

# Interfaces



# Interfaces

Interfaces are a way to describe *what* classes should do without specifying *how* they should do it.



# Introduction to interfaces

- We have already seen how an abstract base class links *related* derived class through inheritance
  - The overridden virtual methods of the abstract base class must be implemented in the derived classes
- There are also occasions when we want *unrelated* classes to exhibit similar behaviour



# Defining Interfaces

```
interface NameofInterface [extends AnotherInterface]
{
    method1;
    method2;
    ...
    constant1;
    constant2; ...
}
```

- All **methods** are **abstract** by default, no need for modifier **abstract**
- All **fields** are **constants** by default, no need for modifier **static final**
- All methods and constants have **public** access by default, no need for modifier **public**.



# Using interfaces

---

- To implement an **interface**, include the **implements** clause in a class definition, and then create the methods defined by the interface

```
modifier class classname [extends superclass]  
[implements interface [,interface...]] {  
// class-body  
}
```

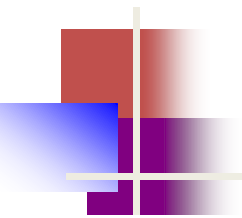


# Understand Interface

---

- An Interface is nothing but a **contract** as to how a class should behave. It just declares the behavior as empty methods and the implementing class actually writes the code that will determine the behavior.
- When you implement an interface, you're agreeing to adhere to the **contract** defined in the interface.
- That means you're agreeing to provide legal implementations for every method defined in the interface, and that anyone who knows what the interface methods look like can rest assured that they can invoke those methods on an instance of your implementing class.

```
interface ShowMessage {  
    void showBrand(String s);  
}  
  
class TV implements ShowMessage {  
    public void showBrand(String s) {  
        System.out.println(s);  
    }  
}  
  
class PC implements ShowMessage {  
    public void showBrand(String s) {  
        System.out.println(s);  
    }  
}
```



```
public class Example {  
    public static void main(String args[]) {  
        ShowMessage sm;           // Interface variable  
        sm=new TV();              //Refers to an object  
        sm. showBrand("Chang Cheng"); //TV' showBrand()  
        sm=new PC();              // Refers to an object  
        sm. showBrand("Thinkpad"); //PC' showBrand()  
    }  
}
```

When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.





# Polymorphism

---

- **Interface variable** holds reference to object of a class that implements the interface.

```
ShowMessage sm;
```

```
sm=new TV();
```

```
sm=new PC();
```

- **Note** that the object to which sm refers doesn't have type ShowMessage;
- The type of the object is some class that implements the ShowMessage interface



# Purpose of Polymorphism

---

- You can call any of the interface methods:  
**sm. showBrand**("Chang Cheng");  
**sm. showBrand**("Thinkpad");
- Which method is called?
- Depends on the actual object.
- If sm refers to a **TV**, calls **TV. showBrand()**
- If sm refers to a **PC**, calls **PC. showBrand()**



# Polymorphism

---

- Polymorphism means "many forms" (**poly = many, morphos = form**) : Behavior can vary depending on the actual type of an object
- The property that we can call **sm. showBrand()** with multiple contexts is an instance of polymorphism
- Called **late binding**: resolved at runtime



# Interfaces Example

---

- Design a small system which includes different types: **Teacher**, **School** and **Printer**.
- The requirements are shown as below:
  - **School** and **Teacher** have a same method: **detail()** to print the detail information.
  - School has a field: **Printer** which can print School and Teacher information.
  - The system should be easy to be extended and maintained.



# Interfaces Example

- Any problem?

```
public class Teacher {  
    public String detail() {  
        return "Software Teacher";  
    }  
}
```

```
public class Printer {  
    public void print(String content) {  
        System.out.println("Printing: ");  
        System.out.println(content);  
    }  
}
```

```
public class School {  
    private Printer printer = new Printer();  
  
    public String detail() {  
        return "This is ECE";  
    }  
  
    public void printP(Teacher t){  
        printer.print(t.detail());  
    }  
    public void printP(School s){  
        printer.print(s.detail());  
    }  
}
```

Adding a new class implies that **printP(ClassType var)** needs to be added.

