

# **Module1: OOP**

## **Software Development**

**Fall 2023**

**Dr. William Greg Johnson**  
**Department of Computer Science**  
**Georgia State University**

# Module1: Review of OOP with Java

- Fundamentals

- Inheritance
- Polymorphism
- Data abstraction/encapsulation

- Basic understanding

- The approach to program (code) organization and development
- OOP programs use best of structured programming features and new concepts
- Allows a software problem to be decomposed into numerous entities, e.g., classes, then build methods and data into these

- S.O.L.I.D. (mnemonic for 5 design principles in OOP)<sup>2</sup>

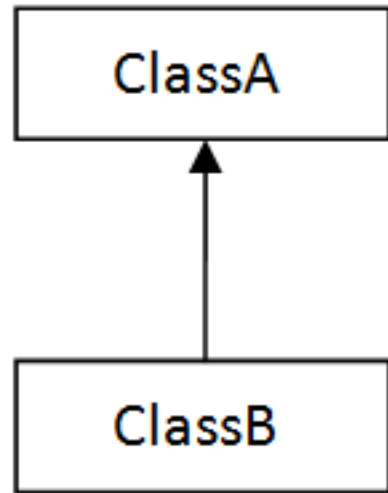
1. Single responsibility
2. Open closed
3. Liskov substitution
4. Interface segregation
5. Dependency inversion

# Inheritance

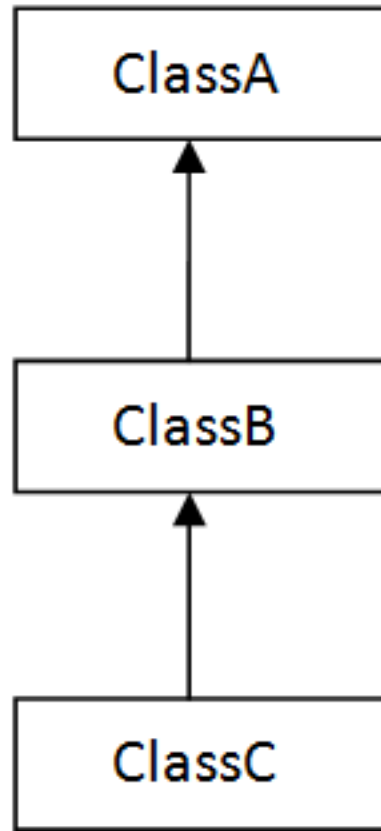
## Important features:

- Hierarchical organization
- Code reusability
- Method overriding and overloading (unless 'static')
- Enables both polymorphism and data encapsulation

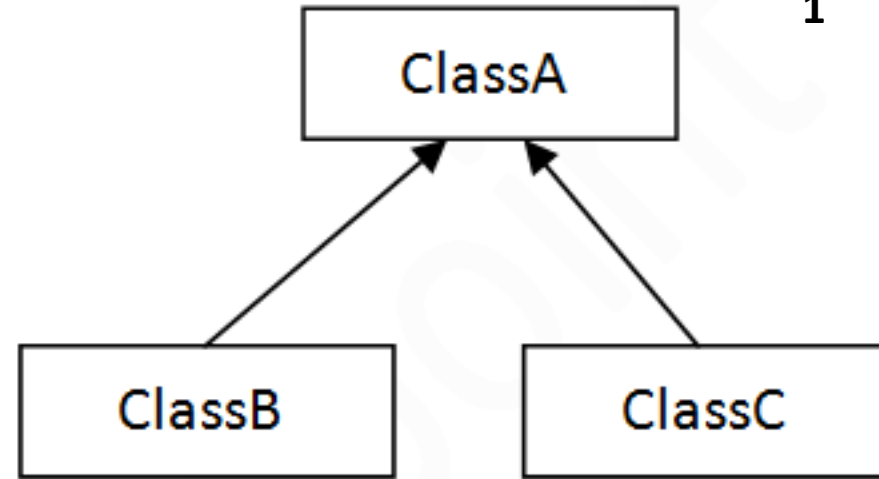
# Inheritance: Models for Java



1) Single



2) Multilevel



3) Hierarchical

1

# Inheritance

## Hierarchical organization

- Natural tree structure for class code (based on relationships and similar tasks)
- Parent class has more general methods and attributes
- Child classes have more that leverage the parent's code and can override if needed
- Relationship enables strong extensibility

