S.O.L.I.D: The First 5 Principles of Object Oriented Design

Trịnh Tuấn Đạt

S.O.L.I.D stands for

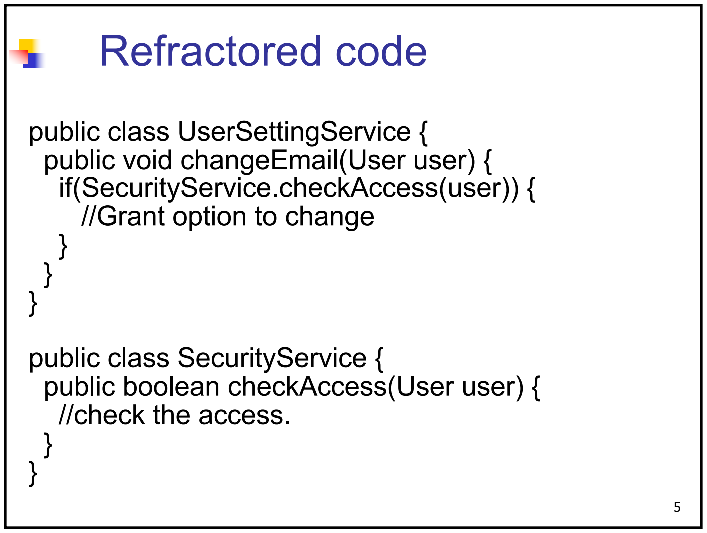
- S – Single-responsiblity principle
- O – Open-closed principle
- L – Liskov substitution principle
- I – Interface segregation principle
- D – Dependency Inversion Principle

1. Single-responsibility Principle

- A class should have one and only one reason to change, meaning that a class should have only one job

Example 1 – UserSettingService

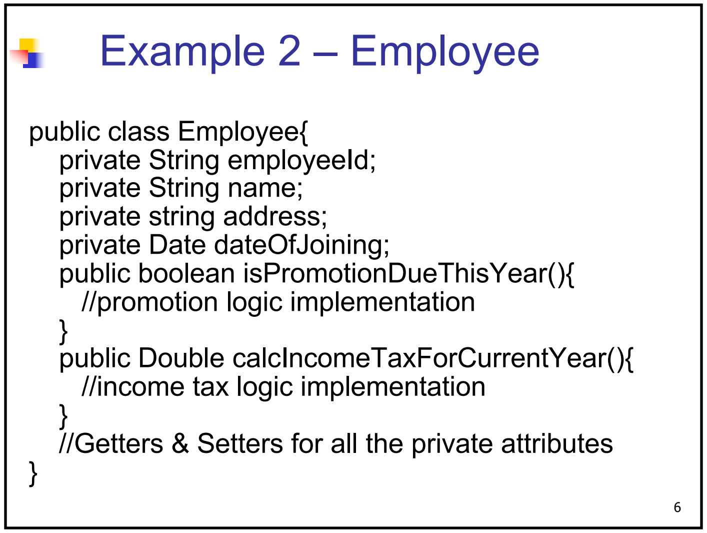
```
public class UserSettingService {
  public void changeEmail(User  user)  {
    if(checkAccess(user))  {
       //Grant option to change
    }
  }
  public boolean checkAccess(User  user) {
    //Verify if the user is valid.
  }
}
```



Refractored code

```
public class UserSettingService {
  public void changeEmail(User user) {
    if(SecurityService.checkAccess(user)) {
       //Grant option to change
    }
  }
}

public class SecurityService {
  public boolean checkAccess(User user) {
    //check the access.
  }
}
```

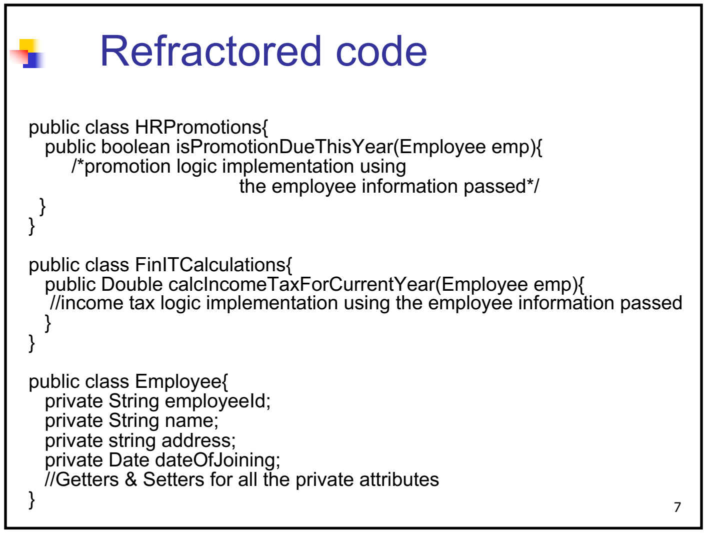
5



Example 2 – Employee

```
public class Employee{
    private String employeeId;
    private String name;
    private string address;
    private Date dateOfJoining;
    public boolean isPromotionDueThisYear(){
       //promotion logic implementation
    }
    public Double calcIncomeTaxForCurrentYear(){
       //income tax logic implementation
    }
    //Getters & Setters for all the private attributes
}
```

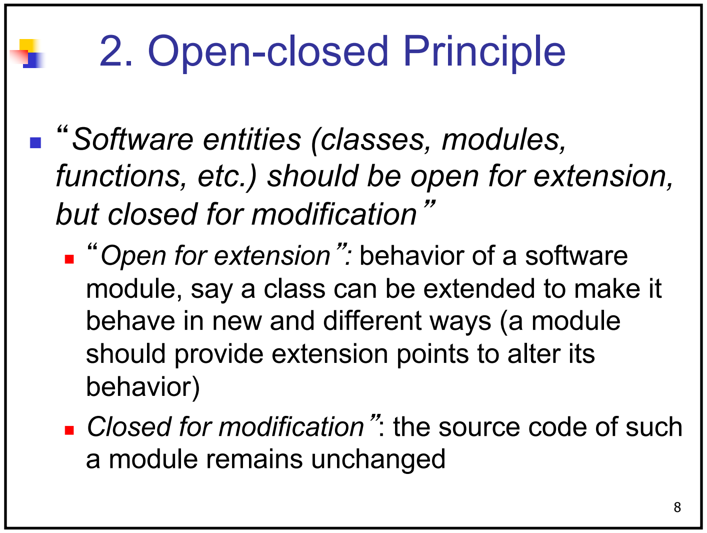
6



Refractored code

```
public class HRPromotions{
   public boolean isPromotionDueThisYear(Employee emp){
        /*promotion logic implementation using
                                  the employee information passed*/
  }
}

public class FinITCalculations{
   public Double calcIncomeTaxForCurrentYear(Employee emp){
    //income tax logic implementation using the employee information passed
   }
}

public class Employee{
   private String employeeId;
   private String name;
   private string address;
   private Date dateOfJoining;
   //Getters & Setters for all the private attributes
}
```

7



2. Open-closed Principle

- *"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"*
  - *"Open for extension":* behavior of a software module, say a class can be extended to make it behave in new and different ways (a module should provide extension points to alter its behavior)
  - *Closed for modification"*: the source code of such a module remains unchanged

8

Example – HealthInsuranceSurveyor

```
public class HealthInsuranceSurveyor{

    public boolean isValidClaim(){
        System.out.println(“Validating ...");
        /*Logic to validate health insurance claims*/
        return true;
    }
}
```

ClaimApprovalManager

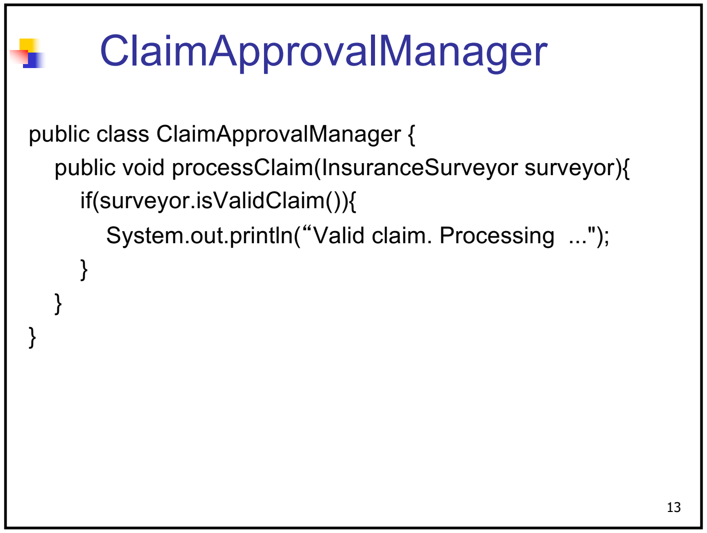
```
public class ClaimApprovalManager {

    public void processHealthClaim (HealthInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
          System.out.println(“Valid claim. Processing claim for approval....");
        }
    }
}
```