- — Poor extensibility and maintainability
- — Does not meet the requirements



# Interface-based Programming

---

- *Interface-based programming* is a popular and convenient technique frequently used in object-oriented software development.
- Interfaces make a design more flexible because it can reduce coupling.



# Coupling

---

- **Coupling** is where one type depends on another type.
- **Tight coupling** occurs when the type of a variable is not abstract, and cannot be changed without having to change other code accordingly.
- **Loose coupling** leads to code that is isolated, in the sense that changing one piece of code is less likely to affect another section of code. When coupling is tight, changing code in one location can cause a ripple effect that affects code elsewhere.

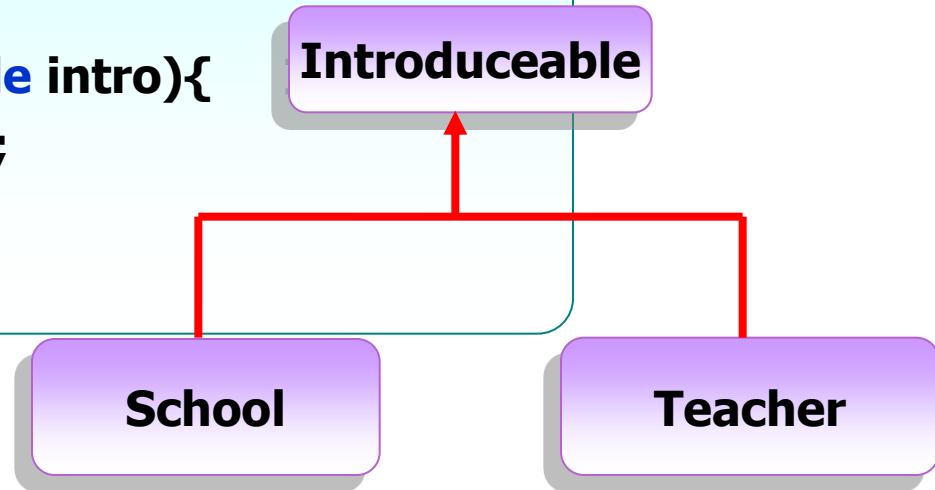
# Interface-based Programming

```
public class School implements Introduceable{  
    private Printer printer = new Printer();  
    public String detail() {  
        return "This is ECE";  
    }  
    public void printP(Introduceable intro){  
        printer.print(intro.detail());  
    }  
}
```