## Code reusability

- Reduces time to code commonalities (avoids duplicating)
- Refactoring is easier and more effective (does not break existing use of parent class)
- No need to change child classes, but need to recompile ☹️

# Inheritance

- **Method overriding**
  - Child class can have a different implementation of code (methods)
  - Allows specialized behavior in child with option to use parent's method.
- **Enables both polymorphism and encapsulation**
  - Makes things transparent using child code when instantiation is parent code
  - Defines only basic functionality and forces subclasses to implement (abstract) remaining details. This promotes flexible hierarchies.

# Polymorphism

## Important features:

- Compile time vs runtime
- Method overriding and overloading
- Interfaces

# Polymorphism

3

## OVERRIDING

Implements at runtime

The method call is determined at runtime based on the object type

Occurs automatically with inherited methods

Have the same signature (name and method arguments)

On error, the effect will be visible at runtime

## OVERLOADING

Implements at compile time

The method call is determined at compile time

Occurs between the methods in the same class

Have the same name, but the parameters are different

On error, it can be caught at compile time



# Polymorphism

3

Processor.java

```
1 package com.journaldev.examples;
2
3 import java.util.Arrays;
4
5 public class Processor {
6
7     public void process(int i, int j) {
8         System.out.printf("Processing two integers:%d, %d", i, j);
9     }
10
11     public void process(int[] ints) {
12         System.out.println("Adding integer array:" + Arrays.toString(ints));
13     }
14
15     public void process(Object[] objs) {
16         System.out.println("Adding integer array:" + Arrays.toString(objs));
17     }
18 }
19
20 class MathProcessor extends Processor {
21
22     @Override
23     public void process(int i, int j) {
24         System.out.println("Sum of integers is " + (i + j));
25     }
26
27     @Override
28     public void process(int[] ints) {
29         int sum = 0;
30         for (int i : ints) {
31             sum += i;
32         }
33         System.out.println("Sum of integer array elements is " + sum);
34     }
35 }
36 }
```

Overloading: Same Method Name but different parameters in the same class

Overriding: Same Method Signature in both superclass and child class

# Polymorphism

```
interface parentBase {  
    void draw();  
}  
=====
```

```
class childBase implements parentBase  
{  
    public void draw() {  
        System.out.println("childbase");  
    }  
}  
=====
```

...

```
childBase c = new childBase();  
c.draw();
```

# Data Encapsulation has features:

- Data hiding keeps details of a class where a user does not need to know how it works in order to use it. This makes the code more readable and maintainable.
- Data abstraction and reusability allows you to abstract away the implementation class details and focus on the essential features. This makes the code more reusable and easier to understand.
- Data protection guards the data of a class from unauthorized access. This can be achieved by declaring the data members of the class as private. Only the methods of the class can access them.
- Maintenance makes code easier to keep running by isolating the implementation details of a class. This means that you can change the implementation of the class without affecting the code that uses it.

# S.O.L.I.D.

- S.O.L.I.D. (mnemonic for 5 design principles in OOP) <sup>2</sup>
  1. Single responsibility
  2. Open closed
  3. Liskov substitution
  4. Interface segregation
  5. Dependency inversion

# Single Responsibility

Every class should have a single job (responsibility) or single purpose <sup>2</sup>

1. Front-end designers create user interfaces
2. Testers do the functional and non-functional testing
3. Backend developer does server side, or API implementation

Every person (type) has a single job

# Single Responsibility

Motivations-

- Maintainability
- Testability
- Flexibility and Extensibility
- Parallel Development
- Loose Coupling

How?