ClaimApprovalManager

```
public class ClaimApprovalManager {
    public void processHealthClaim (HealthInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println(“Valid claim. Processing ...");
        }
    }
    public void processVehicleClaim (VehicleInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println(“Valid claim. Processing ...");
        }
    }
}
```

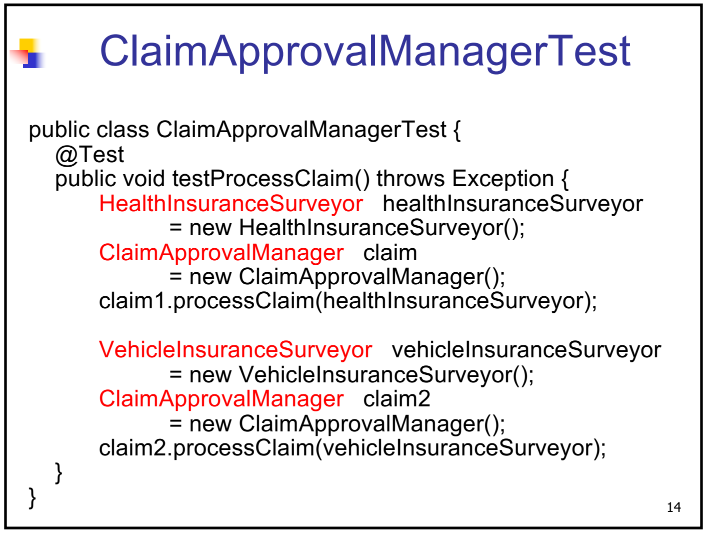
Refractored code

```
public abstract class InsuranceSurveyor {
    public abstract boolean isValidClaim();
}

public class HealthInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("HealthInsuranceSurveyor: Validating claim...");
        /*Logic to validate health insurance claims*/
        return true;
    }

}

public class VehicleInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
       System.out.println("VehicleInsuranceSurveyor: Validating claim...");
        /*Logic to validate vehicle insurance claims*/
        return true;
    }
}
```



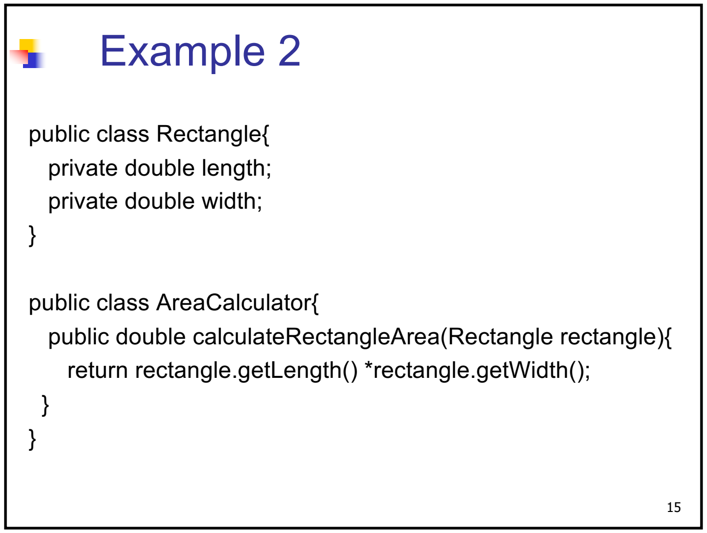
ClaimApprovalManager

```
public class ClaimApprovalManager {
    public void processClaim(InsuranceSurveyor surveyor){
        if(surveyor.isValidClaim()){
            System.out.println(“Valid claim. Processing  ...");
        }
    }
}
```



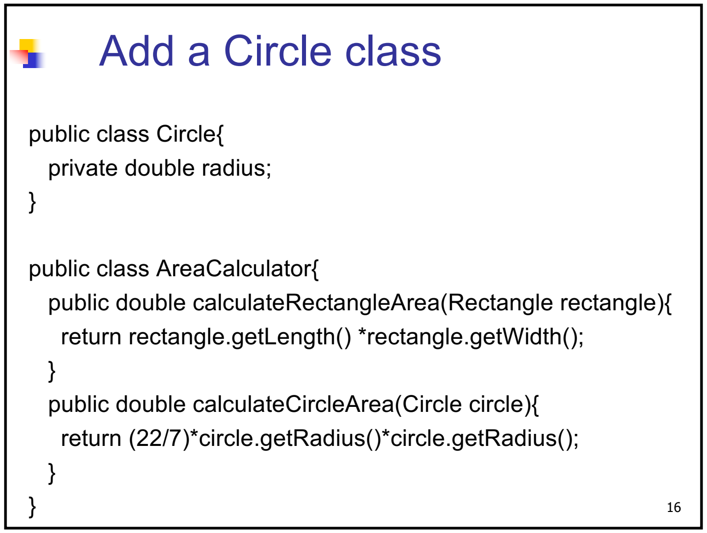
ClaimApprovalManagerTest

```
public class ClaimApprovalManagerTest {
    @Test
    public void testProcessClaim() throws Exception {
        HealthInsuranceSurveyor   healthInsuranceSurveyor
                = new HealthInsuranceSurveyor();
        ClaimApprovalManager   claim
                = new ClaimApprovalManager();
        claim1.processClaim(healthInsuranceSurveyor);

        VehicleInsuranceSurveyor   vehicleInsuranceSurveyor
                = new VehicleInsuranceSurveyor();
        ClaimApprovalManager   claim2
                = new ClaimApprovalManager();
        claim2.processClaim(vehicleInsuranceSurveyor);
    }
}
```



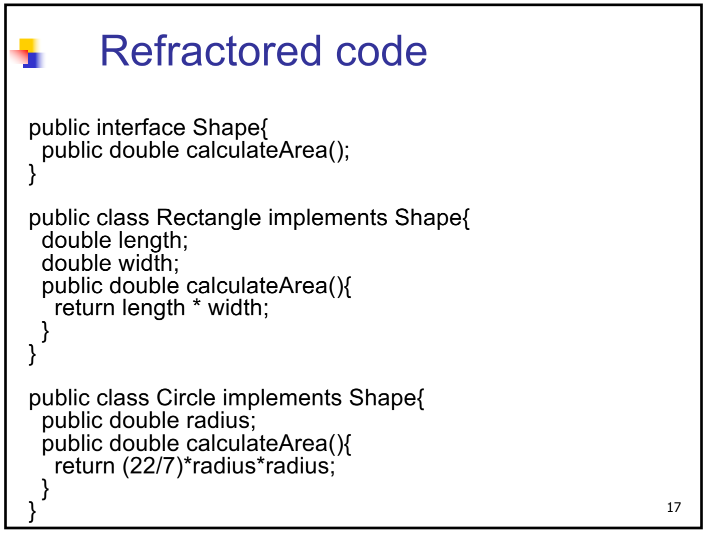
Example 2

```
public class Rectangle{
   private double length;
   private double width;
}

public class AreaCalculator{
   public double calculateRectangleArea(Rectangle rectangle){
      return rectangle.getLength() *rectangle.getWidth();
  }
}
```



