9.Cohesion and Coupling

Cohesion

Cohesion refers to what the class (or module) can do. Low cohesion would mean that the class does a great variety of actions - it is broad, unfocused on what it should do. High cohesion means that the class is focused on what it should be doing, i.e. only methods relating to the intention of the class.

Example of Low Cohesion:

```
 Staff
------------------
| checkEmail()    |
| sendEmail()     |
| emailValidate() |
| PrintLetter()   |
------------------
```

When a class is designed to do **many different tasks** rather than focussing on a single specialized task, this class is said to be a "low cohesive" class. **Low cohesive classes are said to be badly designed**, as it requires a lot of work at creating, maintaining and updating them.

```
class PlayerDatabase
{
public void connectDatabase();
public void printAllPlayersInfo();
public void printSinglePlayerInfo();
public void printRankings()
public void closeDatabase();
}
```

Here, we have a class PlayerDatabase which is performing many different tasks like connecting to a database, printing the information of all the players, printing an information of a single player, printing all the players, printing all the rankings and finally closing all opened database connections. Now, such a class is not easy to create, maintain and update, as it is involved in doing so many different tasks i.e. a programming design to avoid.

Example of High Cohesion:

```
 Staff
----------------------------
| -salary              |
| -emailAddr           |
----------------------------
| setSalary(newSalary)    |
| getSalary()          |
| setEmailAddr(newEmail)  |
| getEmailAddr()       |
----------------------------
```

A good application design is creating an application with high cohesive classes, which are targeted towards a specific specialized task and such class is not only easy to create, but also easy to maintain and update.

```
class PlayerDatabase
{
ConnectDatabase connectD= new connectDatabase();
PrintAllPlayersInfo allPlayer= new PrintAllPlayersInfo();
PrintRankings rankings = new PrintRankings();
CloseDatabase closeD= new CloseDatabase();
PrintSinglePlayerInfo singlePlayer = PrintSinglePlayerInfo();
}
class ConnectDatabase
{    //connecting to database.    }
class CloseDatabase
{   //closing the database connection.  }

class PrintRankings
{   //printing the players current rankings.   }

class PrintAllPlayersInfo
{ //printing all the players information.   }

class PrintSinglePlayerInfo
{   //printing a single player information.  }
```

here we have created several different classes, each class performing a specific specialized task, leading to an easy creation, maintenance and modification of these classes. Classes created by following this programming design are said to performing a cohesive role and are high cohesion classes, which is an appropriate programming design while creating an application.

Coupling:

As for coupling, it refers to how related or dependent two classes/modules are toward each other. For low coupled classes, changing something major in one class should not affect the other. High coupling would make it difficult to change and maintain your code; since classes are closely knit together, making a change could require an entire system revamp.

Coupling refers to the extent to which a class knows about the other class.

Types of coupling -

1. **Tight Coupling(bad programming design) :**
   Tight coupling means the two classes often change together. In other words, if B knows more than it should about the way in which A was implemented, then A and B are tightly coupled.

If class A apart from interacting class B by means of its interface also interacts through the non-interface stuff of class B then they are said to be tightly coupled.

If a class A has some public data members and another class B is accessing these data members directly using the dot operator (which is possible because data members were declared public), the two classes are said to be tightly coupled.

Such tight coupling between two classes leads to the bad designing. You would ask - Why? Well, here's the explanation.

Let's say, the **class A** has a **String data member, name**, which is declared **public** and the class also has **getter and setter methods** that have implemented some checks to make sure -

- A valid access of the data member, name i.e. it is only accessed when its value is **not null**, and
- A valid setting of the data member, name i.e. it **cannot be set to a null** value.

But these checks implemented in the methods of class A to ensure a valid access and a valid setting of its data member, name, are bypassed by its direct access by class B i.e. tight coupling between two classes.

```
class A
{
public String name; //public data member of A class

public String getName()
{
if(name!=null)                    //Checking a valid access to "name"
        return name;
else
        return"not initiaized";
}

public void setName(String s)
{
if (s==null)    //Checking a valid setting to "name"
{
        System.out.println("You can't initialized name to a null");
}
}
}
```

```
class B
{
public static void main(String... ar)
{
A ob= new A();

//Directly setting the value of data member "name" of class A, due to tight coupling between the two classes
ob.name=null;

//Direct access of data member "name" of class A, due to tight coupling between two classes
System.out.println("Name is " + ob.name);
}
}
```

Program Analysis

- Class A has an instance variable, **name**, which is declared **public**.
- Class A has two **public getter and setter methods** which check for a valid access and a valid setting of data member - name.
- Class B creates an object of A class and sets the value of its data member, name, to null and accesses it value directly by the dot operator, because it was declared public.
- Hence, the checks implemented in getName() and setName() methods of class Names, to access and set the data member's value of class A are never called and are rather, bypassed. It shows class A is tighly coupled to class B, which is a bad design, compromising the data security checks.

