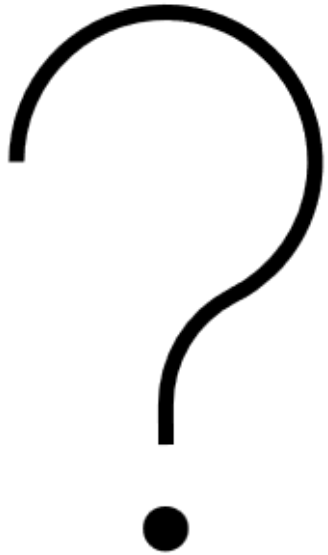
Lecture 4

OOP Inheritance

Inheritance

- What is inheritance?
- Why do we need inheritance?



Inheritance

- **Imagine**, in an app, you model two positions:
  - `Programmer` and `Manager`
- These positions have common properties:
  - *name, address, phone number*

`Programmer` class	**`Manager` class**

`public class Programmer {`
`  private String address;`
`  private String name;`
`  private String phone;`
`}`
	`public class Manager {`
	`  private String address;`
	`  private String name;`
	`  private String phone;`
	`}`

Inheritance

- Programmers and managers also have **different** properties.
  - Programmer: programming languages
  - Manager: project status reports

Programmer class

```
public class Programmer {
    private String address;
    private String name;
    private String phone;
    private String[] languages;
    void writeCode(){}
}
```

Manager class

```
public class Manager {
    private String address;
    private String name;
    private String phone;
    void reportProjectStatus(){}
}
```

Problem with the design

A solution to the problem

- Put common properties in a new position
  - A new class called ***parent*** class or ***super-class***
  - **`Programmer`** and **`Manager`** become ***sub-classes***

`Employee` class

```
public class Employee {
    private String address;
    private String name;
    private String phone;
}
```

`Programmer` class

```
public class Programmer
                extends Employee {
    private String[] languages;
    void writeCode(){}
}
```

`Manager` class

```
public class Manager
                extends Employee {
    void reportProjectStatus(){}
}
```

Inheritance

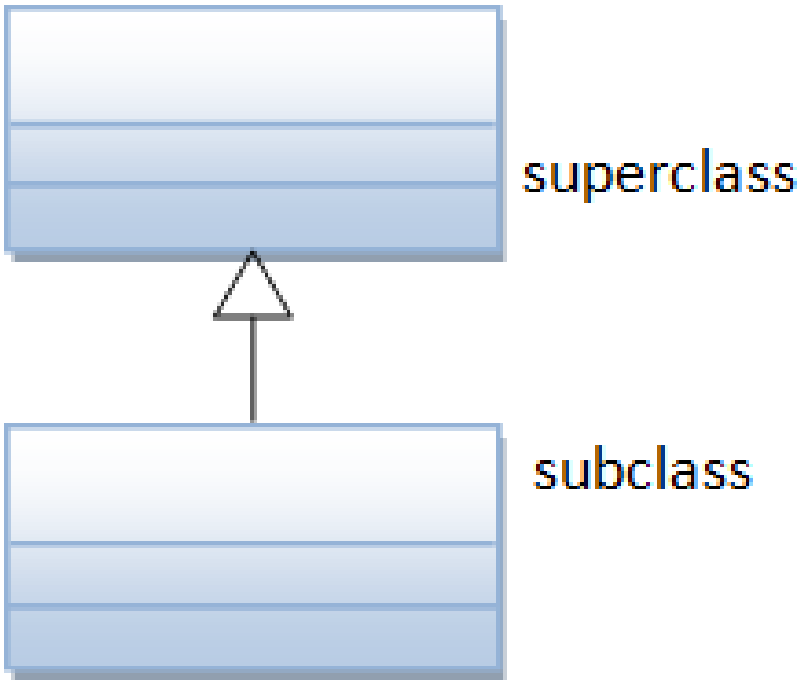
Benefits
Smaller class definition
Ease of modification to **common** properties and behavior
Extensibility
Enable to reuse code

Inheritance Hierarchy

- In OOP, classes are organized hierarchically to prevent duplication and reduce redundancy.
- Classes in Java exist within a hierarchical system, known as an inheritance tree.
- The top-level class is called the **superclass**, while the lower-level classes are called **subclasses**.
- In Java, a class can only have a single superclass (single inheritance).

Subclasses and Superclasses

- **Subclasses** inherit variables and methods from higher-level superclasses.
- **Subclasses** are also known as derived, child, or extended classes.
- **Superclasses** are referred to as base or parent classes.