- Single File - Single Class
- The class/function itself should only perform one task

# Single Responsibility

Every class should have a single job (responsibility) or single purpose <sup>2</sup>

1. Front-end designers create user interfaces
2. Testers do the functional and non-functional testing
3. Backend developer does server side, or API implementation

Every person (type) has a single job.

# Single Responsibility

Problematic code:

```
1 public class RegisterService
2 {
3     public RegisterService(String username)
4     {
5         if (username.ToLowerCase().equals("admin")){
6             throw new InvalidOperationException();
7         }
8         SqlConnection conn = new SqlConnection();
9         conn.Open();
10        SqlCommand c = new SqlCommand("INSERT INTO db.USERS (Username) '"+username+"'");
11
12        SmtpClient s_Client = new SmtpClient("smtp.host.com");
13        s_Client.send(new MailMessage());
14    }
15 }
16 }
```

RegisterService violates SRP with different things

- 1.Connect to the database and Register a user
- 2.Send an email



# Single Responsibility

Solution code:

```
1 ▼ public class RegisterService{  
2 ▼     public RegisterService(String username){  
3 ▼         if (username.toLowerCase().equals("admin")){  
4             throw new InvalidOperationException();  
5 ▲         }  
6         _userRepository.Insert(username);  
7         _emailService.Send(...);  
8 ▲     }  
9 ▲ }
```

It needs to split the class into 3 specific classes that each have a single job –

1. \_userRepository
2. \_emailService

# Open Closed

2

Classes, modules, functions, etc. should be open for extension, but closed for modification which means you should be able to extend a class behavior, without modifying it.

1. Developer **A** needs to release an update for a library or framework. (Open for change)
2. Developer **B** wants some modification or add some feature on that.
3. Then developer **B** is allowed to extend the existing class created by developer **A** which is extension.
4. Developer **B** is not supposed to modify the class directly. (Closed for change)

# Open Closed

## Motivations-

- Maintain stability of the existing codes
- Reduce the cost of a business change requirement
- Reduce testing of existing code
- Write new functionality without altering existing codes
- Loose coupling

## How?

- Inheritance
- Polymorphism
- Abstraction

# Open Closed

Problematic code:

```
1
2 ▼ public double Area(object[] shapes){
3     double area=0.0;
4 ▼   for( object[] s : shapes){
5 ▼       if(s.equals(Rectangle)){
6           Rectangle rect= (Rectangle)s;
7           area+= rect.width*rect.height;
8 ▲       }
9 ▼       else {
10          Circle circ= (Circle)s;
11          area+= circ.radius*circ.radius*Math.PI;
12 ▲       }
13 ▲   }
14     return area;
15 ▲ }
16
```

```
1 ▼ public class AreaCaclulator {
2 ▼     public double Area(Rectangle[] shapes){
3         double area=0.0;
4 ▼     for( object[] s : shapes){
5         area+= s.width*s.height;
6 ▲     }
7         return area;
8 ▲     }
9 ▲ }
```

AreaCalculator does not follow Open Closed -

1. Only handles rectangle and circle
2. Not able to calculate the 'area' of other shapes

# Open Closed

Solution code:

```
1 ▼ public class Rectangle extends Shape {  
2     private double width, height;  
3 ▼ public Rectangle(double _width, double _height){  
4         width=_width;  
5         height=_height;  
6 ▲ } |  
7 ▼ public double Area(){  
8     return width*height;  
9 ▲ }  
10 ▲ }
```

```
1 ▼ public abstract class Shape {  
2     public abstract double Area();  
3 ▲ }  
4
```

```
1 ▼ public class Circle extends Shape {  
2     private double radius, height;  
3 ▼ public Circle(double _radius){  
4         radius=_radius;  
5 ▲ }  
6 ▼ public double Area(){  
7     return radius*radius*Math.PI;  
8 ▲ }  
9 ▲ }
```

Using an abstract class – Shape, any area can be managed, regardless of the object.