2. **Loose Coupling(good programming design) :**

loose coupling means they are mostly independent. If the only knowledge that class B has about class A, is what class A has exposed through its interface, then class A and class B are said to be loosely coupled.

A good application designing is creating an application with loosely coupled classes by following proper encapsulation, i.e. by declaring data members of a class with the **private access** modifier, which disallows other classes to directly access these data members, **forcing them to call public getter, setter methods to access these private data members**. Let's understand this by an example –

```
class A
{
private String name;     //data member "name" is declared private to implement loose coupling.
public String getName()
{
if(name!=null)                  //Checking a valid access to name
        return name;
else
        return"not initiaized";
}
public void setName(String s)
{
if (s==null)    //Checking a valid setting to name
{
        System.out.println("You can't initialize name to a null");
}
}
}
```

```
class B
{
public static void main(String... ar)
{
A ob= new A();

//Calling setter method, as the direct access of "name" is not possible i.e. loose coupling between classes
ob.setName(null);

//Calling getter method, as the direct access of "name" is not possible i.e. loose coupling
System.out.println("Name is " + ob.getName());
}
}
```

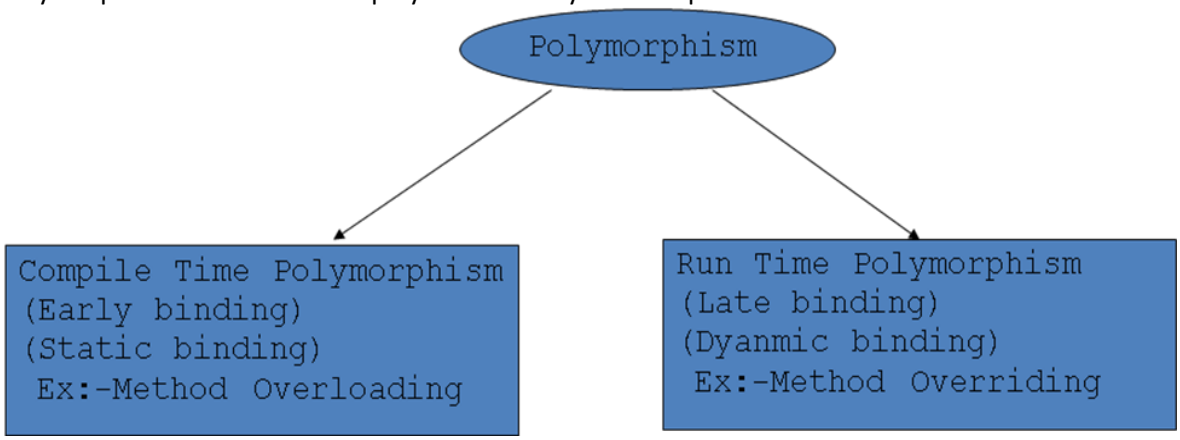
Program Analysis

- Class A has an instance variable, name, that is declared private.
- Class A has two public getter and setter methods methods which check for a valid access and a valid setting of data member, name.
- Class B creates an object of class A, calls the getName() and setName() methods and their implemented checks are properly executed before the value of instance member, name, is accessed or set. It shows class A is loosely coupled to class B, which is a good programming design.

10. PolyMorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

- One thing can exhibits more than one form called polymorphism.
- The ability to appear in more forms.
- Polymorphism is a Greek word poly means many and morphism means forms.



Binding:
The process of associating a method definition with a method invocation is called binding.