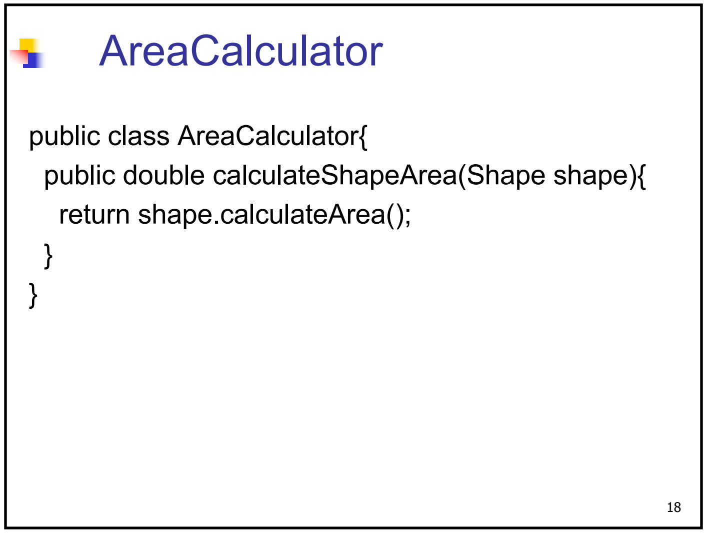
Add a Circle class

```
public class Circle{
   private double radius;
}

public class AreaCalculator{
   public double calculateRectangleArea(Rectangle rectangle){
     return rectangle.getLength() *rectangle.getWidth();
   }
   public double calculateCircleArea(Circle circle){
     return (22/7)*circle.getRadius()*circle.getRadius();
   }
}
```



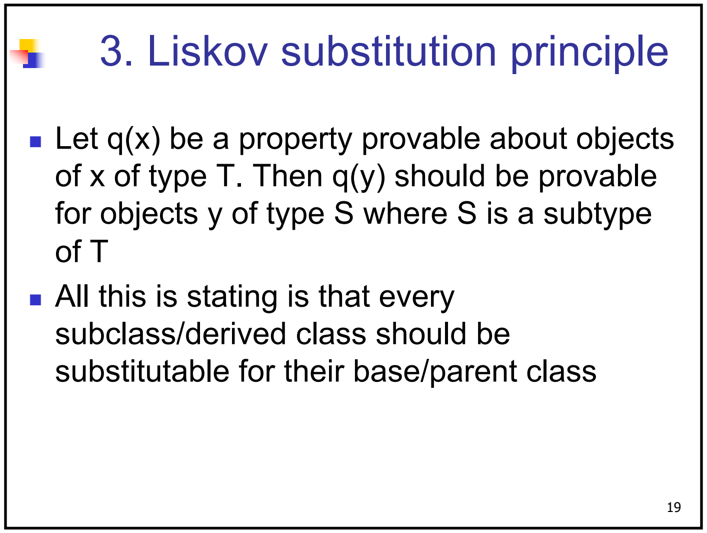
Refractored code

```
public interface Shape{
  public double calculateArea();
}

public class Rectangle implements Shape{
  double length;
  double width;
  public double calculateArea(){
    return length * width;
  }
}

public class Circle implements Shape{
  public double radius;
  public double calculateArea(){
    return (22/7)*radius*radius;
  }
}
```



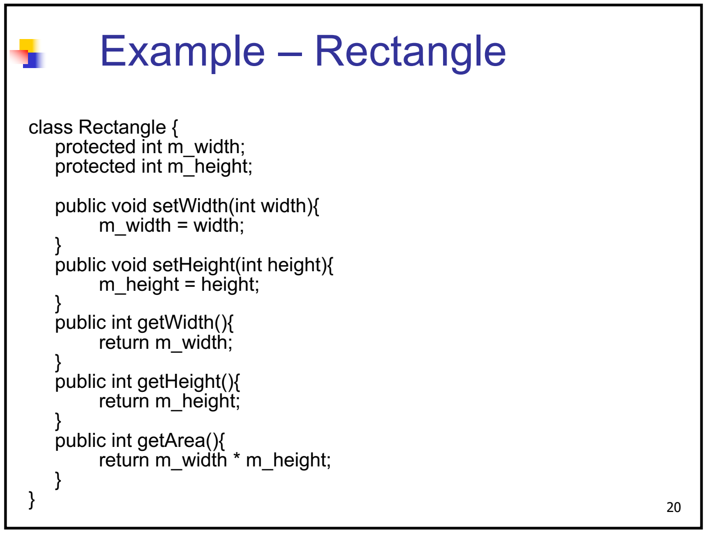
AreaCalculator

```
public class AreaCalculator{
  public double calculateShapeArea(Shape shape){
    return shape.calculateArea();
  }
}
```



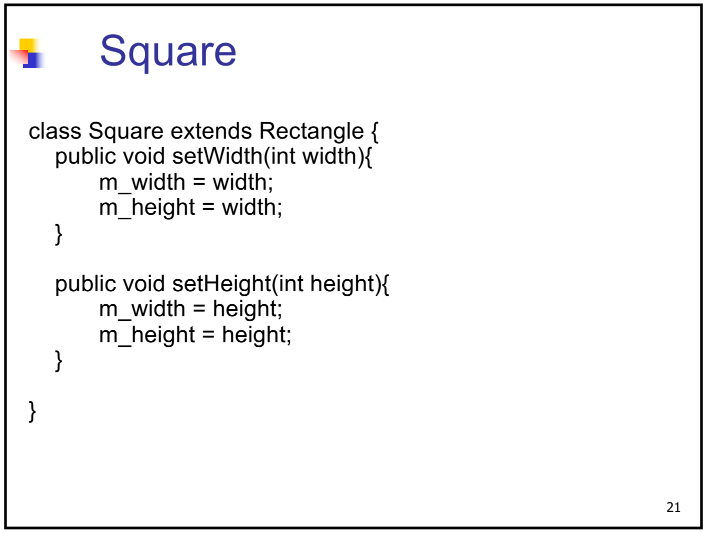
3. Liskov substitution principle

- Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T
- All this is stating is that every subclass/derived class should be substitutable for their base/parent class



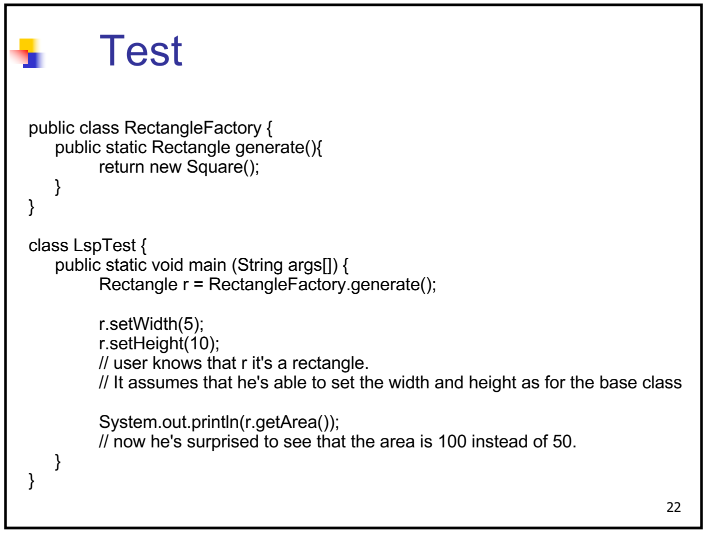
Example – Rectangle

```
class Rectangle {
    protected int m_width;
    protected int m_height;

    public void setWidth(int width){
          m_width = width;
    }
    public void setHeight(int height){
          m_height = height;
    }
    public int getWidth(){
          return m_width;
    }
    public int getHeight(){
          return m_height;
    }
    public int getArea(){
          return m_width * m_height;
    }
}
```



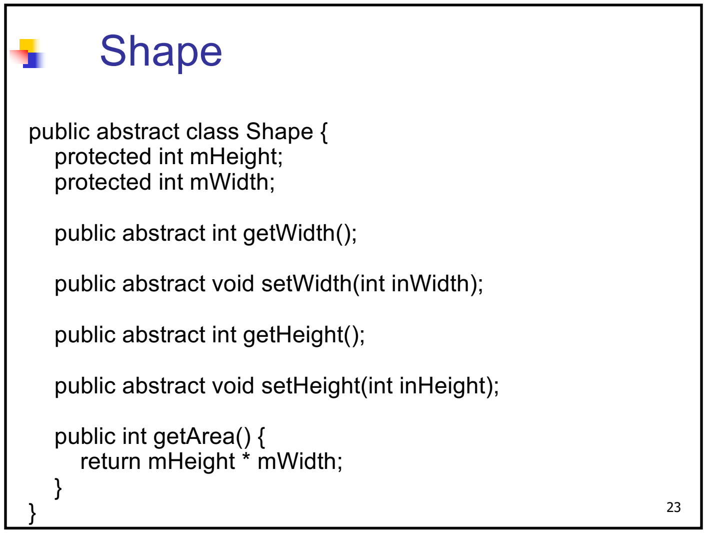
Square

```
class Square extends Rectangle {
    public void setWidth(int width){
          m_width = width;
          m_height = width;
    }

    public void setHeight(int height){
          m_width = height;
          m_height = height;
    }

}
```