Introduceable

School

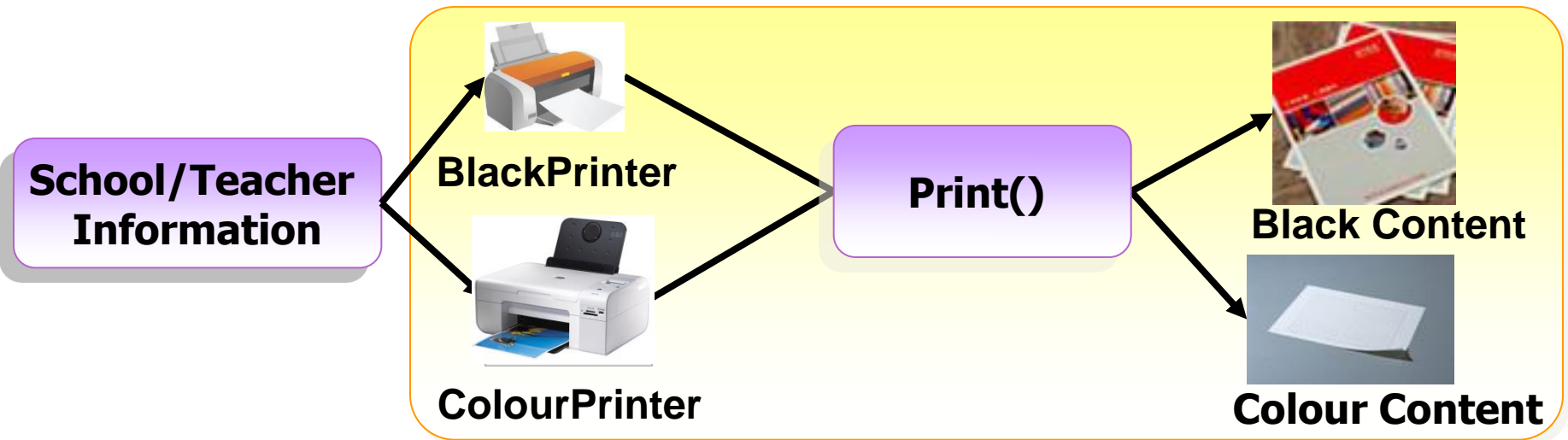
Teacher





# Interface-based Programming

- Upgrade the previous system, the requirements are:
  - The **Printer** has different kinds: **BlackPrinter** and **ColourPrinter**
  - School can use both Printer to print the information for **Teacher** or **School**
  - The system should be easy to be extended and maintained.





# Interface-based Programming

## **ANALYZE THE PROBLEM**

**BlackPrinter** and **ColourPrinter** have a same method: **print()**, however, the implementation of the **print()** methods are different.

## **SOLUTION**

Using Interfaces can solve this problem.

```
public interface PrinterFace {  
    public void print(String content);  
}
```



# Interface-based Programming

```
public class ColorPrinter
    implements PrinterFace {
    public void print(String content) {
        System.out.println("Colour Print: ");
        System.out.println(content);
    }
}
```

```
public class BlackPrinter
    implements PrinterFace {
    public void print(String content) {
        System.out.println("Black Print: ");
        System.out.println(content);
    }
}
```

# Interface-based Programming

```
public class School implements Introduceable{  
    private PrinterFace printer;  
    public void setPrinter(PrinterFace p) {  
        this.printer = p;  
    }  
    public String detail() {  
        return "This is CIE ";  
    }  
    public void printP(Introduceable intro){  
        printer.print(intro.detail());  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        School school=new School();  
  
        //set BlackPrinter  
        school.setPrinter(new BlackPrinter());  
        school.printP(school);  
  
        //set ColourPrinter  
        school.setPrinter(new ColorPrinter());  
        school.printP(school);  
    }  
}
```



# Interfaces Review

- Using **interface**, we can specify what a class must do, but not how it does it.
- **Interfaces** are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- Once it is defined, any number of classes can implement an **interface**.
- One class can implement any number of **interfaces**.
- To implement an interface, a class must create the complete set of methods defined by the **interface**.
- Each class is free to determine the details of its own implementation



# Abstract Classes and Interfaces

Two stories as one



# The Door Story

- Door is an abstract concept which has two different behaviors: **open** and **close**, we can use **abstract class** or **interface** to define

```
abstract class Door{  
    abstract void open();  
    abstract void close();  
}
```

```
interface Door{  
    void open();  
    void close();  
}
```



# The Door Story

Other classes can:

- **extends** the abstract class: Door
- **Implements** the interface: Door

Now, it seems that the abstract class and the interface are almost the same. What if the Door has an alarm?

**How to design our program?**





# Method 1

```
abstract class Door{  
    abstract void open();  
    abstract void close();  
    abstract void alarm();  
}
```

```
class AlarmDoor extends Door{  
    void open(){...}  
    void close(){...}  
    void alarm() {...}  
}
```

---

```
interface Door{  
    void open();  
    void close();  
    void alarm();  
}
```