How to create a subclass?

- Subclasses are created by extending an existing class with the `extends` keyword.

```
class Triangle extends TwoDShape {
    String style;

    double area() {
        return width * height / 2;
    }

    void showStyle() {
        System.out.println("Triangle is " + style);
    }
}
```

Triangle can refer to the members of `TwoDShape` as if they were declared in `Triangle`

The `TwoDShape` class

```
class TwoDShape {
    double width;
    double height;

    void showDim() {
        System.out.println("Width and height are " +
                width + " and " + height);
    }
}
```

Subclasses and Superclasses

- Placing common variables and methods in superclasses minimizes redundancy.
- Specialized variables and methods remain in subclasses.
- This approach enhances code organization and maintenance.
- Redundancy is reduced across subclasses, promoting code reuse.

Using subclasses

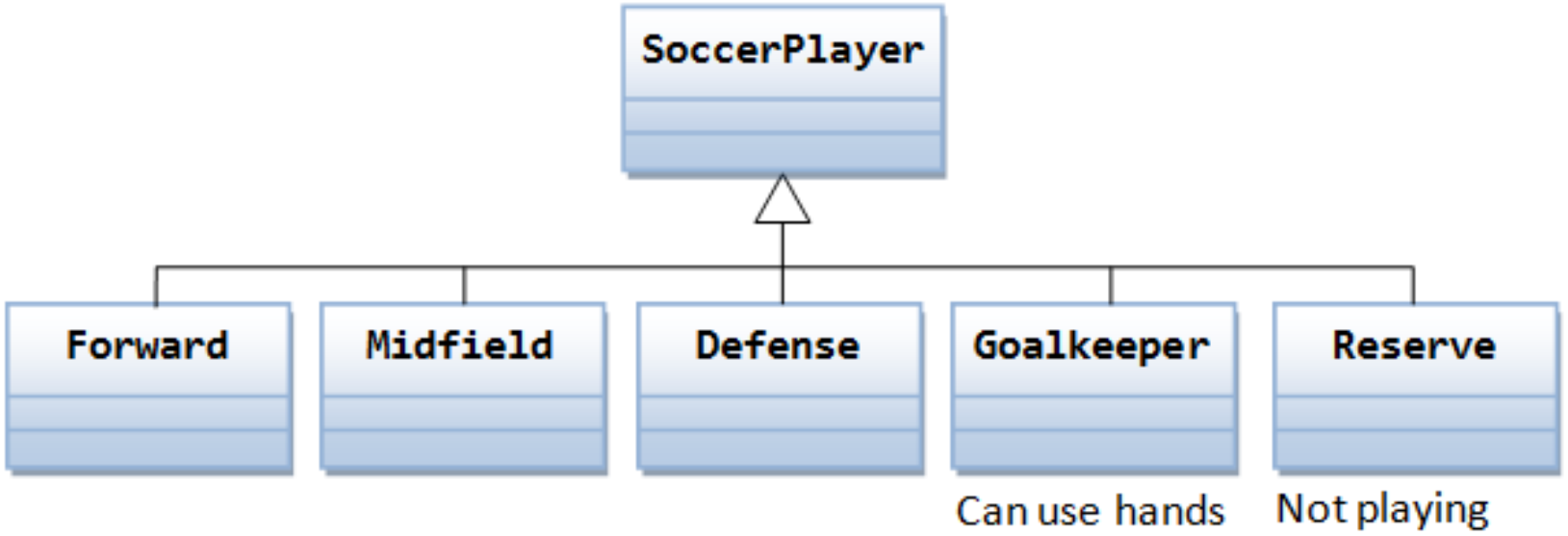
```
class Shapes {
    public static void main(String[] args) {
        Triangle t1 = new Triangle();
        t1.width = 4.0;
        t1.height = 4.0;
        t1.style = "filled";
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());
    }
}
```

All members of `Triangle` are available to `Triangle` objects, including those inherited from `TwoDShape`

Subclasses and Superclasses

- The **subclass** inherits variables and methods from its superclasses (including immediate parent and ancestors)
  - The **subclass** is permitted to possess the `public` or `protected` properties of the parent class.
  - The subclass is also allowed to possess the `{default}` properties of the parent class if both are in the same package.
  - The subclass cannot access the `private` members of the parent class.
- The subclass does not inherit the constructors of the parent class (but default constructor is inherited)

