

Universitatea Tehnica din Cluj-Napoca  
Departament Calculatoare

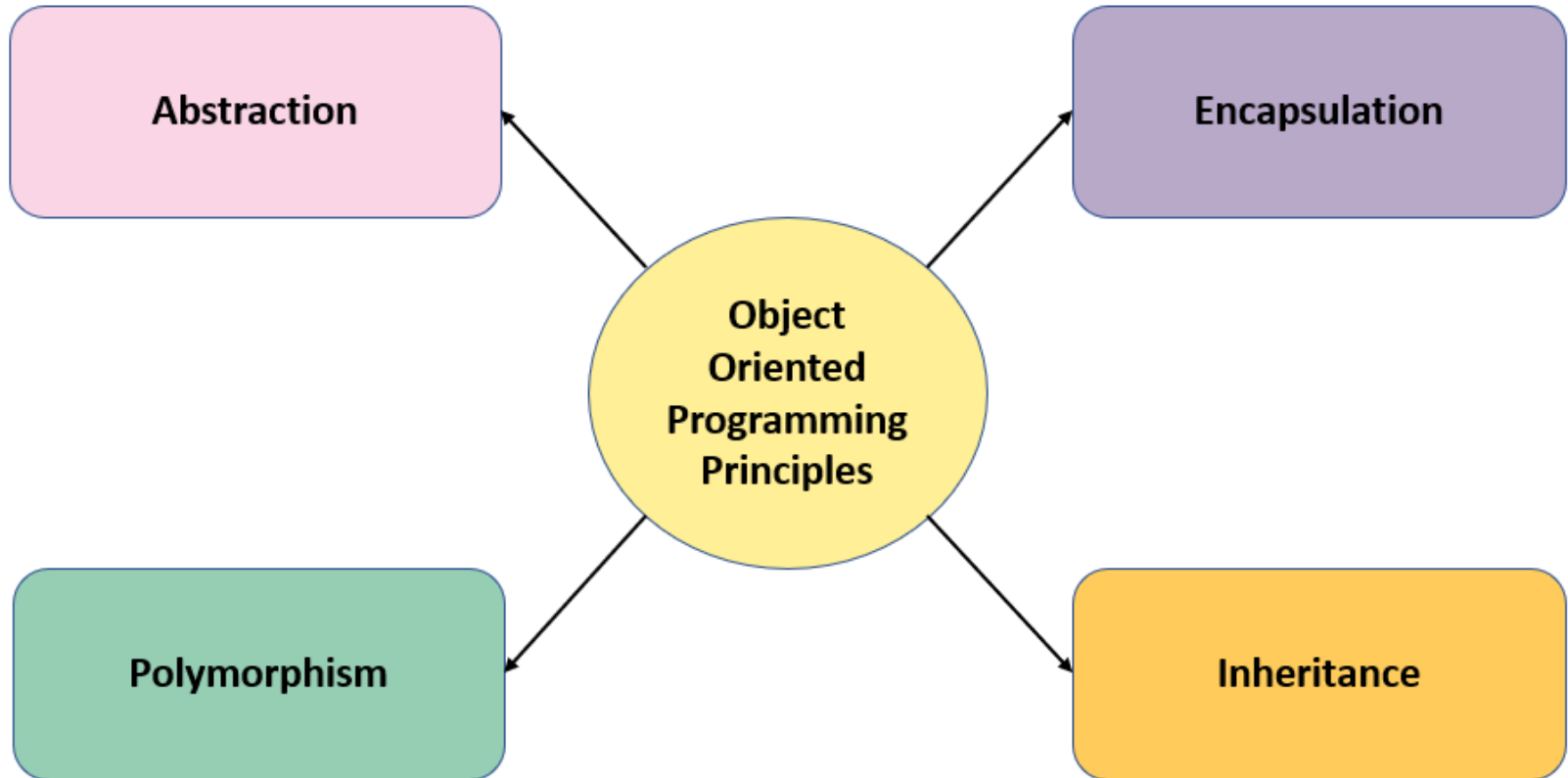
# Programming Techniques in Java

## OOP Paradigms

T. Cioara, V. Chifu, C. Pop  
2025

# OOP Paradigms

---



# Abstraction

- Allows us to take a simpler view of a complex concept.