# Liskov Substitution

“Derived or child classes must be substitutable for their base or parent classes” which means any class that is the child should be usable in place of its parent without any unexpected behavior. <sup>2</sup>

Motivations-

1. If **S** is a subtype of **T**, then objects of type **T** should be replaced with the objects of type **S**.
2. No type casting is required so that codebase is easier to maintain.
3. Without LS, every time we add or modify a subclass and consequently to be changed in multiple places. This is difficult and error- prone.

# Liskov Substitution

Problematic code:

```
1 ▼ public class Apple {  
2 ▼     public virtual String GetColor(){  
3         return "Red";  
4 ▲     }  
5 ▲ }  
6 =====  
7 ▼ public class Orange extends Apple {  
8     @Override  
9 ▼     public String GetColor(){  
10         return "Orange";  
11 ▲     }  
12 ▲ }
```

```
1 ▼ class Program {  
2 ▼     public static void main(String[] args) {  
3         Apple _apple = new Orange();  
4         System.out.println("apple color is "+_apple.GetColor());  
5 ▲     }  
6 ▲ }
```

Orange class is breaking base class behavior -

1. The output is – “apple color is orange”
2. Child class changed the behavior of base class which is opposite of LSP

# Liskov Substitution

Solution code:

```
1 ▼ public abstract class Fruit {
2     public abstract String GetColor();
3 ▲ }
4 =====
5 ▼ public class Apple extends Fruit {
6     @Override
7 ▼ public String GetColor(){
8     return "Apple color is red";
9 ▲ }
10 ▲ }
11 =====
12 ▼ public class Orange extends Fruit {
13     @Override
14 ▼ public String GetColor(){
15     return "Orange is orange";
16 ▲ }
17 ▲ }
```

```
1 ▼ class Program {
2 ▼     public static void main(String[] args) {
3         Fruit _fruit = new Orange();
4         System.out.println(_fruit.GetColor());
5         _fruit = new Apple();
6         System.out.println(_fruit.GetColor());
7 ▲     }
8 ▲ }
```

Based on generic base class Fruit

1. Both orange and apple implement it.
2. Parent class's behavior is not changed.



# Interface Segregation

“Do not force any client to implement an interface which is irrelevant to them” which means – “Many client-specific interfaces are better than one general-purpose interface.”

2

## Motivations-

1. First, no class should be forced to implement any method(s) of an interface they don't use.
2. Secondly, instead of creating overly large interfaces, create multiple smaller interfaces with the aim that the clients should only think about the methods that are of interest to them.
3. There is less code carried between classes. Less code means fewer bugs.

# Interface Segregation

Problematic code:

```
1 ▼ public interface IWorker {  
2     public String getID();  
3     public void setID(String _ID);  
4     public String getName();  
5     public void setName(String _name);  
6     public String getEmail();  
7     public void setEmail(String _email);  
8     public double getMonthSalary();  
9     public void setMonthSalary(double _salary);  
10    public double getMonthBenefit();  
11    public void setMonthBenefit(double _Mon);  
12    public double getHourRate();  
13    public void setHourRate(double _hr_Rate);  
14    public double getHoursMonthly();  
15    public void setHoursMonthly(double _Hours_Month);  
16    public double CalculatePay();  
17 ▲ }
```

This code violates ISP in the following ways –

1. Both the classes implemented same interface, IWorker, and inherited unnecessary methods.
2. SalaryWorker class does not need the hourly rate or HoursMonthly.
3. ContractWorker class does not need the MonthSalary or MonthBenefit.