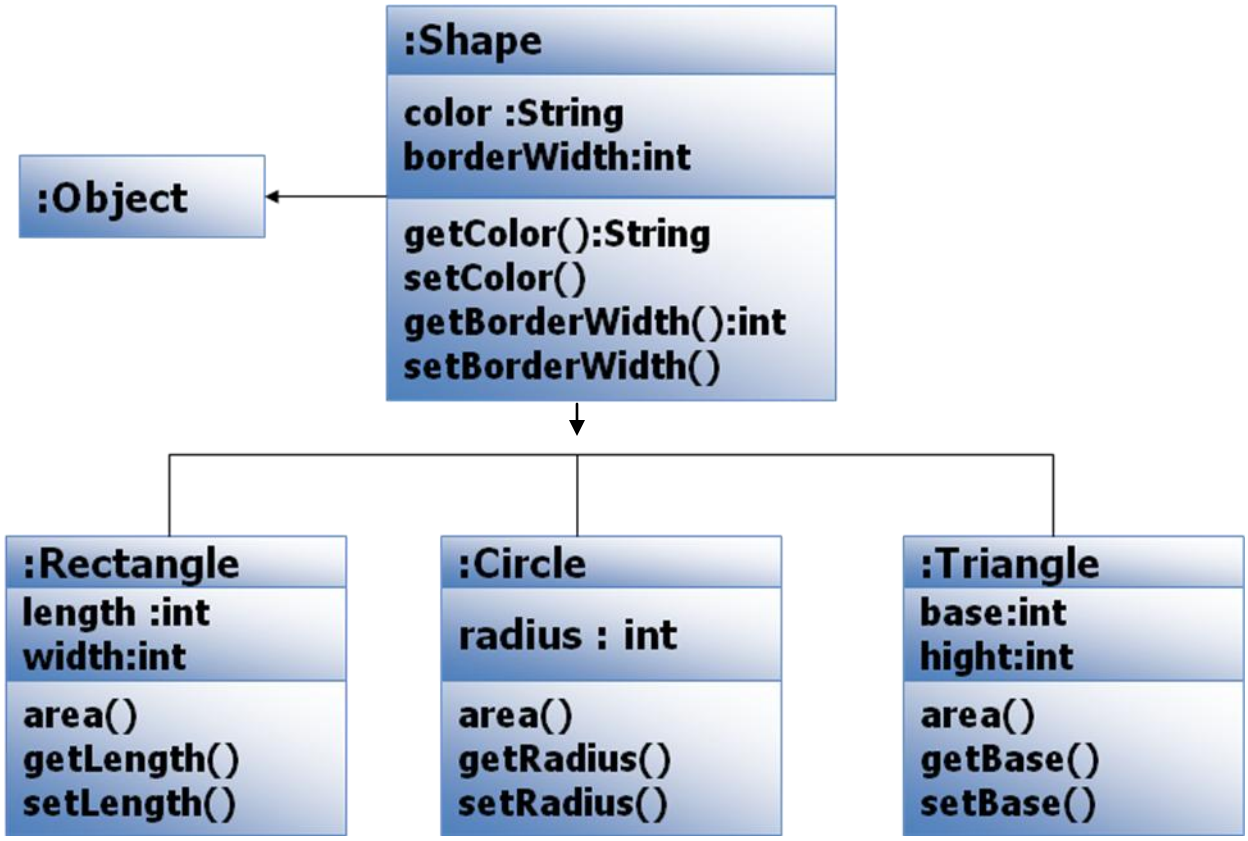
If the method definition is associated with its invocation when the code is compiled, that is called early binding .

If the method definition is associated with its invocation when the method is invoked (at run time), that is called late binding or dynamic binding.

Uses of PolyMorphism:

- Three Common Uses
  - Using Polymorphism in Arrays
  - Using Polymorphism for Method Arguments
  - Using Polymorphism for Method Return Type
- Through Interfaces
- Method Overriding
- Method Overloading

Lets Have a example of each above uses:



Using Polymorphism in Arrays:

Shape s[] = new Shape[3];
s[0] = new Rectangle()
s[1] = new Circle()
s[2] = new Triangle()

Using Polymorphism for Method Arguments:

```
public static void main(String[] args) {
Shape[] s = new Shape[3];
s[0] = new Rectangle();
s[1] = new Circle();
s[2] = new Triangle();
double totalArea = calcArea(s);
System.out.println(totalArea);
```

```
}
public static double calcArea(Shape[] s) {
double totalArea = 0;
for(int i =0;i<s.length; i++){
        totalArea += s[i].area();
}
return totalArea;
}
```

Using Polymorphism for Method Return Type:

```
public static Shape getShape(int i) {
if (i == 1) return new Rectangle();
if (i == 2) return new Circle();
if (i == 3) return new Triangle();
}
```

Using Polymorphism for Method Return Type:

```
public static Shape getShape(int i) {
if (i == 1) return new Rectangle();
if (i == 2) return new Circle();
if (i == 3) return new Triangle();
}
```