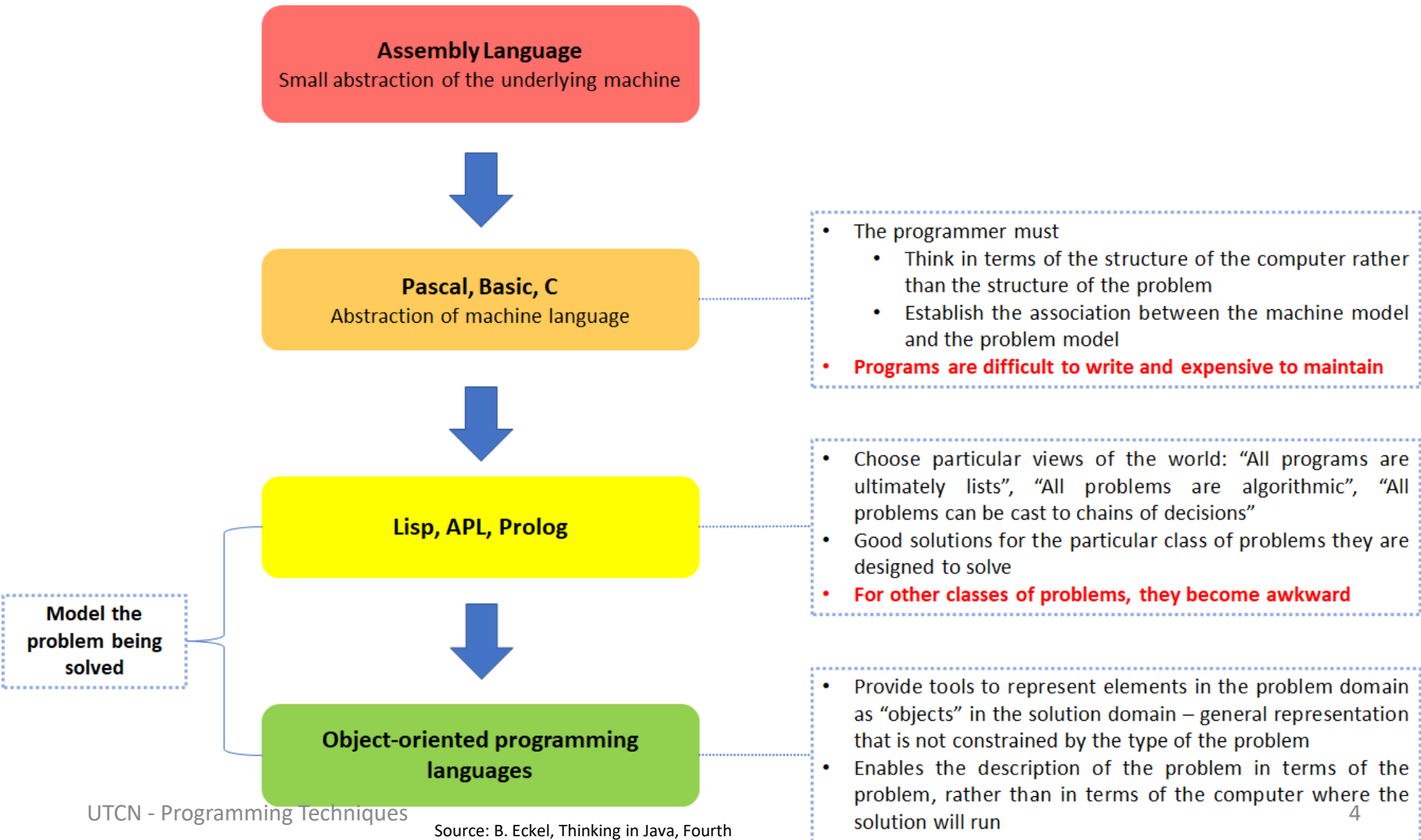
- State of the art: the Digital Twins

[\[Source\]](#)



- Identifying only the required characteristics of an object ignoring the irrelevant details

# Abstraction



# Abstraction

**Class is the fundamental concept in object-oriented programming**

FIND REAL WORLD ENTITIES

IDENTIFY THE OBJECTS AND THEIR ATTRIBUTES

DETERMINE WHAT CAN BE DONE WITH EACH OBJECT

DETERMINE HOW OBJECTS INTERACT

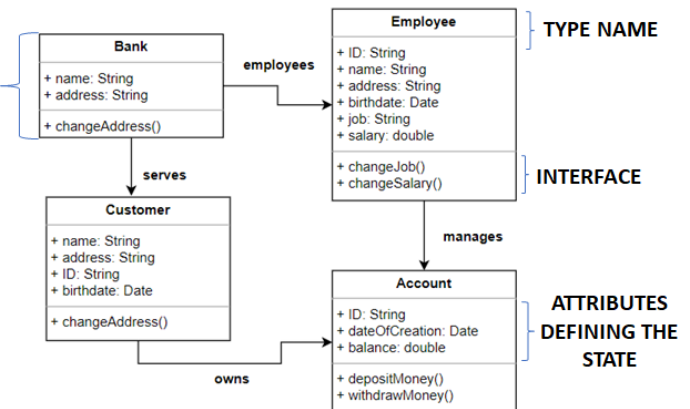
DETERMINE THE PARTS OF EACH OBJECT THAT WILL BE VISIBLE TO OTHER OBJECTS

DEFINE THE INTERFACE OF EACH OBJECT

Bank Management Problem



ABSTRACT  
DATA TYPE  
DEFINITION



# Encapsulation

---

## Problem

- Tight coupling because a class exposes its methods and fields
- Difficult maintenance and modification
- Containment of development bugs

## Solution

- Hide implementation details through encapsulation
- Assign fields and methods the most possible restrictive visibility while still providing the needed functionality
- Constructors should build only valid objects
- Accessors (get methods) and mutators (set methods)

## Advantages

- Mutators allow assigning only valid values to instance variables
- Accessors and mutators insulate the class from changes to a property's implementation
- Thread – safe objects (due to private qualifier)

# Encapsulation

- Non-encapsulated vs. encapsulated class

## NON-ENCAPSULATED CLASS

```
public class Employee {  
    public int employeeID;  
    public String firstName;  
    public String lastName;  
}  
// ... In a program we may have  
Employee emp = new Employee();  
emp.employeeID = 123456;  
emp.firstName = "John";  
emp.lastName = "Smith";
```

## ENCAPSULATED CLASS

```
public class Employee {  
    private int employeeID;  
    private String firstName;  
    private String lastName;  
  
    public Employee (int id, String fn, String ln) {  
        // check validity of parameters  
        // ... assign values if ok  
        // ... throw exception otherwise  
    }  
  
    public int getEmployeeID() { return employeeID;}  
  
    public void setEmployeeID(int id) {  
        // check validity of parameter  
        // ... assign values if ok  
        // ... throw exception otherwise  
    }  
    ... similar for the rest of set/get methods  
}
```

# Encapsulation

---

- Example
  - change employeeID from an **int** to a **String** without affecting other classes
  - perform the appropriate conversions in the accessor and mutator methods
  - The client classes may read / modify it without observing any change
    - due to the encapsulation of employeeID (behind mutator and accessor methods)

```
public class Employee {  
    private String employeeID;  
    private String firstName;  
    private String lastName;  
  
    public int getEmployeeID() {return Integer.parseInt(employeeID); }  
    public void setEmployeeID(int id) { employeeID =Integer.toString(id); }  
    ... }  
}
```



# Encapsulation

---

- **Managing side-effects**

- Any data modification that is observable outside the method
  - The explicit parameter object (i.e., this object)
  - Other objects using explicit parameters, or accessible static fields

**In common OO programming practice, all applications need to have side effects because all applications have/manipulate states. However, try to get rid of unnecessary side-effects!**

## **Code with unnecessary side-effects**

- Difficult to reason about program behavior, debug
- Difficult to prove program correctness
- Hard to use the code concurrently
- Hard to optimize
- Hard to test
- Hard to reuse
- Is fragile

# Encapsulation

---

## Method without side-effect

- Always returns the same answer when it is called
- “pure function” (in math’s terms)
- Are better composed to form up new functionality
- Thread safe and could be better debugged

