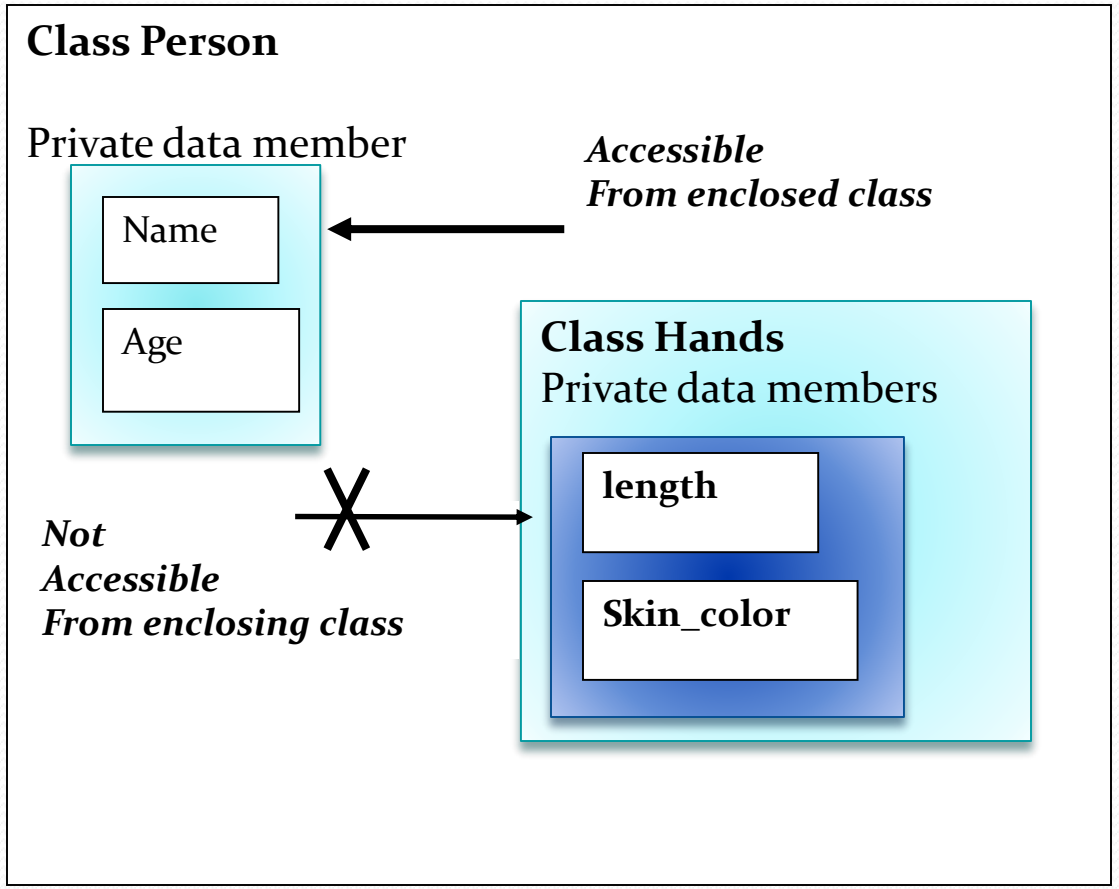
is internally represented as

```
void setName(String a, const Student * this)
```

Compiler Generated Code

```
Student(){
  rollNo = 0;
}

Student(){
  this.rollNo = 0;
}
```

Introducing Nested and Inner Classes



Introducing Nested and Inner Classes

```
class Outer {
        private int out_A =100;

        void access()
        {
        Inner inn = new Inner();
        inn.display();
        }
    class Inner {
        void display() {
        System.out.println("  display  the  private  data  of
       Outer class, out_A =  " + out_A);
        }
    }
}
```

}

Introducing Nested and Inner Classes

```
class Output
{
    public static void main(String args[])
    {
        Outer out = new Outer();
        out.access();
   }
}
```