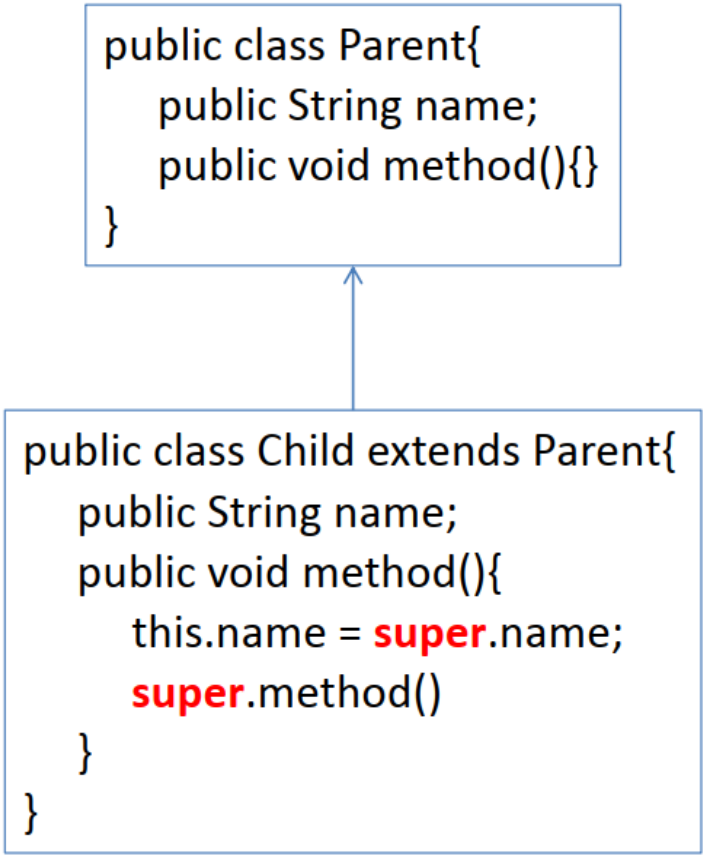
Example:




```
class Goalkeeper extends SoccerPlayer {......}
class MyApplet extends java.applet.Applet {.....}
class Cylinder extends Circle {......}
```

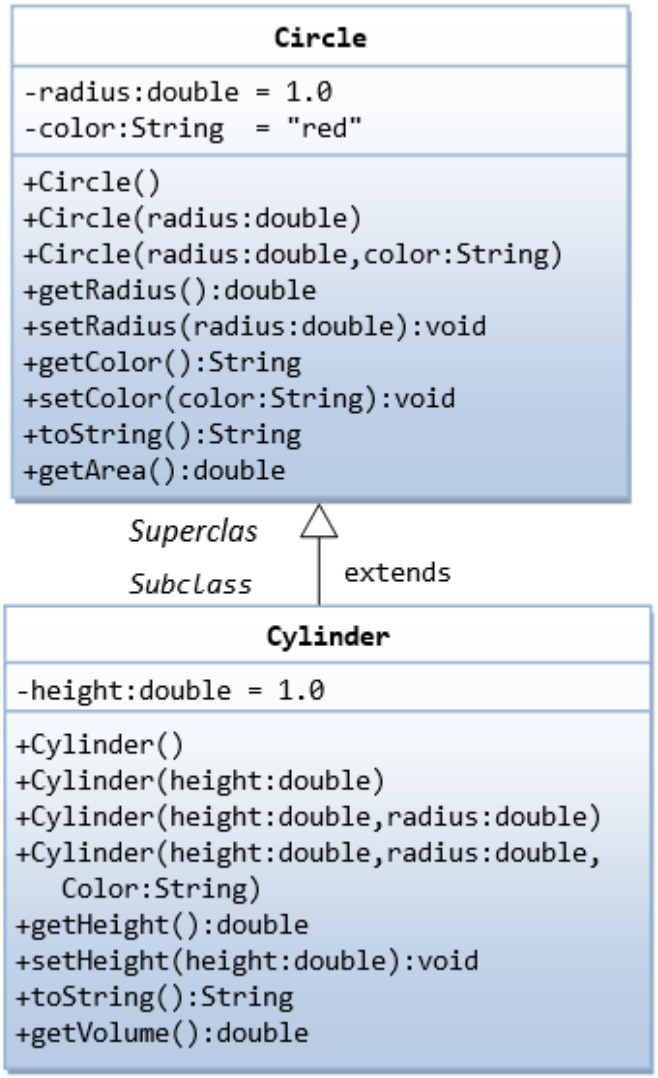
Keyword `super`

- Accessing the members of the parent class is done using the `super` keyword.
- `super` can be used to call the constructor of the parent class.