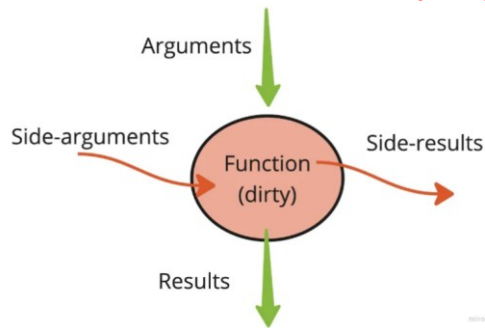
## Functional programming

- No side effects
- No state
- Immutable variables
- Favors recursion instead of loops


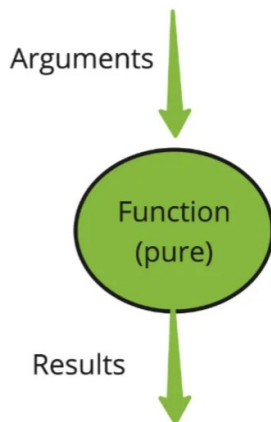
# Encapsulation

- Managing side-effects





## Method with side-effects (dirty)




## Method without side-effects (pure/clean)



```
public int sumWithSideEffects(int a, int b) {  
    Date date = new Date();  
    int result = a + b;  
    System.out.println("At time " + date.toString() +  
        " the result is " + result);  
    return result;  
}
```



```
public int sumPure(int a, int b) {  
    return a+b;  
}
```



[Source](#)

# Encapsulation

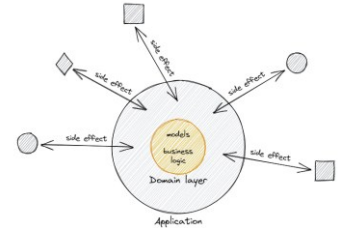
- **Side-effects as interactions with the outside world**

- Examples

- Reading a file, and/or writing to a file
- Making a network request (calling an API, downloading a file...)
- Reading from a global state (e.g. global variable, a parameter from the parent's closure...)
- Throwing/intercepting an exception
- Logging messages to the console

- A program with no side effects is not very useful

- We need interactions with the outside world (e.g., getting input from a user, writing to a database, or displaying something to the screen)



[Source](#)

# Encapsulation

- Examples of methods with side-effects

Side effect of a method  
through accessible static field

```
public void push() {  
    if (stk.isFull())  
        System.out.println ("Out of Space" );  
    // Wrong approach  
}
```

Instead, an  
exception  
should have  
been thrown!

Method with side-effects

```
public class Ex3_NonPure {  
    private int partial = 0;  
    public int sum (int iV) {  
        partial += iV;  
        return partial;  
    }  
}
```

Method with side-effects

```
public class Ex4 {  
    ...  
    public int processAddr(Address adr){  
        ...  
        adr.setStreet(newStreet) { ...}  
    }  
}
```

# Encapsulation

- Refactoring code for reducing side-effects
  - Simple approach - example

```
public class FileProcessor {...
    public static void processFile(String filePath) {
        try {
            List<String> lines = Files.readAllLines(Path.of(filePath)); // Side argument
            for (String line : lines) {
                System.out.println(line.toUpperCase()); // Side effect (printing)
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

**Method with side-effects**

```
public class FileProcessor {
    // Step 1: Move side-argument (file reading) up
    public static List<String> readFile(String filePath) throws IOException {
        return Files.readAllLines(Path.of(filePath));
    }
    // Step 2: Convert processing into a pure function
    public static List<String> processLines(List<String> lines) {
        return lines.stream().map(String::toUpperCase).toList();
    }
    // Step 3: Move side-result (printing) down
    public static void displayLines(List<String> processedLines) {
        processedLines.forEach(System.out::println);
    }
}
```

**Method without side-effects**

# Encapsulation

- **Law of Demeter**

- A read/write/call method **should only use**
  - Instance fields of its class
  - Parameters passed as arguments
  - Objects that it constructs with new
  - Methods of its class
  - Global static fields (in Java)
- A method that follows the Law of Demeter **should NOT**
  - operate on global (non-static) objects or objects that are a part of another object
  - ask another object to give it a part of its internal state to work on (i.e., avoid call getters of other objects)
- A class should not return a reference to an object that is a part of its internal implementation

~~obj.getX().getY().getZ().doSomething();~~

# Encapsulation

- **Law of Demeter - Benefits**

- Simplifies methods by limiting the number of types used inside them
- Restricted operation only on types that you directly have access
- Simplifies changes needed to be done in the future during code extension or maintenance => easier code maintenance

## Situation 1 - law respected

```
public final class Order {  
    public void close() {  
        sendCloseOrderMessage();  
    }  
    private void sendCloseMessage() {  
        ...  
    }  
}
```

## Situation 2 – law violated

```
public final class Order {  
    private int orderValue;  
    private Customer cust;  
    ...  
    public void closeCustOrder () {  
        clearInvoice(cust.getDebt().clearDebt(orderValue));  
    }  
    public void clearInvoice(...) { ... }  
}
```



# Encapsulation

- **Law of Demeter - Examples**

```
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;
    public Car() {
        this.engine = new Engine();
    }
    public Engine getEngine() {
        return engine; // Violation of LoD
    }
}

class Driver {
    public static void main(String[] args) {
        Car car = new Car();
        car.getEngine().start(); // Violates LoD
    }
}
```

```
class Engine {
    public void start() {
        System.out.println("Engine started");
    }
}

class Car {
    private Engine engine;
    public Car() {
        this.engine = new Engine();
    }
    public void startCar() {
        engine.start(); // Encapsulation,
    }
}