# Interface Segregation

Problematic code:

```
1 ▼ public class SalaryWorker implements IWorker {
2     private String ID, Name, Email;
3     private double MonthSalary, MonthBenefit;
4     private double HrRate, HrMonthly;
5     public String getID(){ return ID; }
6     public void setID(String _ID){ }
7     public String getName(){ return Name; }
8     public void setName(String _name){ }
9     public String getEmail(){ return Email; }
10    public void setEmail(String _email){ }
11    public double getMonthSalary(){ return MonthSal; }
12    public void setMonthSalary(double _salary){ }
13    public double setMonthBenefit(double _Mon ){ }
14    public void setMonthBenefit(){ return MonthBenefit; }
15    public double getHourRate(){ return HrRate }
16    public void setHourRate(double _hr_Rate){ }
17    public double getHoursMonthly(){ return HrMonthly; }
18    public void setHoursMonthly(double _Hours_Month){ }
19 ▼    public double CalculatePay(){
20        return getMonthSalary()+getMonthBenefit();
21 ▲    }
22 ▲ }
```

```
1 ▼ public class ContractWorker implements IWorker {
2     private String ID, Name, Email;
3     private double MonthSalary, MonthBenefit;
4     private double HrRate, HrMonthly;
5     public String getID(){ return ID; }
6     public void setID(String _ID){ }
7     public String getName(){ return Name; }
8     public void setName(String _name){ }
9     public String getEmail(){ return Email; }
10    public void setEmail(String _email){ }
11    public double getMonthSalary(){ return MonthSal; }
12    public void setMonthSalary(double _salary){ }
13    public double setMonthBenefit(double _Mon ){ }
14    public void setMonthBenefit(){ return MonthBenefit; }
15    public double getHourRate(){ return HrRate }
16    public void setHourRate(double _hr_Rate){ }
17    public double getHoursMonthly(){ return HrMonthly; }
18    public void setHoursMonthly(double _Hours_Month){ }
19 ▼    public double CalculatePay(){
20        return getHourRate()*getHoursMonthly();
21 ▲    }
22 ▲ }
```

\* Identical and repeated code except for CalculatePay() computation.

# Interface Segregation

## Solution:

- It needs to split the general interface IWorker into one base interface, IBaseWorker, and two child interfaces IFullTimeWorkerSalary and IContractWorkerSalary.
- The general interface - IBaseWorker contains methods that all workers share.
- The child interfaces split up methods by worker type, FullTime with a salary or Contract that gets paid hourly.



# Interface Segregation

Solution:

Interfaces can inherit other interfaces.

```
1 ▼ public interface IBaseWorker {  
2     public String getID();  
3     public void setID( String _ID );  
4     public String getName();  
5     public void setName( String _Name );  
6     public String getEmail();  
7     public void setEmail( String _Email );  
8 ▲ }
```

```
1 ▼ public interface IFullTimeWorkerSalary extends IBaseWorker {  
2     public double getMonthSalary();  
3     public void setMonthSalary( double _SalRate);  
4     public double getMonthBenefit();  
5     public void setMonthBenefits( double _Benefits);  
6     public double CalculateFullTimePay();  
7 ▲ }
```

```
1 ▼ public interface IContractWorkerSalary extends IBaseWorker {  
2     public double getHourlyRate();  
3     public void setHourlyRate( double _HrlyRate);  
4     public double getHoursMonth();  
5     public void setHoursMonth( double _HrsMonth);  
6     public double CalculateMonthlyContractPay();  
7 ▲ }
```

# Interface Segregation

Solution:

```
1 ▼ public class FullTimeEmployee implements IFullTimeWorkerSalary {
2     private String ID, Name, Email;
3     private double MonthSalary, MonthBenefit;
4     public String getID(){ return ID; }
5     public void setID(String _ID){ }
6     public String getName(){ return Name; }
7     public void setName(String _name){ }
8     public String getEmail(){ return Email; }
9     public void setEmail(String _email){ }
10    public double getMonthSalary(){ return MonthSal; }
11    public void setMonthSalary(double _salary){ }
12    public double setMonthBenefit(double _Mon ){ }
13    public void setMonthBenefit(){ return MonthBenefit; }
14 ▼    public double CalculateFullTimePay(){
15        return getMonthSalary()+getMonthBenefit();
16 ▲    }
17 ▲ }
```

To solve the problems:

1. It needs to split the general interface IWorker into one base interface, IBaseWorker, and two child interfaces IFullTimeWorkerSalary and IContractWorkerSalary.
2. The general interface - IBaseWorker contains methods that all workers share.
3. The child interfaces split up methods by worker type, FullTime with a salary or Contract that gets paid hourly.