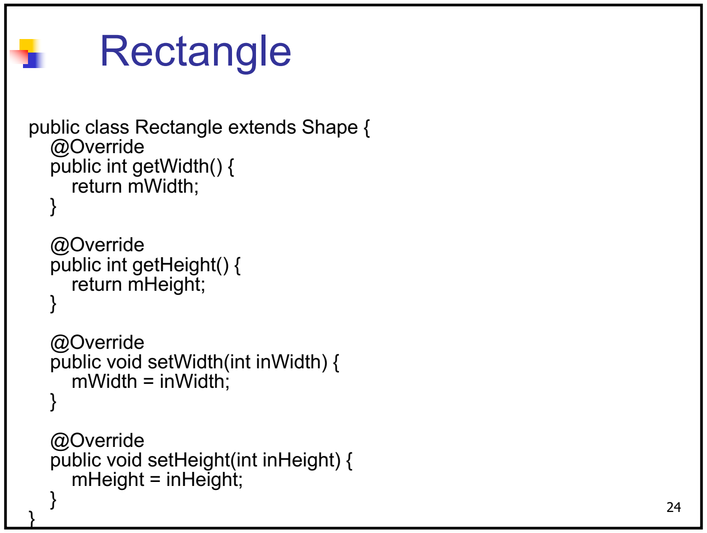
Test

```
public class RectangleFactory {
     public static Rectangle generate(){
            return new Square();
     }
}

class LspTest {
     public static void main (String args[]) {
            Rectangle r = RectangleFactory.generate();

            r.setWidth(5);
            r.setHeight(10);
            // user knows that r it's a rectangle.
            // It assumes that he's able to set the width and height as for the base class

            System.out.println(r.getArea());
            // now he's surprised to see that the area is 100 instead of 50.
     }
}
```



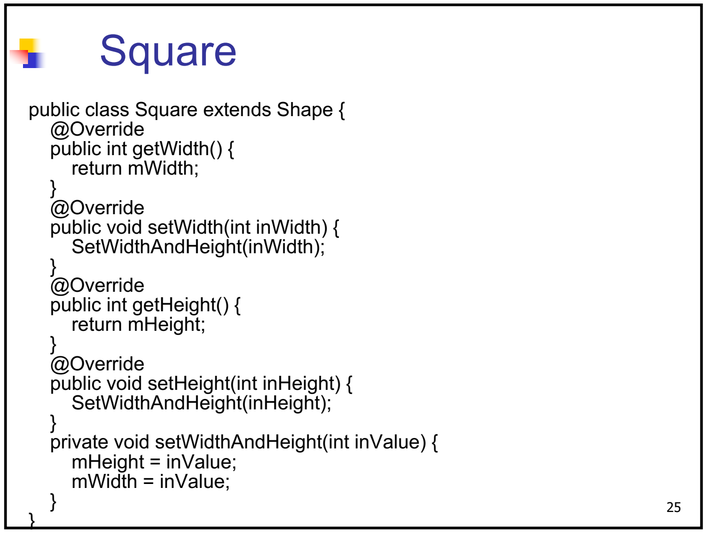
Shape

```
public abstract class Shape {
    protected int mHeight;
    protected int mWidth;

    public abstract int getWidth();

    public abstract void setWidth(int inWidth);

    public abstract int getHeight();

    public abstract void setHeight(int inHeight);

    public int getArea() {
        return mHeight * mWidth;
    }
}
```



Rectangle

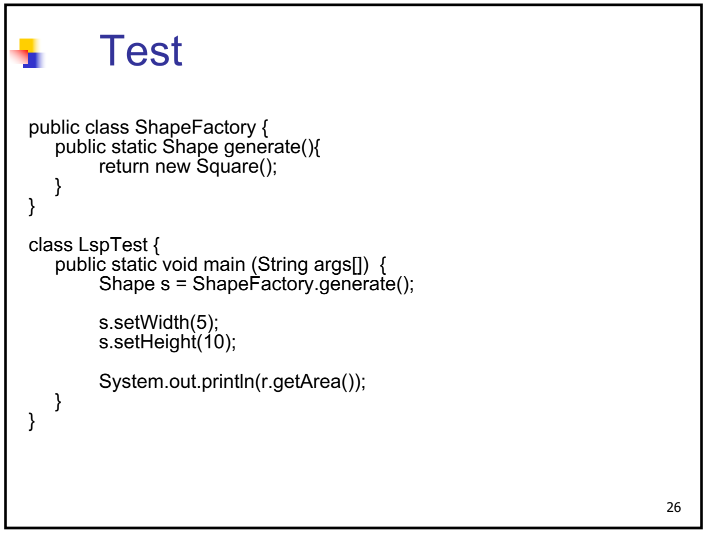
```
public class Rectangle extends Shape {
    @Override
    public int getWidth() {
       return mWidth;
    }

    @Override
    public int getHeight() {
       return mHeight;
    }

    @Override
    public void setWidth(int inWidth) {
       mWidth = inWidth;
    }

    @Override
    public void setHeight(int inHeight) {
       mHeight = inHeight;
    }
}
```



Square

```
public class Square extends Shape {
    @Override
    public int getWidth() {
        return mWidth;
    }
    @Override
    public void setWidth(int inWidth) {
        SetWidthAndHeight(inWidth);
    }
    @Override
    public int getHeight() {
        return mHeight;
    }
    @Override
    public void setHeight(int inHeight) {
        SetWidthAndHeight(inHeight);
    }
    private void setWidthAndHeight(int inValue) {
        mHeight = inValue;
        mWidth = inValue;
    }
}
```

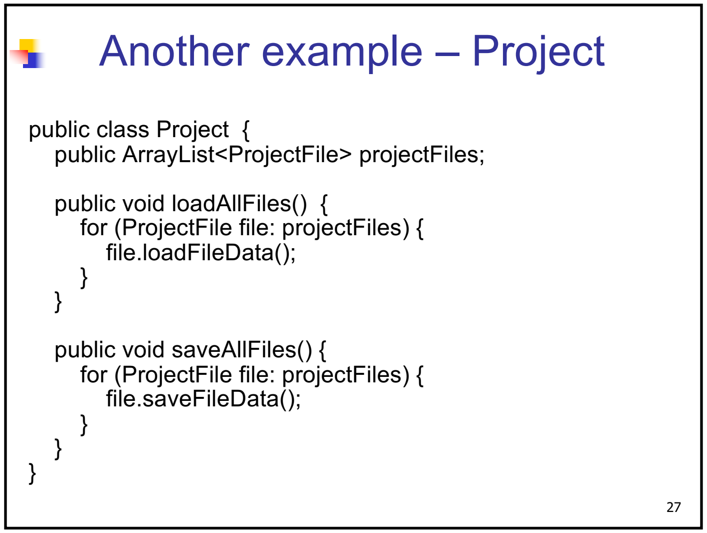
25



Test

```
public class ShapeFactory {
    public static Shape generate(){
            return new Square();
    }
}

class LspTest {
    public static void main (String args[])  {
            Shape s = ShapeFactory.generate();

            s.setWidth(5);
            s.setHeight(10);

            System.out.println(r.getArea());
    }
}
```

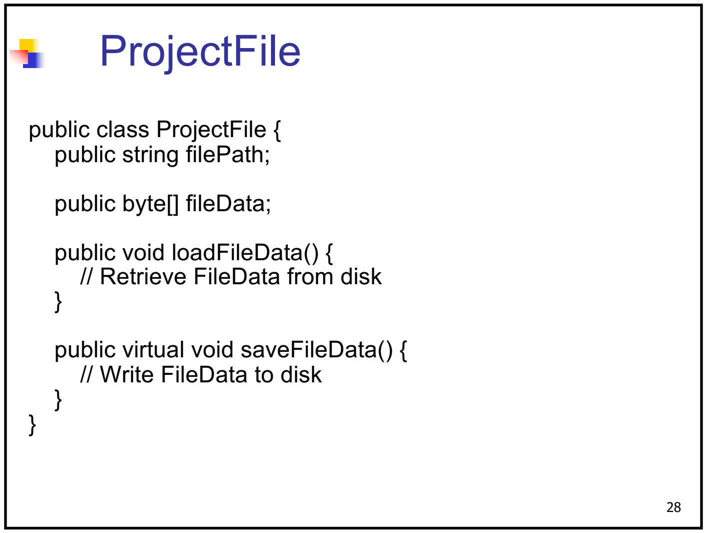
26



Another example – Project

```
public class Project  {
    public ArrayList<ProjectFile> projectFiles;

    public void loadAllFiles()  {
        for (ProjectFile file: projectFiles) {
            file.loadFileData();
        }
    }

    public void saveAllFiles() {
        for (ProjectFile file: projectFiles) {
            file.saveFileData();
        }
    }
}
```