class Driver {
    public static void main(String[] args) {
        Car car = new Car();
        car.startCar(); // No direct access to Engine
    }
}
```

# Inheritance



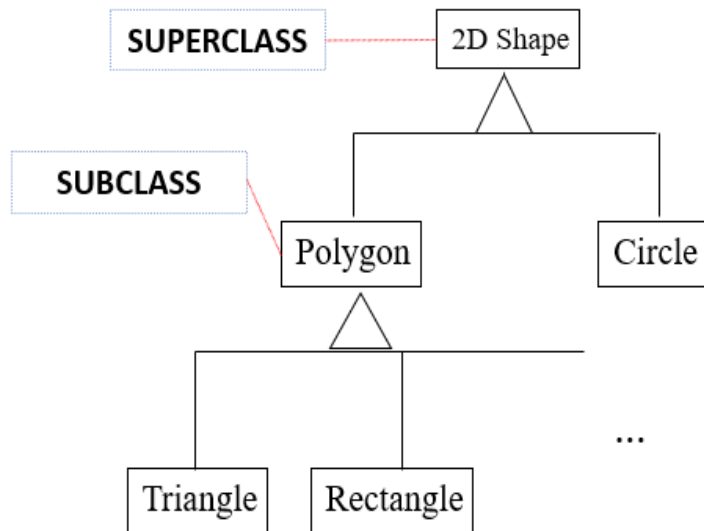
A class implementation is derived from another class



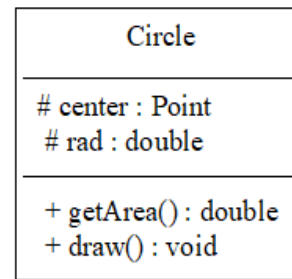
Code-reuse



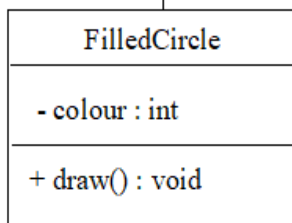
Use of an object as its own type or of its base type



UTCN - Programming Techniques



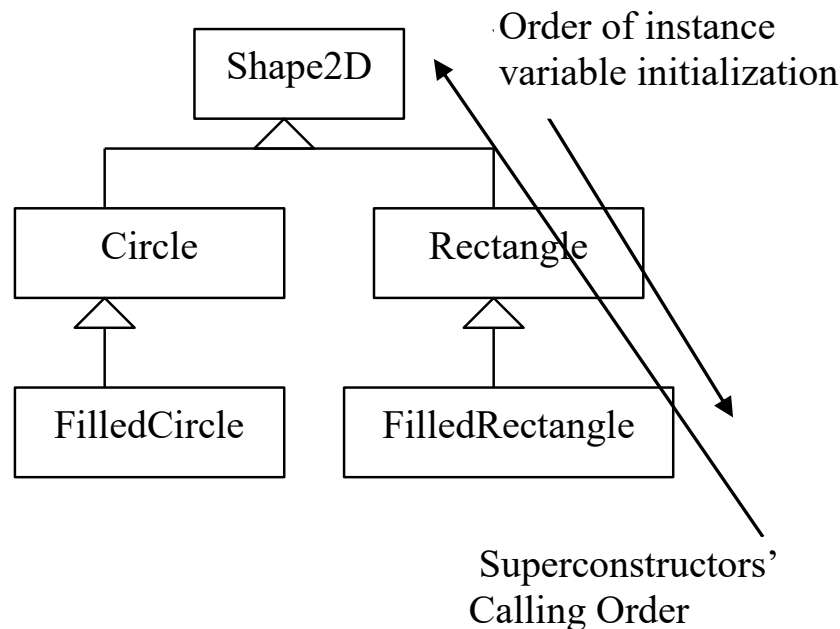
```
class Circle {
    protected Point center;
    protected double rad;
    public double getArea() { ... }
    public void draw() { ... }
}
```



```
class FilledCircle extends Circle {
    private int colour;
    public void draw() { ... }
}
```

# Inheritance

- Handling constructors in inheritance hierarchies
  - Inheritance introduces a strong coupling between constructors
  - Limit the number of hierarchy levels to 3 or 4
- **For code reuse, prefer delegation over inheritance**



# Inheritance

- **Handling constructors in inheritance hierarchies - example**

```
public class Shape2D {
    protected Point org;
    public Shape2D(Point org) { this.org =org; }
    public String whoAreYou() {return "SHAPE2D"; }
    public void draw() {
        System.out.println("I am a generalized Shape2D.");
    }
    public double perimeter() { return 0.0; }
    public double area() { return 0.0; }
}
```

```
public class Circle extends Shape2D {
    protected double radius;
    public Circle(Point pc, double radius) {
        super(pc);
        this.radius = radius;
    }
    ...overriding...
}
```

```
public class FilledCircle extends Circle {
    private int color;
    public FilledCircle(Point pc, double
                        radius, int color) {
        super(pc, radius);
        this.color = color;
    }
    ...overriding ...
}
```

# Inheritance

*this* used to refer current-class's instance as well as static members

- to refer instance variable of current class
- to invoke or initiate current class constructor
- can be passed as an argument in the method OR constructor call
- can be used to return the current class instance

*super* is used to refer super-class's instance as well as static members

- *super* is also used to invoke super-class's method or constructor

use *this* and *super* anywhere except for static contexts

# Inheritance

Feature	this	super
Refers to	Current instance	Parent class
Used for	Instance variables, methods, constructors	Parent variables, methods, constructors
Calls constructor?	this(...) (same class)	super(...) (parent class)
Allowed in static methods?	No	No
Allowed in instance methods?	Yes	Yes

```
import java.awt.Color;

public class ColoredPoint extends Point{
    private Color color;

    public ColoredPoint(double x, double y, Color color) {
        super(x,y);
        this.color = color;
    }

    public ColoredPoint(double x, double y) {
        this(x, y, Color.black);
    }

    public ColoredPoint() {
        color = Color.black;
    }
}
```

What does  
it call?

What is the role of "this" here?

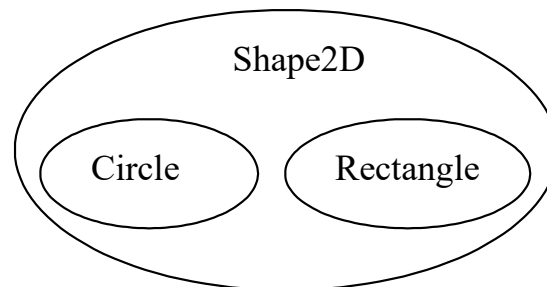
What is the role of "this" here?

Is it correct?!

# Inheritance

---

- Inheritance and subtypes
  - A subclass extends features of its superclass
  - Every instance of the subclass is an instance of the superclass and not vice versa
    - Each class defines a type, and all instances are the set of valid type values
    - Every instance of a subclass is also an instance of its superclass
    - The type defined by subclass is valid subtype of the type defined by its superclass
- Subtype relation applies to class types, interface types and primitive types
  - Class inheritance, interface implementation and extension



# Inheritance

Condition	Subtype Relationship
If C1 extends C2	C1 is a subtype of C2
If I1 extends I2	I1 is a subtype of I2
If C implements I	C is a subtype of I
For every I	I is a subtype of Object
For every T	T[] is a subtype of Object
If T1 is a subtype of T2	T1[] is a subtype of T2[]
For any C that is not Object	C is a subtype of Object

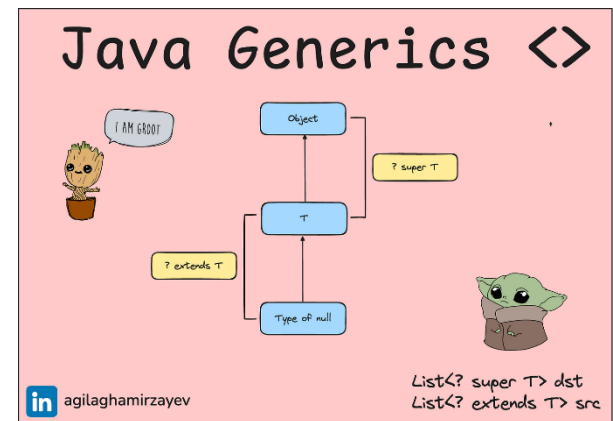
A **subtype** is a class that **can be used in place of another type** without breaking the expected behavior