# Dependency Inversion

The rule states that -

2

One should depend upon abstractions, but not concretizations.

Motivations-

1. High-level modules should not depend on low-level modules. Instead, both should depend on abstractions (interfaces)
2. Abstractions should not depend on details. Details (like concrete implementations) should depend on abstractions.
3. If dependencies are minimized, changes will be more localized and require less work to find all affected components.

# Dependency Inversion

“Dependency Inversion enables the construction of loosely coupled components—leading to simpler testing and replacement of modules without causing disruptions to the entire system. Furthermore, this approach fosters the utilization of S.O.L.I.D. principles, ultimately leading to a more refined and sustainable codebase.”

4




# Dependency Inversion

## Problematic code:

```
1 ▼ public class Book {
2 ▼     void seeReviews() {
3         ...
4 ▲     }
5 ▼     void readSample() {
6         ...
7 ▲     }
8 ▲ }
9
10 =====
11
12 ▼ public class Shelf {
13     Book book;
14 ▼     void addBook(Book book) {
15         ...
16 ▲     }
17 ▼     void customizeShelf() {
18         ...
19 ▲     }
20 ▲ }
```

Everything looks fine, but as the high-level Shelf class depends on the low-level Book, the above code violates the Dependency Inversion Principle. This becomes clear when the store asks us to enable customers to add DVDs to their shelves, too. To fulfil this, we create a new DVD class. Now, we should modify the Shelf class so that it can accept DVD, and this will break the Open/Closed Principle. 5

```
1 ▼ public class DVD {
2 ▼     void seeReviews() {
3         ...
4 ▲     }
5 ▼     void watchSample() {
6         ...
7 ▲     }
8 ▲ }
```



```
1 ▼ public class NewShelf {
2     Book book;
3     DVD dvd;
4 ▼     void addDVD(DVD dvd) {
5         ...
6 ▲     }
7 ▼     void addBook(Book book) {
8         ...
9 ▲     }
10 ▼     void customizeShelf() {
11         ...
12 ▲     }
13 ▲ }
```

# Dependency Inversion

Solution:

```
1 ▼ public interface Product {
2     void seeReviews();
3     void getSample();~
4 ▲ }
5 =====
6 ▼ public class Book implements Product {
7     @Override
8 ▼ public void seeReviews() {
9         ...
10 ▲ }
11     @Override
12 ▼ public void getSample() {
13         ...
14 ▲ }
15 ▲ }
16 =====
17 ▼ public class DVD implements Product {
18     @Override
19 ▼ public void seeReviews() {
20         ...
21 ▲ }
22     @Override
23 ▼ public void getSample() {
24         ...
25 ▲ }
26 ▲ }
```

```
1 ▼ public class Shelf {
2     Product product;
3 ▼ void addProduct(Product product) {
4         ...
5 ▲ }
6 ▼ void customizeShelf() {
7         ...
8 ▲ }
9 ▲ }
```

Shelf can reference the Product interface instead of its implementations (Book and DVD). The refactored code also allows us to later introduce new product types (for instance, Magazine) that customers can put on their shelves, too.

# Summary of OOP basics

The following are the basic concepts:

1. Objects.
2. Classes.
3. Abstraction.
4. Encapsulation.
5. Inheritance.
6. Polymorphism.
7. Interfaces.
8. Message Passing.

# Attributions

1. <https://www.slideshare.net/KunalYadav65140/javapptscompletechromepptx>
2. <https://www.slideshare.net/kumaresbbaruri/solid-principles>
3. <https://www.digitalocean.com/community/tutorials/overriding-vs-overloading-in-java>
4. <https://stackify.com/dependency-inversion-principle/>
5. <https://javatechonline.com/solid-principles-the-dependency-inversion-principle/>

# Resources

- <https://www.javatpoint.com/inheritance-in-java>
- <https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/>
- <https://dotnettutorials.net/course/solid-design-principles/> (.NET)
- <https://www.javatpoint.com/solid-principles-java>