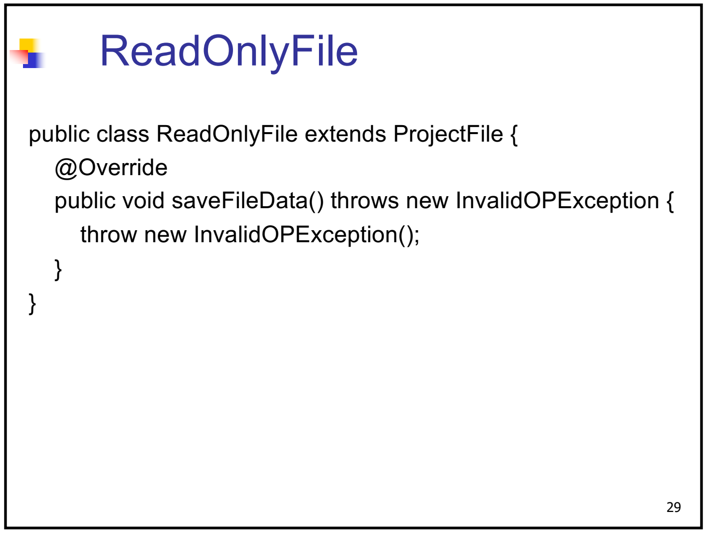
27



ProjectFile

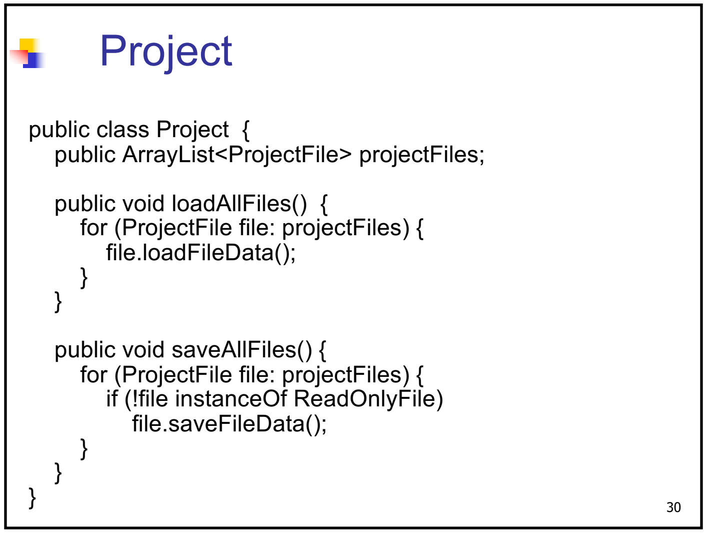
```
public class ProjectFile {
    public string filePath;

    public byte[] fileData;

    public void loadFileData() {
        // Retrieve FileData from disk
    }

    public virtual void saveFileData() {
        // Write FileData to disk
    }
}
```

28



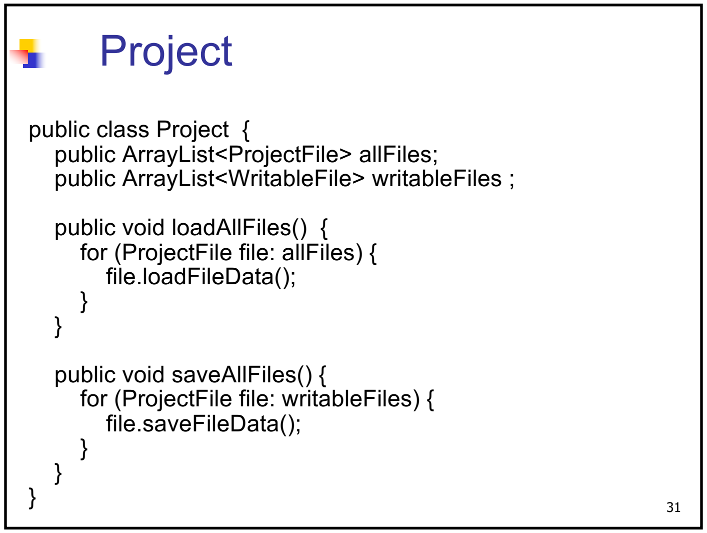
ReadOnlyFile

```
public class ReadOnlyFile extends ProjectFile {
   @Override
   public void saveFileData() throws new InvalidOPException {
      throw new InvalidOPException();
   }
}
```



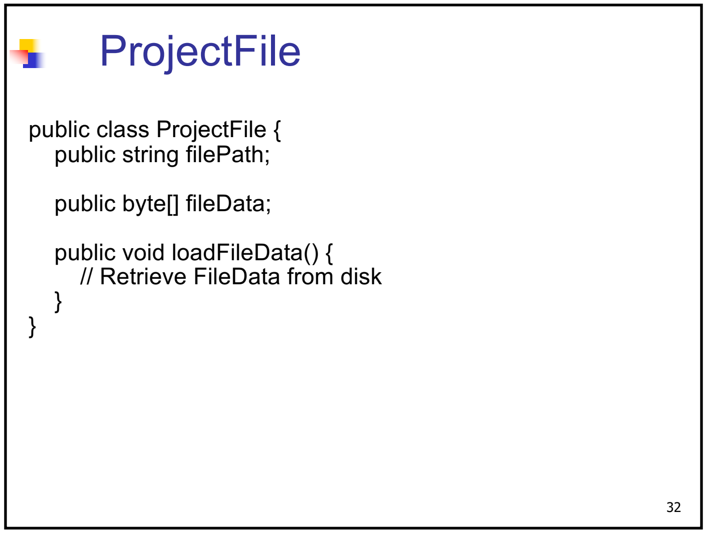
Project

```
public class Project  {
   public ArrayList<ProjectFile> projectFiles;

   public void loadAllFiles()  {
      for (ProjectFile file: projectFiles) {
         file.loadFileData();
      }
   }

   public void saveAllFiles() {
      for (ProjectFile file: projectFiles) {
         if (!file instanceOf ReadOnlyFile)
            file.saveFileData();
      }
   }
}
```



Project

```
public class Project  {
   public ArrayList<ProjectFile> allFiles;
   public ArrayList<WritableFile> writableFiles ;

   public void loadAllFiles()  {
      for (ProjectFile file: allFiles) {
         file.loadFileData();
      }
   }

   public void saveAllFiles() {
      for (ProjectFile file: writableFiles) {
         file.saveFileData();
      }
   }
}
```



ProjectFile

```
public class ProjectFile {
   public string filePath;

   public byte[] fileData;

   public void loadFileData() {
      // Retrieve FileData from disk
   }
}
```

WritableFile

```
public class WritableFile extends ProjectFile {
    public void saveFileData() {
        // Write FileData to disk
    }
}
```

Question

- Is method overriding always a violation of Liskov Substitution Principle?