```
class Animal {}  
class Dog extends Animal {}
```

```
Dog[] dogs = new Dog[3];  
Animal[] animals = dogs;
```

```
Animal[] animals = new Dog[3]; // Allowed at compile
```

```
animals[0] = new Dog(); // Works fine  
animals[1] = new Animal(); // run time error
```





# Inheritance

---

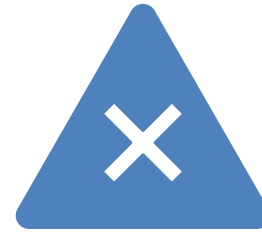


## Substitutability of subtypes

A value of a subtype can appear wherever a super-type is expected

In the **context of classes and objects**

- A subclass object can appear wherever a superclass object is expected
- An instance of a subclass can always substitute for an instance of its superclass



## Conversion of reference types

Governed by subtype relation

### Upcasting

- The conversion of a subtype to one of its supertypes
- Widening is always allowed
- Is carried out implicitly whenever necessary

# Inheritance

- **Conversion of reference types – downcasting techniques**
  - The conversion of a supertype to one of its subtypes
  - Requires explicit casts
  - Allowed at compile time
  - Not always safe – may generate run time exception

## Pessimistic approach

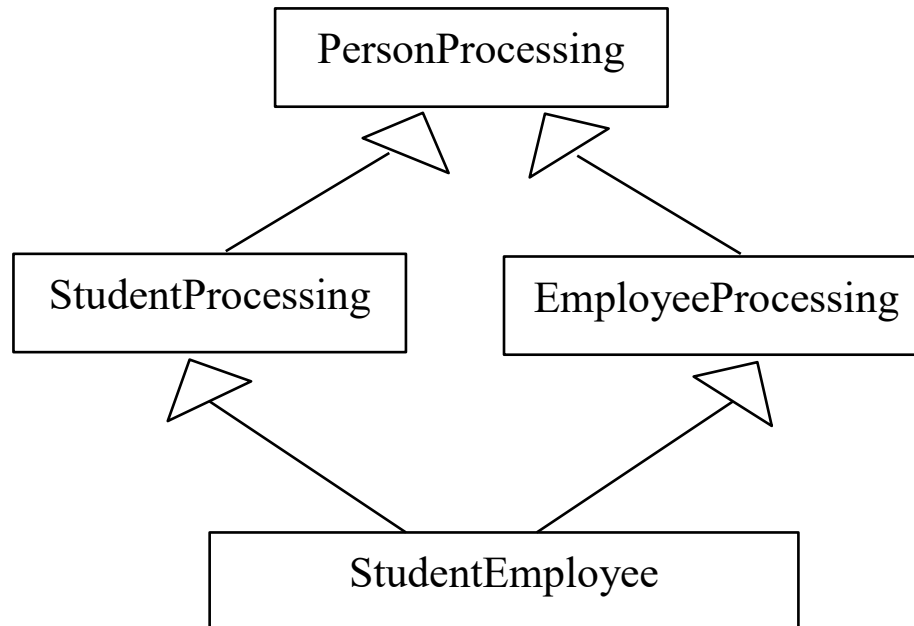
```
if(s2 instanceof Circle) {  
    Circle cref = (Circle) s2;  
}  
else {  
    // ... do something else  
}
```

## Optimistic approach

```
try {  
    // ...  
    Circle cref = (Circle) s2;  
    // ...  
} catch (ClassCastException e) {  
    // ... do something  
}
```

# Inheritance

- **Multiple inheritance issue**
  - C++ feature multiple inheritance
  - Java features simple inheritance
  - Diamond shape inheritance



# Inheritance

- Multiple inheritance issue with beyond Java 8 interfaces

```
public interface Interface1 {  
    default void defaultMethod() {  
        System.out.println("Default method in Interface1");  
    }  
}
```

```
public interface Interface2 {  
    default void defaultMethod() {  
        System.out.println("Default method in Interface2");  
    }  
}
```

```
public class ClassImplementingInterfaces implements Interface1, Interface2 {  
    com.compan.y.ClassImplementingInterfaces inherits unrelated defaults for defaultMethod() from types com.compan.y.Interface1 and com.compan.y.Interface2  
    public static void main(String[] args) {  
        ClassImplementingInterfaces aClass = new ClassImplementingInterfaces();  
    }  
}
```



```
public class ClassImplementingInterfaces implements Interface1, Interface2 {  
  
    @Override  
    public void defaultMethod() {  
        Interface1.super.defaultMethod();  
        Interface2.super.defaultMethod();  
    }  
  
    public static void main(String[] args) {  
        ClassImplementingInterfaces aClass = new ClassImplementingInterfaces();  
        aClass.defaultMethod();  
    }  
}
```

# Polymorphism

Aspect	Compile-time Polymorphism (Method Overloading)	Runtime Polymorphism (Method Overriding)
Binding	Happens at compile-time. <i>Early</i>	Happens at runtime. <i>Late</i>
Method resolution	Based on method signature (parameters).	Based on the actual object type.
Inheritance	Not required. Methods can be in the same class.	Requires inheritance and method overriding.
Example	Method overloading (same method name, different parameters).	Method overriding (same method name, different class implementations).

# Polymorphism

---

- **Method call binding**

- **Early binding** (i.e., static binding) – the binding is performed before the program is run by the compiler
- **Late binding** (i.e., dynamic binding, runtime binding) – the binding is performed at run time based on the type of the object
  - The method call mechanism finds out and calls the correct method body
  - Happens **polymorphically** unless the method is **static** or **final** (**private** methods are implicitly final)