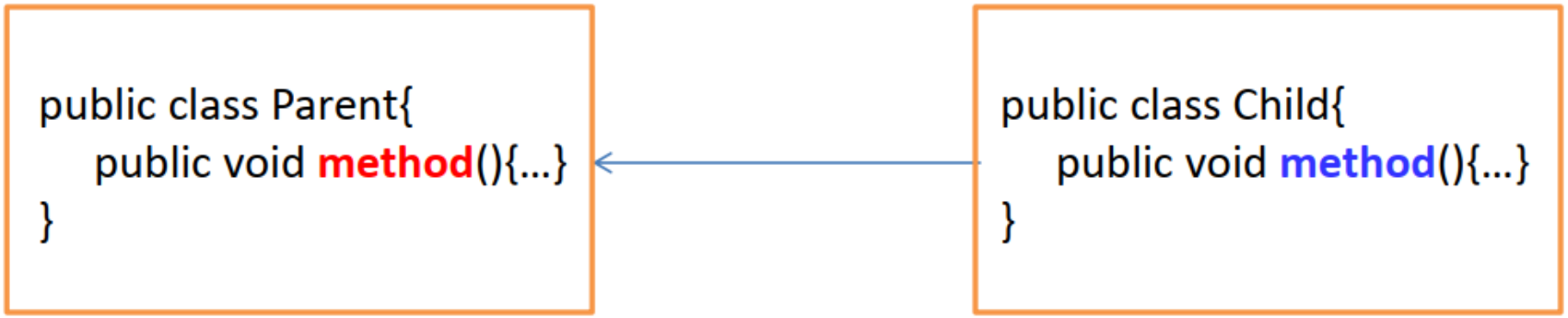
EG. 1: Deriving a Subclass from a Superclass

- Reuse of the `Circle` class is emphasized
- `Cylinder` inherits member variables (`radius`, `color`) and methods (`getRadius()`, `getArea()`, etc.) from `Circle`
- `Cylinder` introduces its own variable (`height`) and methods (`getHeight()`, `getVolume()`)
- Constructors for `Cylinder` are defined



Method Overriding & Variable Hiding

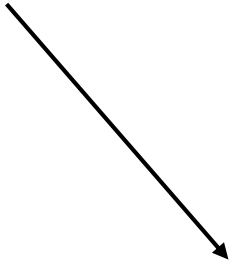
- Overriding occurs when both the subclass and the superclass have methods with the same syntax.



- Both the `Parent` and `Child` classes have a `method()` with the same signature, so the `method()` in `Child` will override the `method()` in `Parent`.

Method Overriding & Variable Hiding

```
Parent o = new Child();
o.method()
```



Even though the object has the type `Parent`, when `o.method()` is called, the `method()` of the `Child` class will execute because it overrides the `method()` of the `Parent` class.

Method Overriding & Variable Hiding

- Subclass overriding a method of the superclass will hide the superclass method.
- The purpose of overriding is to modify the method of the superclass in the subclass.
- Use the keyword `super` to access the overridden method of the superclass.
- The access modifier of the subclass method must be **at least as** public as the access modifier of the superclass method.

Example for Method Overriding

The inherited method `getArea()` in a `Circle` object computes the base area of the circle.

Suppose that we decide to override the `getArea()` to compute the surface area of the cylinder in the subclass `Cylinder`.

Example for Method Overriding

```
public class Cylinder extends Circle {
   ......
   // Override the getArea() method inherited from superclass Circle
   @Override
   public double getArea() {
      return 2*Math.PI*getRadius()*height + 2*super.getArea();
   }
   // Need to change the getVolume() as well
   public double getVolume() {
      return super.getArea()*height;   // use superclass' getArea()
   }
   // Override the inherited toString()
   @Override
   public String toString() {
      return "Cylinder[" + super.toString() + ",height=" + height + "]";
   }
}
```

Example for Method Overriding

- If `getArea()` is called from a `Circle` object, it computes the area of the circle.
- If `getArea()` is called from a `Cylinder` object, it computes the surface area of the cylinder using the overridden implementation.
- **Note that** you have to use public getter method `getRadius()` to retrieve the `radius` of the `Circle`, because `radius` is declared `private` and therefore not accessible to other classes, including the subclass `Cylinder`.