Example

```
public class Report{
   private Foo foo;
   public String toString(){
        return "";
   }
}
```

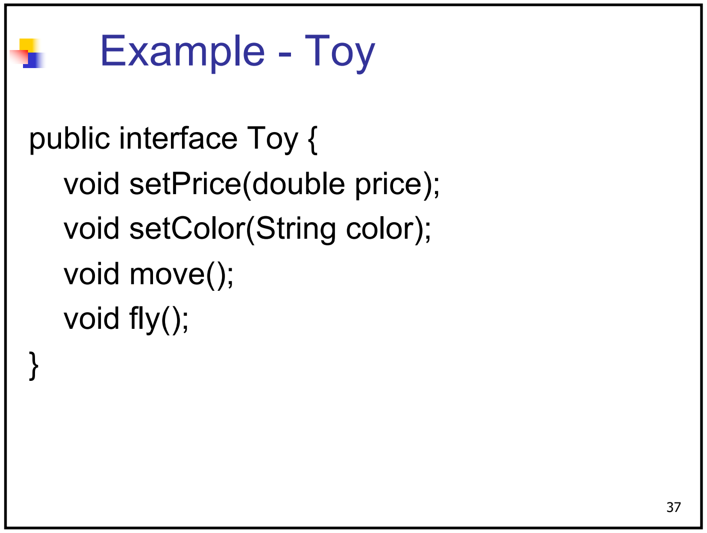
Three subclasses

1. HTMLReport
2. XMLReport
3. TextReport

- Contract:
  - toString() shall deliver a string with a textual representation of Foo in a certain text format (and the empty string if the format is not defined so far).
  - toString() shall not mutate the Report object
  - toString() shall never throw an Exception

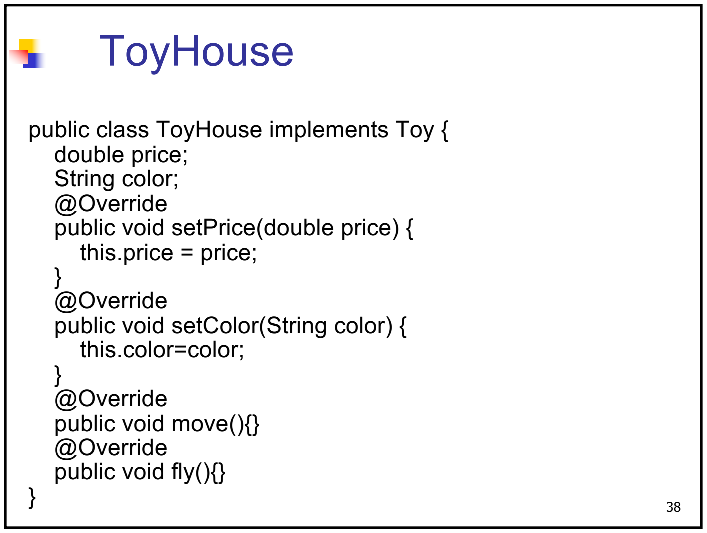
4. Interface Segregation Principle

- Clients should not be forced to depend on methods that they do not use. In other words, interface should not be bloated with methods that implementing classes don’t require
- “Fat interface” should be segregated into smaller and highly cohesive interfaces, known as “role interfaces”



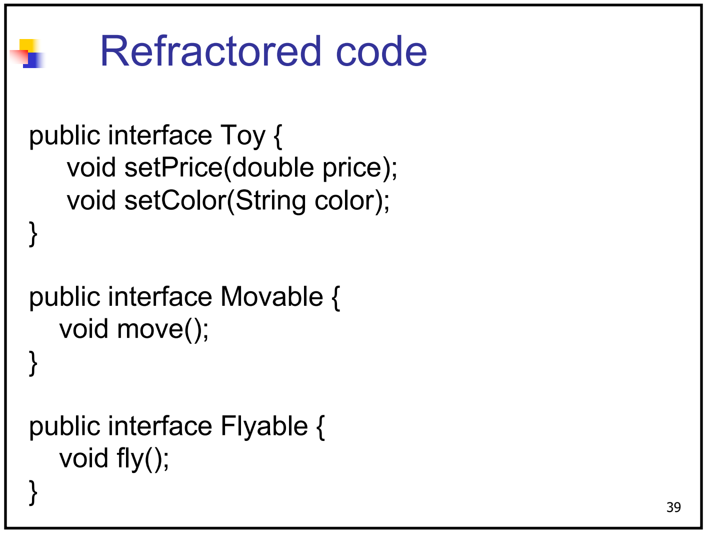
Example - Toy

```
public interface Toy {
    void setPrice(double price);
    void setColor(String color);
    void move();
    void fly();
}
```



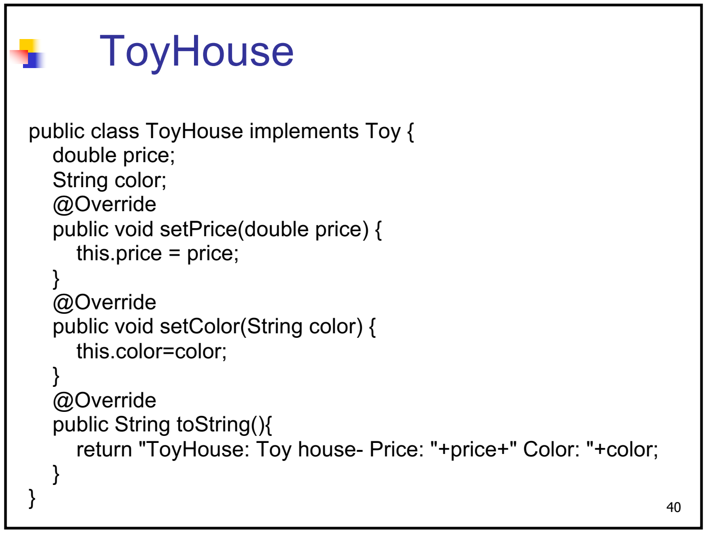
ToyHouse

```
public class ToyHouse implements Toy {
    double price;
    String color;
    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){}
    @Override
    public void fly(){}
}
```



Refractored code

```
public interface Toy {
     void setPrice(double price);
     void setColor(String color);
}

public interface Movable {
    void move();
}

public interface Flyable {
    void fly();
}
```



ToyHouse

```
public class ToyHouse implements Toy {
   double price;
   String color;
   @Override
   public void setPrice(double price) {
      this.price = price;
   }
   @Override
   public void setColor(String color) {
      this.color=color;
   }
   @Override
   public String toString(){
      return "ToyHouse: Toy house- Price: "+price+" Color: "+color;
   }
}
```

```
public class ToyCar implements Toy, Movable {
  double price;
  String color;
  @Override
  public void setPrice(double price) {
     this.price = price;
  }
  @Override
  public void setColor(String color) {
    this.color=color;
  }
  @Override
  public void move(){
     System.out.println("ToyCar: Start moving car.");
  }
  @Override
  public String toString(){
     return "ToyCar: Moveable Toy car- Price: "+price+" Color: "+color;
  }
}
```

41

```
public class ToyPlane implements Toy, Movable, Flyable {
  double price;
  String color;
  @Override
  public void setPrice(double price) {
     this.price = price;
  }
  @Override
  public void setColor(String color) {
     this.color=color;
  }
  @Override
  public void move(){
     System.out.println("ToyPlane: Start moving plane.");
  }
  @Override
  public void fly(){
     System.out.println("ToyPlane: Start flying plane.");
  }
  @Override
  public String toString(){
     return "ToyPlane: Moveable and flyable toy plane- Price: "+price+"
   Color: "+color;
  }
}
```

42

```
public class ToyBuilder {
   public static ToyHouse buildToyHouse(){
      ToyHouse toyHouse=new ToyHouse();
      toyHouse.setPrice(15.00);
      toyHouse.setColor("green");
      return toyHouse;
   }
   public static ToyCar buildToyCar(){
      ToyCar toyCar=new ToyCar();
      toyCar.setPrice(25.00);
      toyCar.setColor("red");
      toyCar.move();
      return toyCar;
   }
   public static ToyPlane buildToyPlane(){
      ToyPlane toyPlane=new ToyPlane();
      toyPlane.setPrice(125.00);
      toyPlane.setColor("white");
      toyPlane.move();
      toyPlane.fly();
      return toyPlane;
   }
}
```

43

Interface Segregation Principle vs. Single Responsibility Principle

- Both have the same goal: ensuring small, focused, and highly cohesive software components
- Single Responsibility Principle is concerned with classes
- Interface Segregation Principle is concerned with interfaces

44

Example 2 – RestaurantInterface

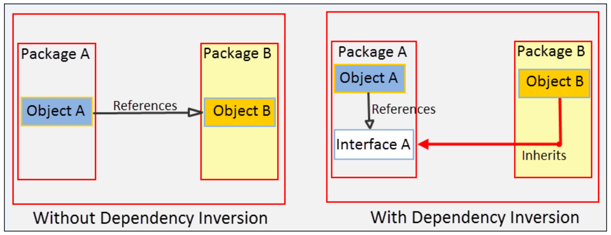
```
public interface RestaurantInterface {
   public  void acceptOnlineOrder();
   public  void takeTelephoneOrder();
   public  void payOnline();
   public  void walkInCustomerOrder();
   public  void payInPerson();
}
```