```
public interface Comparable { public int compareTo(Object o); } ✓  
Comparable c = new Date();  
int comp = c.compareTo("Cluj"); ✗
```

# Polymorphism

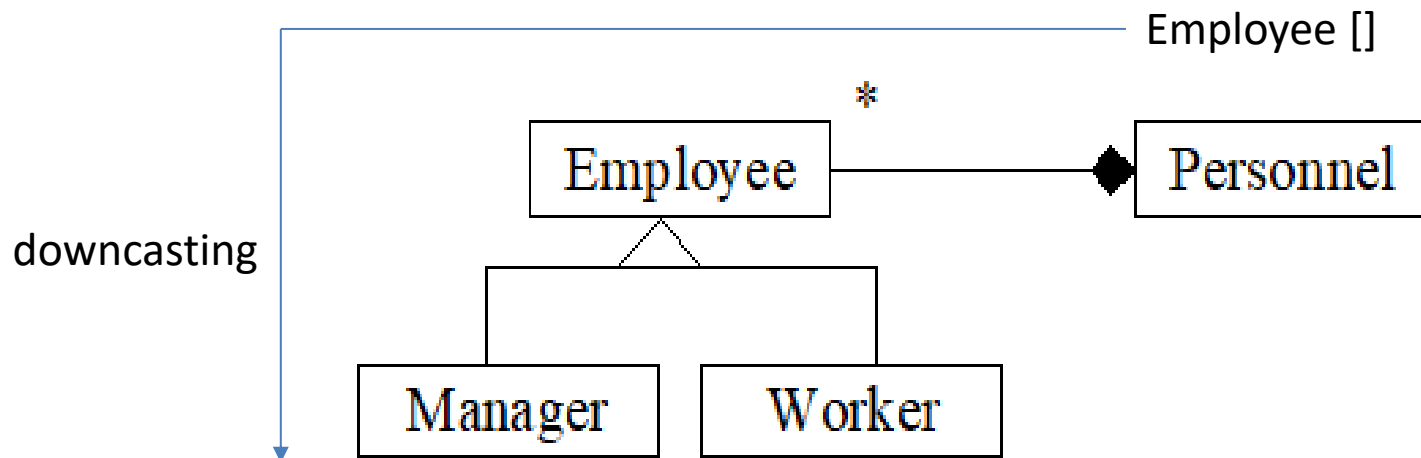
---

Sub-type	Description
PARAMETRIC	Use <b>generic types</b> that are replaced with concrete types when the code is used. The actual type is <b>determined at compile-time</b> , ensuring type safety.
OVERRIDING	One type can be substituted for another when there is an " <b>is-a</b> " <b>relationship</b> between them. <b>At run-time</b>
OVERLOADING	Using a function with the same name, but different parameters and different code. <b>Compile time</b>
COERCION	The <b>implicit conversion</b> of one data type to another. Automatically when converting from <b>smaller to larger types</b>

# Polymorphism

- **Advantages**

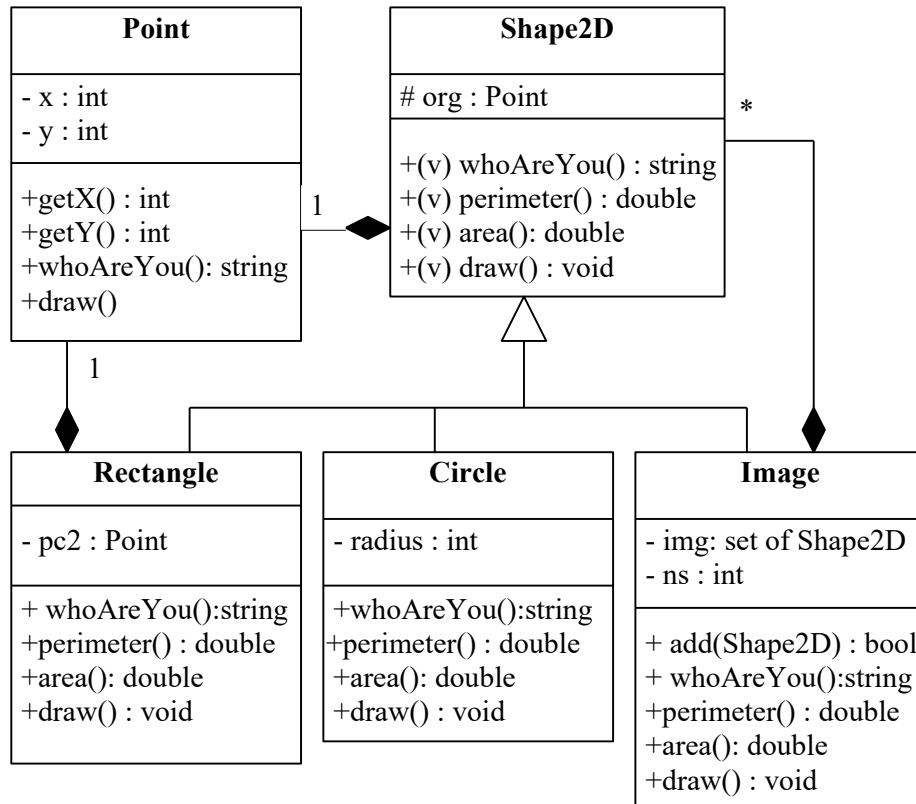
- Decouple what from how and promotes code reuse
- Allows improved code organization and readability
- Extensible programs that can be “grown”





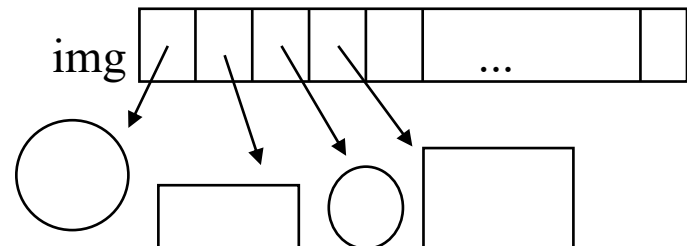
# Polymorphism

- **Example**