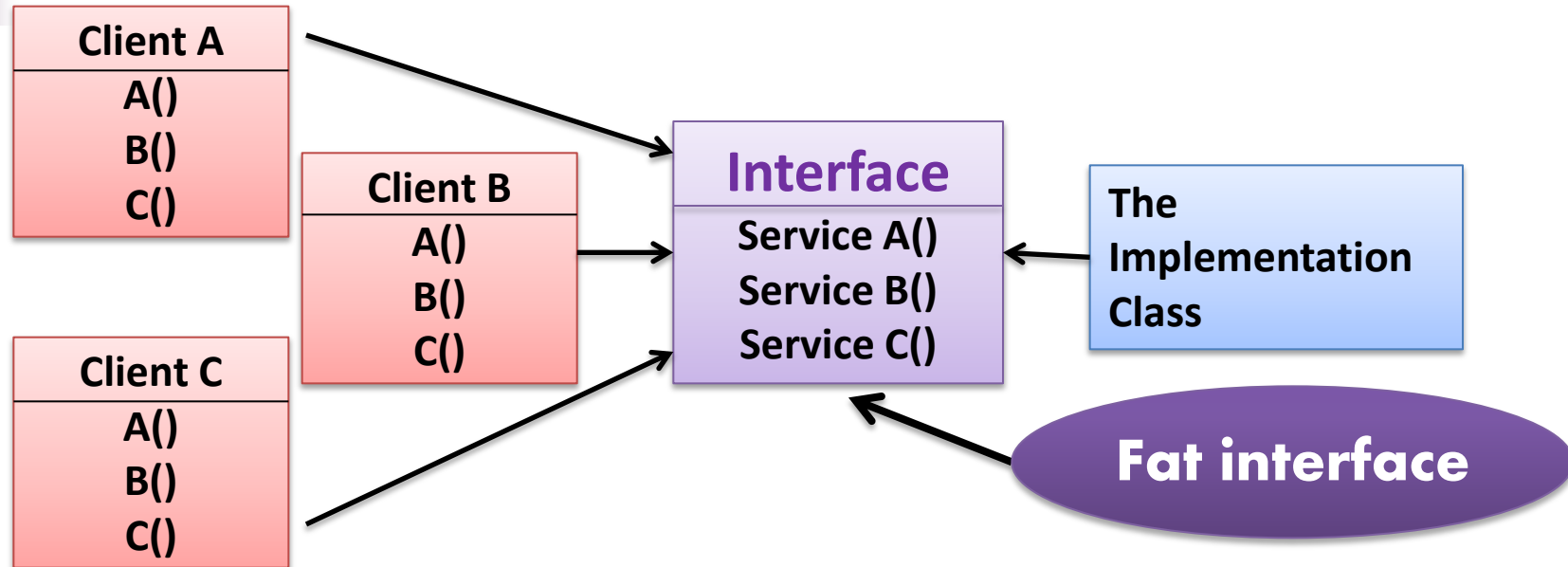
```
class AlarmDoor implements Door{  
    void open(){...}  
    void close(){...}  
    void alarm() {...}  
}
```