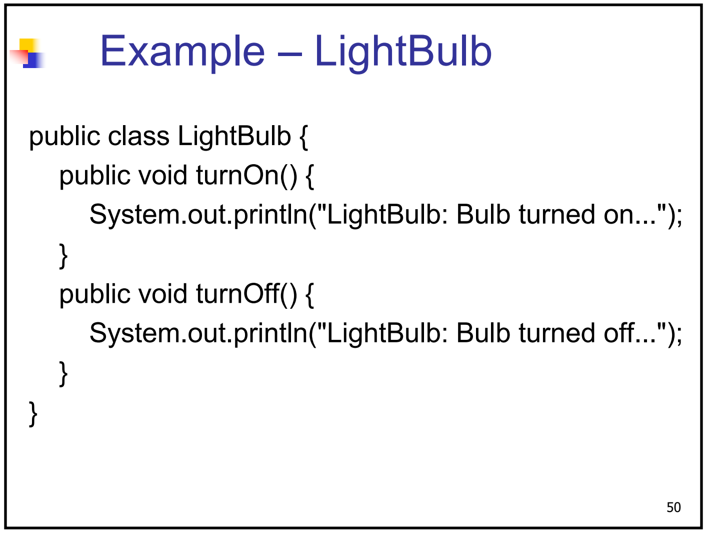
```
public class OnlineClientImpl implements RestaurantInterface{
 @Override
 public void acceptOnlineOrder() {
   //logic for placing online order
 }
 @Override
 public void takeTelephoneOrder() { //Not Applicable for Online Order
   throw new UnsupportedOperationException();
 }
 @Override
 public void payOnline() {
   //logic for paying online
 }
 @Override
 public void walkInCustomerOrder() { //Not Applicable for Online Order
   throw new UnsupportedOperationException();
 }
 @Override
 public void payInPerson() { //Not Applicable for Online Order
   throw new UnsupportedOperationException();
 }
}
```

5. Dependency Inversion Principle

- The Dependency Inversion Principle represents the last “D” of the five SOLID principles of object-oriented programming
  - *A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *B. Abstractions should not depend on details. Details should depend on abstractions.*
- Instead of a high-level module depending on a low-level module, both should depend on an abstraction.

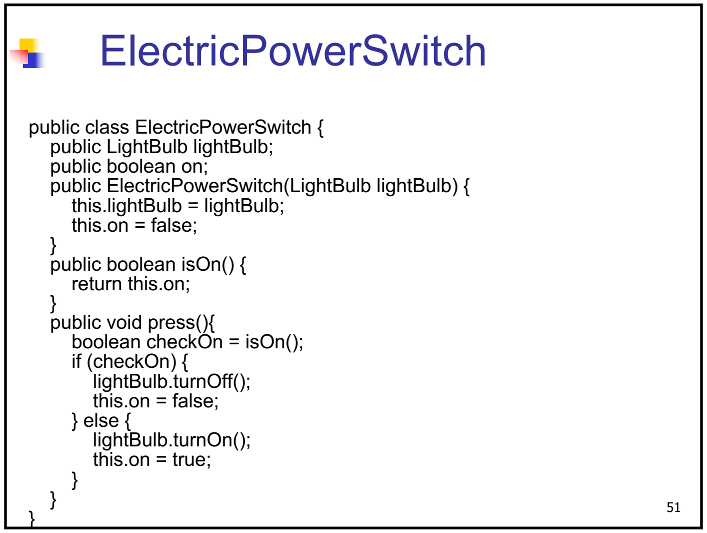
Dependency Inversion Principle





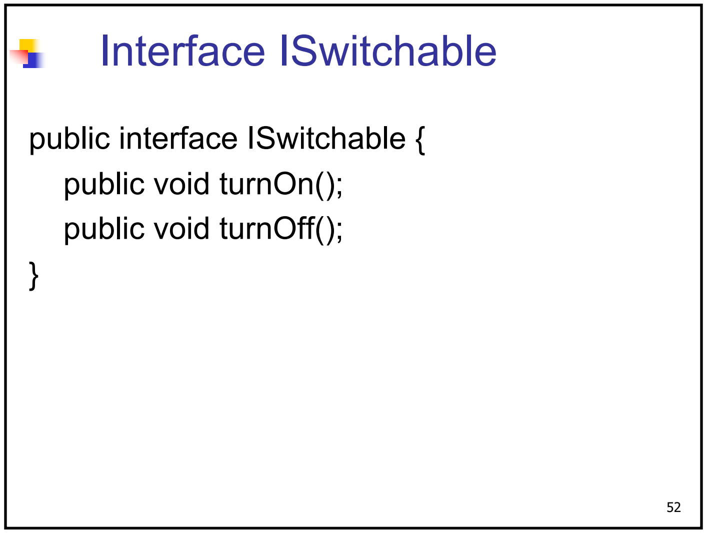
Example – LightBulb

```
public class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```



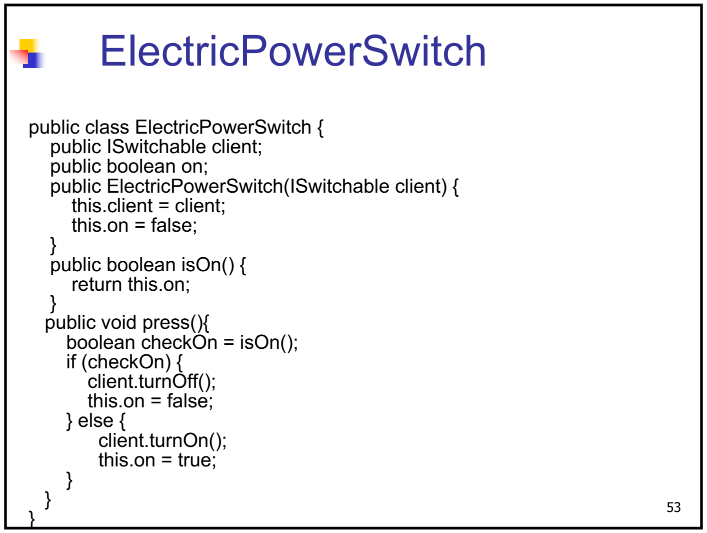
ElectricPowerSwitch

```
public class ElectricPowerSwitch {
    public LightBulb lightBulb;
    public boolean on;
    public ElectricPowerSwitch(LightBulb lightBulb) {
        this.lightBulb = lightBulb;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
    public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            lightBulb.turnOff();
            this.on = false;
        } else {
            lightBulb.turnOn();
            this.on = true;
        }
    }
}
```



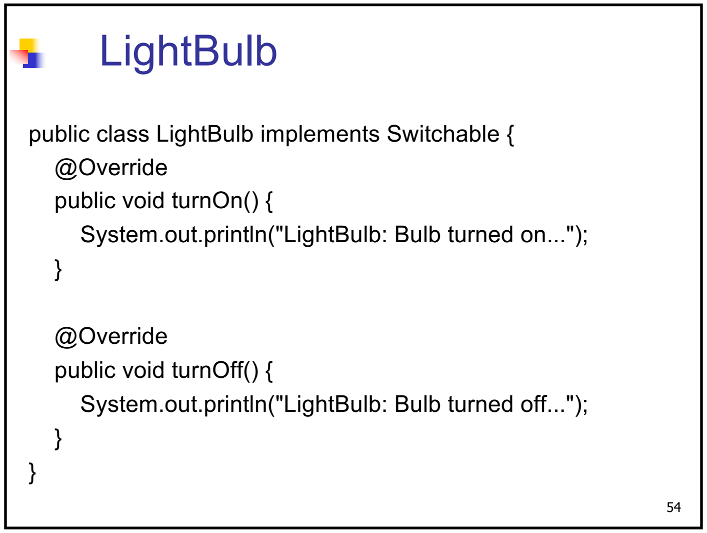
Interface ISwitchable