```
public class Image extends Shape2D {
    private Shape2D[] img = new Shape2D[MAXSIZE];
    private int ns; // effective number of shapes
    public void draw() {
        for(int i = 0; i < ns; i++) {
            // polymorphism - dynamic linking
            img[i].draw();
        }
    }

    public double area() {
        double totalArea = 0.0;
        for (int i = 0; i < ns; i++) {
            // polymorphism - dynamic linking
            totalArea += img[i].area();
        }
        return totalArea;
    }
}
```



# Polymorphism

- **Static (Compile Time) Polymorphism**

The *add* method to be invoked is determined at compile time based on the list of parameters

```
public class AddOperation {
    public static int add(int a, int b){
        return a + b;
    }

    public static int add(int a, int b, int c){
        return a + b + c;
    }

    public static void main(String[] args){
        int result1 = AddOperation.add(4, 5);
        int result2 = AddOperation.add(5, 9, 10);
    }
}
```

- **Dynamic (Runtime) Polymorphism**

The *whoAreYou* method is called through a reference of the *Shape2D* class (parent class) => the method to be executed is determined at runtime based on the type of the assigned object

```
public class Shape2D {
    ...
    public String whoAreYou() {return "SHAPE2D"; }
    ...
}

public class Circle extends Shape2D{
    ...
    public String whoAreYou() { return "CIRCLE"; }
    ...
}

...
Shape2D shape = new Circle();
shape.whoAreYou();
```

# Polymorphism

- **Overriding**

- Subclass method has the same elements with the method in the superclass: name, parameters, return type

## Overriding with dynamic binding algorithm

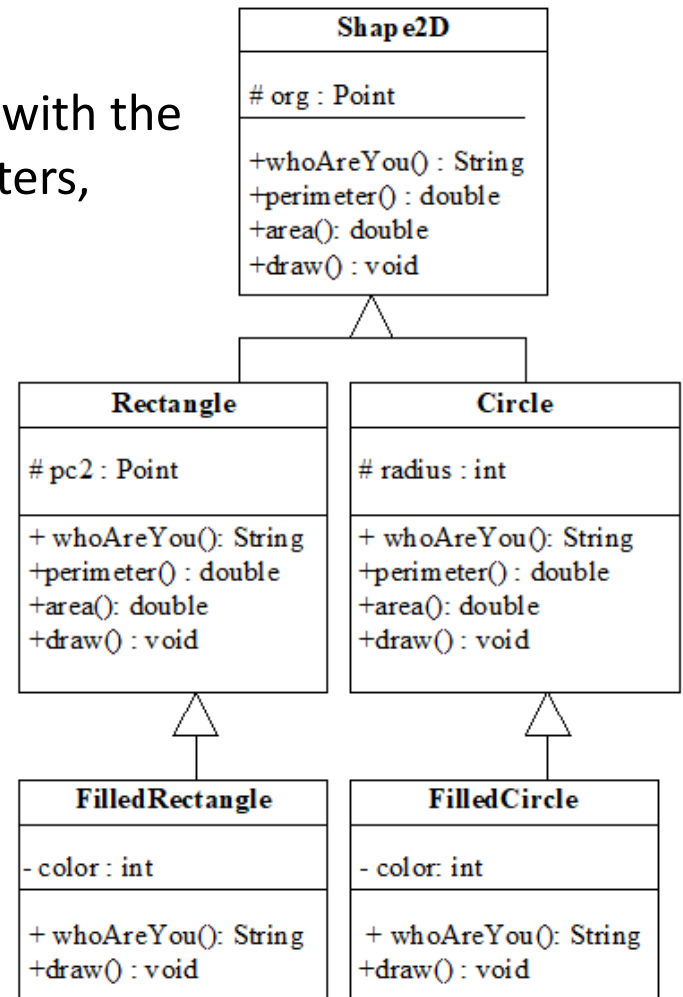
polymorphic method invocation: *obj.m(...)*

*crtClass* = the class of the object referenced *obj*

**repeat**

**if** *m()* is implemented by *crtClass* **then** invoke *m()*

**else** *crtClass* = the superclass of *crtClass*



# Polymorphism

- **Overloading**

- Allows methods of the same class to have the same name and different parameters
- If a Java base class has a method name overloaded several times, redefining that method name in the derived class will not hide any of the base-class versions (unlike C++)

```
public class AClass{  
    public int addNumbers(int n1, int n2){ return n1+n2; }  
  
    public int addNumbers(int n1, int n2, int n3){ return n1+n2+n3; }}
```

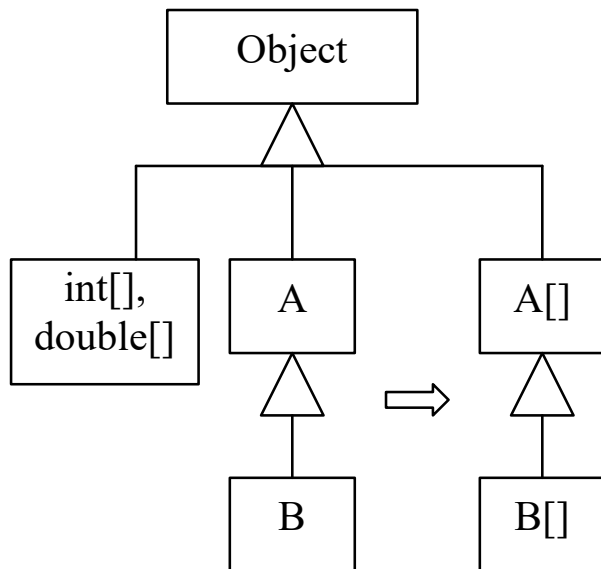
- Overloading on return values – **not possible**

```
public class AClass{  
    public int method1(){ ... }  
  
    public void method1(){ ... }  
}  
...  
method1(); //what method to call?
```

# Polymorphism

- **Arrays**

- **Rule 1:** All array types (i.e. `int[]`, `double[]`) are subtypes of `Object`
- **Rule 2:** If class or interface `B` is subtype of class or interface `A` then `B[]` is also subtype of `A[]`



Example of polymorphic assignments in which no explicit casting is necessary

Compile error!!!

Example of polymorphic assignments with explicit downcasting

```
Shape2D sa[];  
Circle ca[] = new Circle[10];  
// ...  
sa = ca;  
Shape2D s1 = sa[2];  
Circle c1 = ca[3];  
  
Circle c2 = sa[0];  
Circle c2 = (Circle)sa[0];
```

# Polymorphism

- Polymorphic assignments