# Interface Segregation Principle

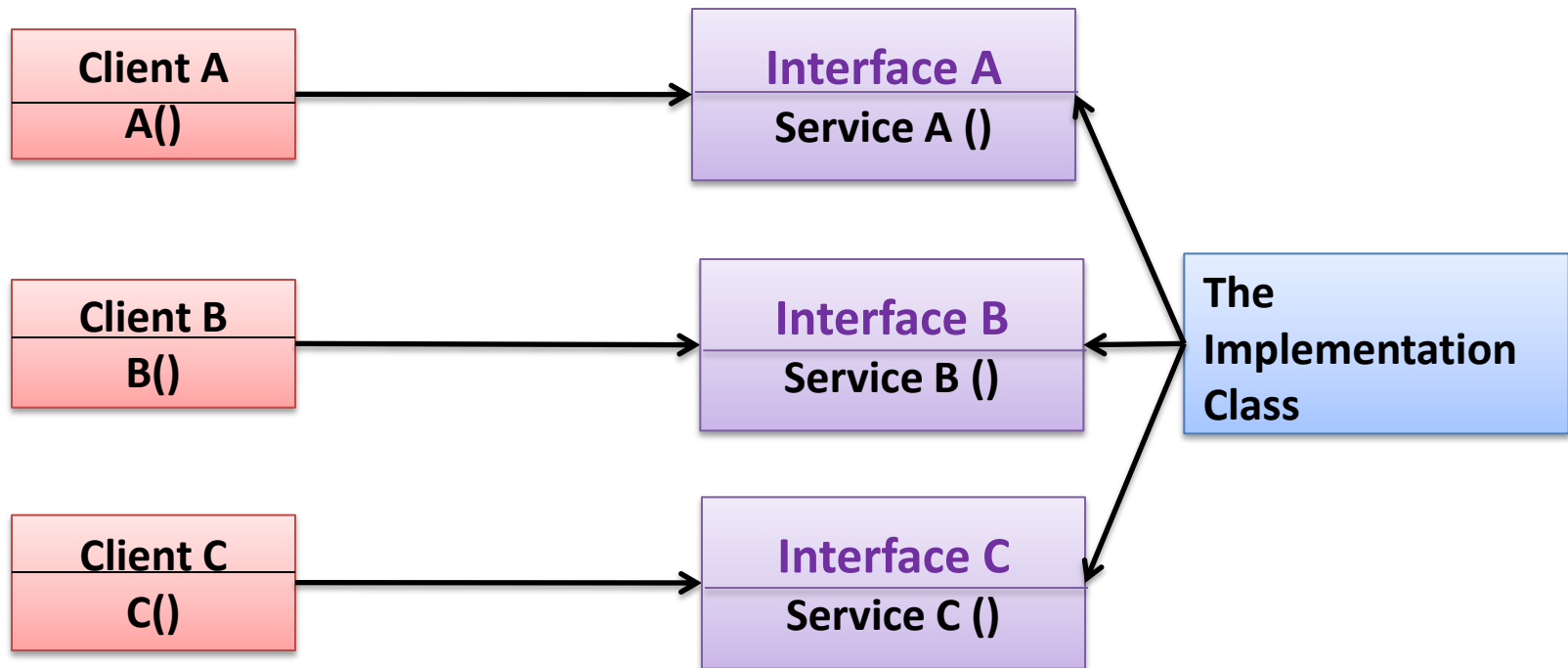
- The **Interface Segregation Principle** states that clients should not be forced to implement interfaces they don't use.
- Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving ONE submodule.

# Interface Segregation Principle



- This example violates the **Interface Segregation Principle**.
- We have a fat interface which has three different services: A(), B() and C() to three different clients: A, B and C.
- Why does client B and C need to know about A()?

# Interface Segregation Principle



Each client only what to know about the services it uses.

Hence, we define an interface as taking on a ROLE.

Now A(), B() and C() are placed in different interfaces, assuming they are three different roles.



# Role Interfaces

- A role interface is defined by looking at a specific interaction between suppliers (interfaces and their implementations) and consumers (users of the code).
- A supplier component will usually implement several role interfaces, one for each of these patterns of interaction.



# Interface Segregation Principle

**METHOD1 VIOLATES THE ISP**



**The concepts of Door and Alarm get mixed up.**



## Method 2

---

Because open()+close() and alarm() belong to different concepts, we should put them into two different abstract classes according to ISP by:

- Both use **abstract class** to define;
- Both use **interface** to define;
- One use **abstract class** and the other use **interface** to define



# Use abstract class

---

- Multi-inheritance is not supported by Java, it is impossible for both of the concept to be defined by using abstract class.







# Use interface

## **ALARMDOOR IS A DOOR OR AN ALARM?**

We notice that AlramDoor is a door, therefore, if both of the concept to be defined by using interface, it can not reveal our design purpose.



# Use abstract class and interface

---

- When an **“is-a”** relationship exists, using **abstract class**. For the concept of Door, we should use **abstract class** to define.
- When a **“like-a”** relationship exists, using **interface**. Because the AlarmDoor has an alarm function, we should use **interface** to define this behavior.



# Implementation

```
abstract class Door{  
    abstract void open();  
    abstract void close();  
}
```

```
interface Alarm{  
    void alarm();  
}
```

```
class AlarmDoor extends Door implements Alarm{  
    void open(){...}  
    void close(){...}  
    void alarm(){...}  
}
```