```
public interface ISwitchable {
    public void turnOn();
    public void turnOff();
}
```



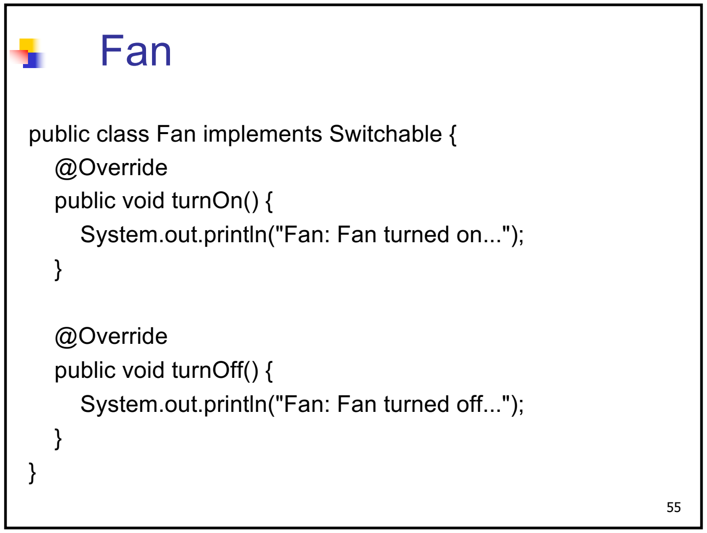
ElectricPowerSwitch

```
public class ElectricPowerSwitch {
    public ISwitchable client;
    public boolean on;
    public ElectricPowerSwitch(ISwitchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
       boolean checkOn = isOn();
       if (checkOn) {
           client.turnOff();
           this.on = false;
       } else {
              client.turnOn();
              this.on = true;
       }
   }
}
```



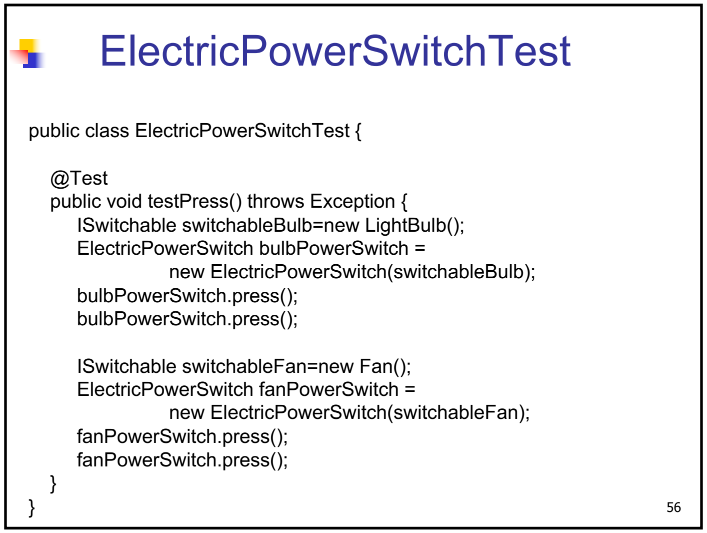
LightBulb

```
public class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```



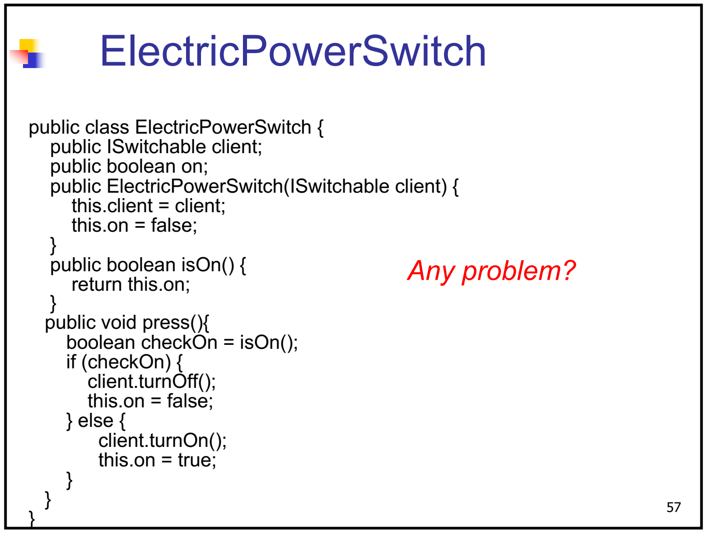
Fan

```
public class Fan implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("Fan: Fan turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("Fan: Fan turned off...");
    }
}
```



ElectricPowerSwitchTest

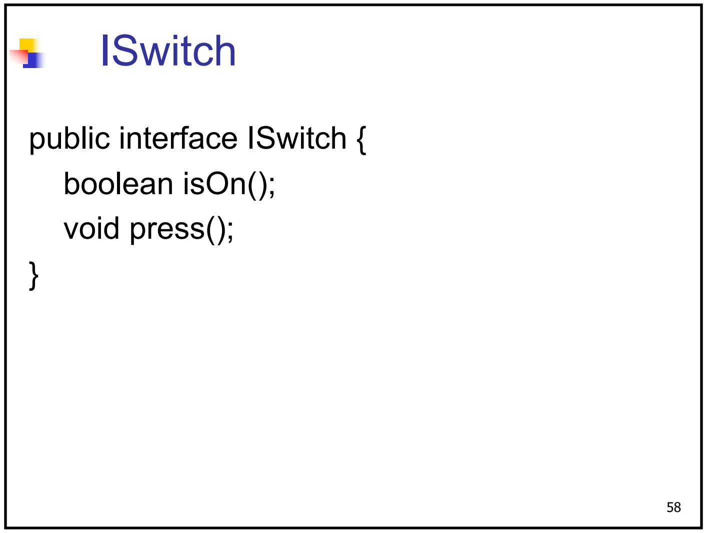
```
public class ElectricPowerSwitchTest {

    @Test
    public void testPress() throws Exception {
        ISwitchable switchableBulb=new LightBulb();
        ElectricPowerSwitch bulbPowerSwitch =
                      new ElectricPowerSwitch(switchableBulb);
        bulbPowerSwitch.press();
        bulbPowerSwitch.press();

        ISwitchable switchableFan=new Fan();
        ElectricPowerSwitch fanPowerSwitch =
                      new ElectricPowerSwitch(switchableFan);
        fanPowerSwitch.press();
        fanPowerSwitch.press();
    }
}
```



ElectricPowerSwitch

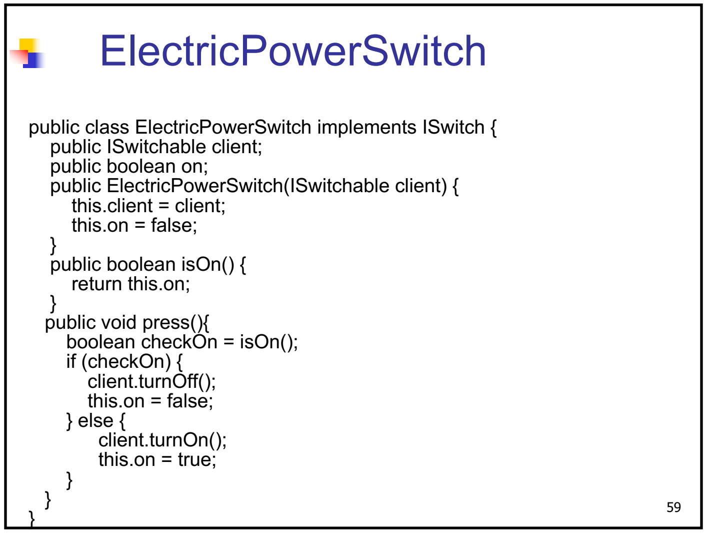
```
public class ElectricPowerSwitch {
    public ISwitchable client;
    public boolean on;
    public ElectricPowerSwitch(ISwitchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
       boolean checkOn = isOn();
       if (checkOn) {
           client.turnOff();
           this.on = false;
       } else {
             client.turnOn();
             this.on = true;
       }
   }
}
```

Any problem?



ISwitch

```
public interface ISwitch {
    boolean isOn();
    void press();
}
```



ElectricPowerSwitch

```
public class ElectricPowerSwitch implements ISwitch {
    public ISwitchable client;
    public boolean on;
    public ElectricPowerSwitch(ISwitchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
       boolean checkOn = isOn();
       if (checkOn) {
           client.turnOff();
           this.on = false;
       } else {
             client.turnOn();
             this.on = true;
       }
   }
}
```