For example for Method Overriding

- But if you override the `getArea()` method in `Cylinder`, the `getVolume()` method (which equals to `getArea() * height`) no longer works. It is because the overridden `getArea()` will be used in `Cylinder`, which does not compute the base area.
- You can fix this problem by using `super.getArea()` to use the superclass' version of `getArea()`
- **Note that** `super.getArea()` can only be issued from the subclass definition, but not from a created instance, e.g. `c1.super.getArea()`, as it break the information hiding and encapsulation principle.

Single Inheritance

- Java does not support multiple inheritance (C++ does). Multiple inheritance permits a subclass to have more than one direct superclasses.
- This has a serious drawback if the superclasses have conflicting implementation for the same method.
- In Java, each subclass can have one and only one direct superclass, i.e., single inheritance. On the other hand, a superclass can have many subclasses.

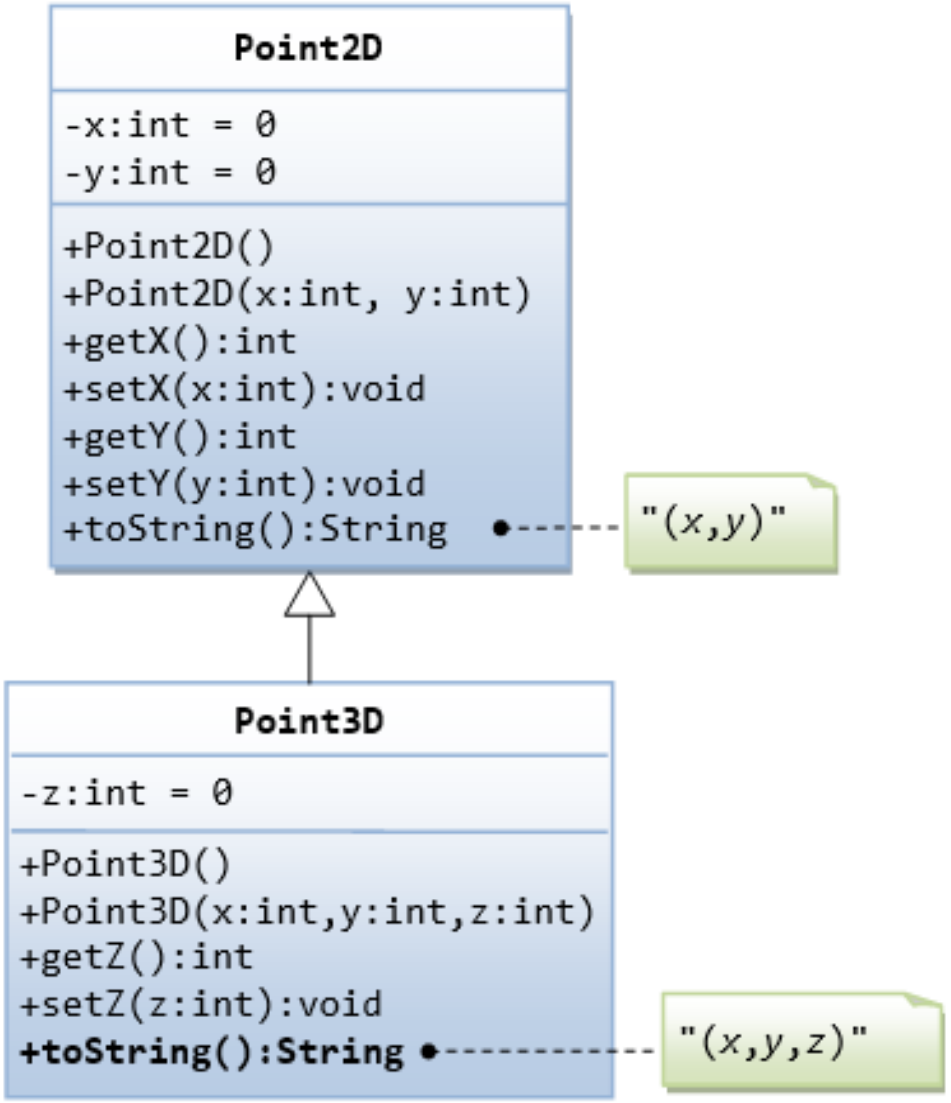
EG. 2: The `Point2D` and `Point3D` Classes

- Implement

  `Point2D.java`,

  `Point3D.java` and

  `TestPoint2DPoint3D.java`



Summary

- Inheritance Hierarchy
- Subclasses and Superclasses
- Method Overriding & Variable Hiding
- Single Inheritance