```
class Shape2D { ... }  
class Circle extends Shape2D { ...}  
class Rectangle extends Shape2D { ... }
```

```
Shape2D s1, s2;
```

```
s1 = new Circle();  
s2 = new Rectangle();
```

```
Circle c1;
```

```
c1 = s1;
```

```
c1 = (Circle) s1;
```

Example of polymorphic assignments in which no explicit casting is necessary

**Compile error!!!:**

Even though actually *s1* holds a reference to a *Circle* instance, the declared type of *s1* is *Shape2D* and this is not subtype of left-hand side (i.e. *Circle*)

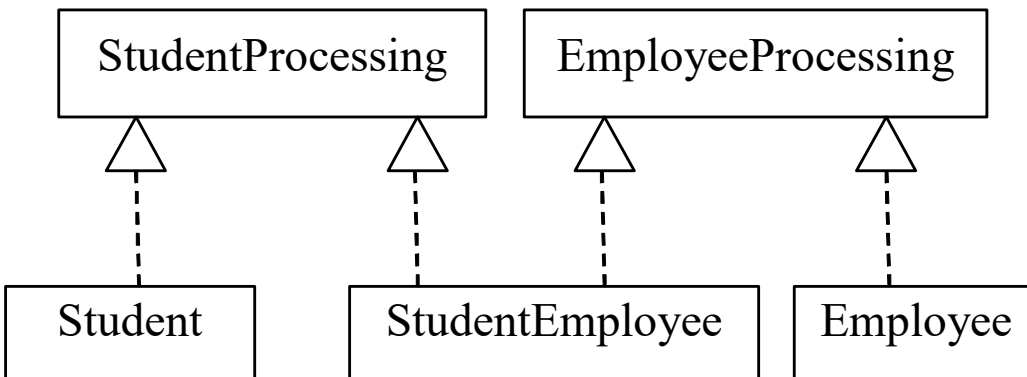
Type checking takes place at compile time

Example of polymorphic assignments with explicit cast

# Polymorphism

- **Interfaces and polymorphic classes**

- Interfaces can be used to define polymorphic classes
- A class can implement multiple interfaces => a class shows polymorphic behavior (assumes different roles in different contexts)



```
public interface StudentProcessing {
    public double calculateAvgMarks();
    public double getAvgMarks();
    // ... other methods
}

public interface EmployeeProcessing {
    public double calculateSalary();
    public double getSalary();
    // ... other methods
}
```

# Polymorphism

- Interfaces and polymorphic classes

## Instances of StudentEmployee seen as student

```
StudentProcessing[] students = new StudentProcessing[SIZE];
students[0] = new Student(...);
students[1] = new StudentEmployee(...);
// ...
for(int i = 0; i < students.length; i++) {
    ... students[i].calculateAvgMarks();
}
```

A student employee  
can play two  
different roles in two  
different contexts

## Instances of StudentEmployee seen as employees

```
EmployeeProcessing[] employees = new EmployeeProcessing[SIZE];
employees[0] = new Employee(...);
employees[1] = new StudentEmployee(...);
// ...
for(int i = 0; i < employees.length; i++) {
    ... employees[i].calculateSalary();
}
```



# Polymorphism

- Interfaces and polymorphic classes

```
public class Student implements
    StudentProcessing {
    protected double avgMarks;
    public double calculateAvgMarks() {
        // ... method implementation
    }
    public double getAvgMarks() {
        // ... method implementation
    }
}

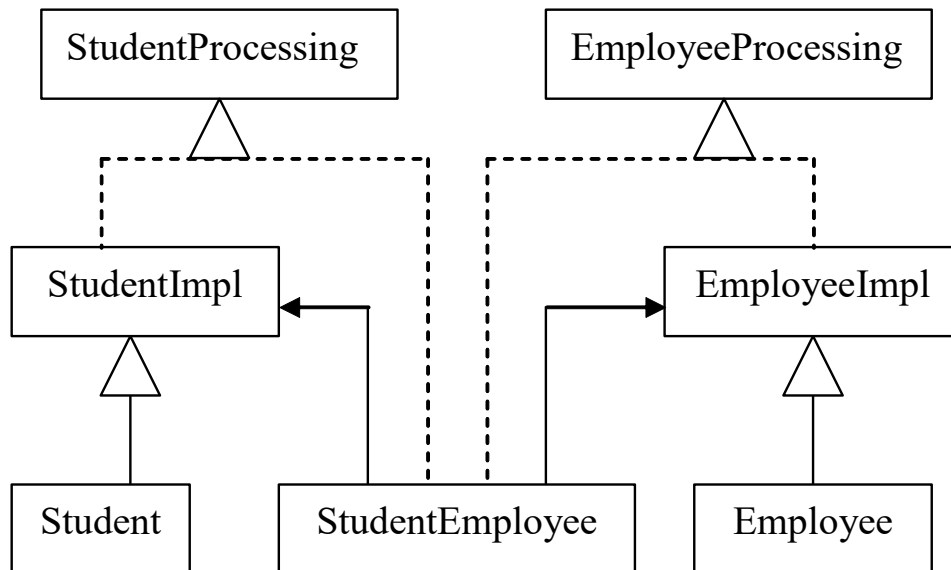
public class Employee implements
    EmployeeProcessing {
    protected double salary;
    public double calculateSalary() {
        // ... method implementation
    }
    public double getSalary() {
        // ... method implementation
    }
}
```

```
public class StudentEmployee implements
    StudentProcessing, EmployeeProcessing {
    protected double avgMarks;
    protected double salary;
    // ... other variables
    public double calculateAvgMarks() {
        // ... method implementation
    }
    public double getAvgMarks() {
        // ... method implementation
    }
    public double calculateSalary() {
        // ... method implementation
    }
    public double getSalary() {
        // ... method implementation
    }
}
```

**Sources of inconsistency: there is no common implementation of the methods that are inherited and specialized.**

# Polymorphism

- Interfaces and polymorphic classes
  - Inconsistency solution: simple inheritance + delegation technique



```
class Delegate {  
    // the "delegate"  
    void print()  
    {  
        System.out.println("The Delegate");  
    }  
}  
  
class Delegator {  
    // the "delegator"  
    Delegate p = new Delegate ();  
  
    // create the delegate  
    void print()  
    {  
        p.print(); // delegation  
    }  
}
```