

# Programming Fundamentals

---

## Representing OOP Concepts in Java II Part I

Week 8 | Vishmi Embuldeniya

# Learning Outcomes

---

- Covers part of LO3 & LO4 for Module
- On completion of this lecture, students are expected to be able to:
  - Identify the need for object-oriented concepts in a program.
  - Build simple java applications using object-oriented concepts.
  - Make use of abstract classes in java programs.

# Object Oriented Programming

The object oriented paradigm is a programming methodology that promoted the efficient design and development of software systems using reusable components that can be quickly and safely assembled into larger systems.

The main aim of the object oriented programming is to implement real world concepts like,

Object → Real world entity  
Classes → Templates/Blueprint  
Inheritance → Parent child relation  
Polymorphism → many forms  
Abstraction → visibility control  
Encapsulation → Protect Data

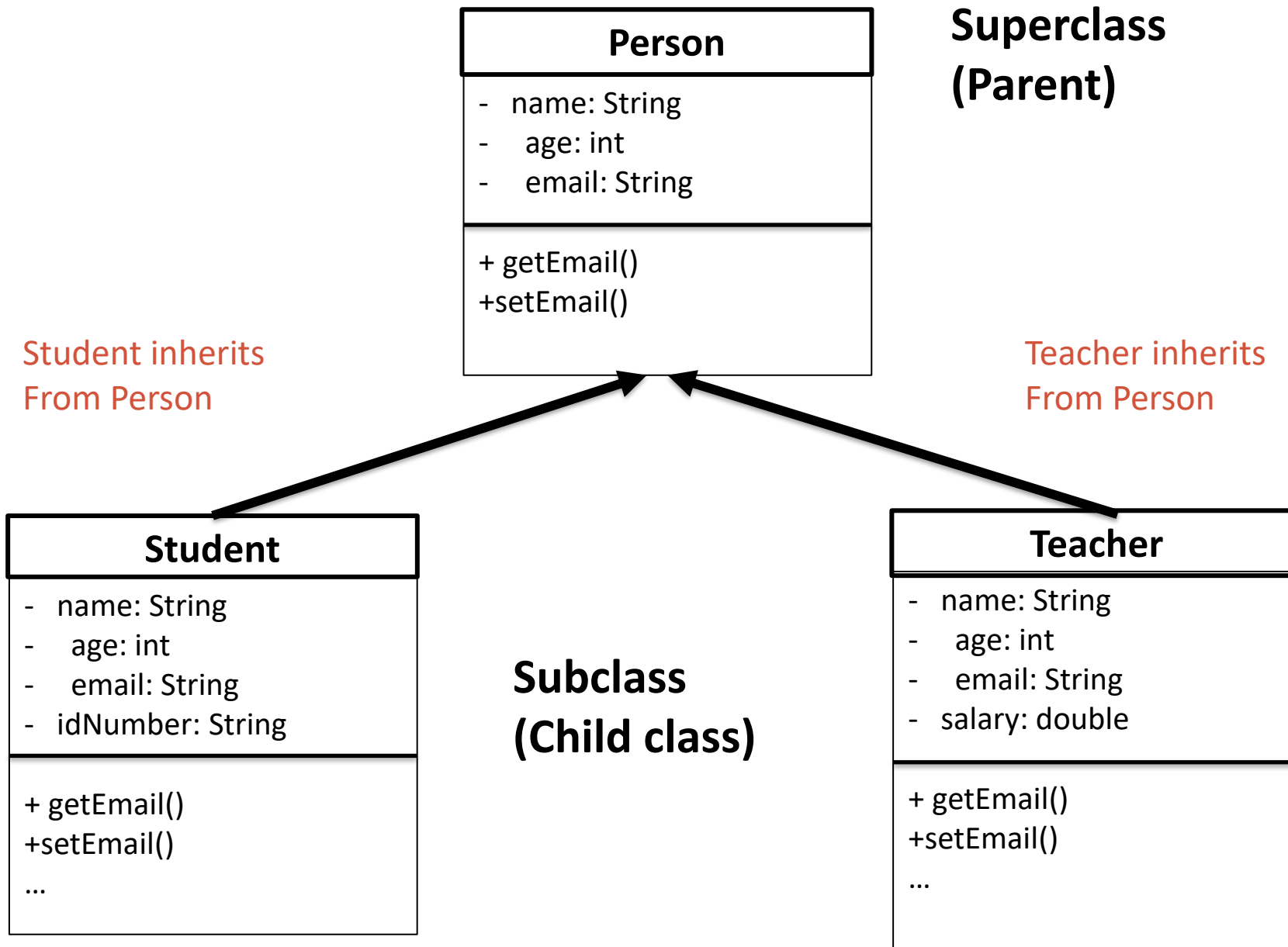
# Inheritance

Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.

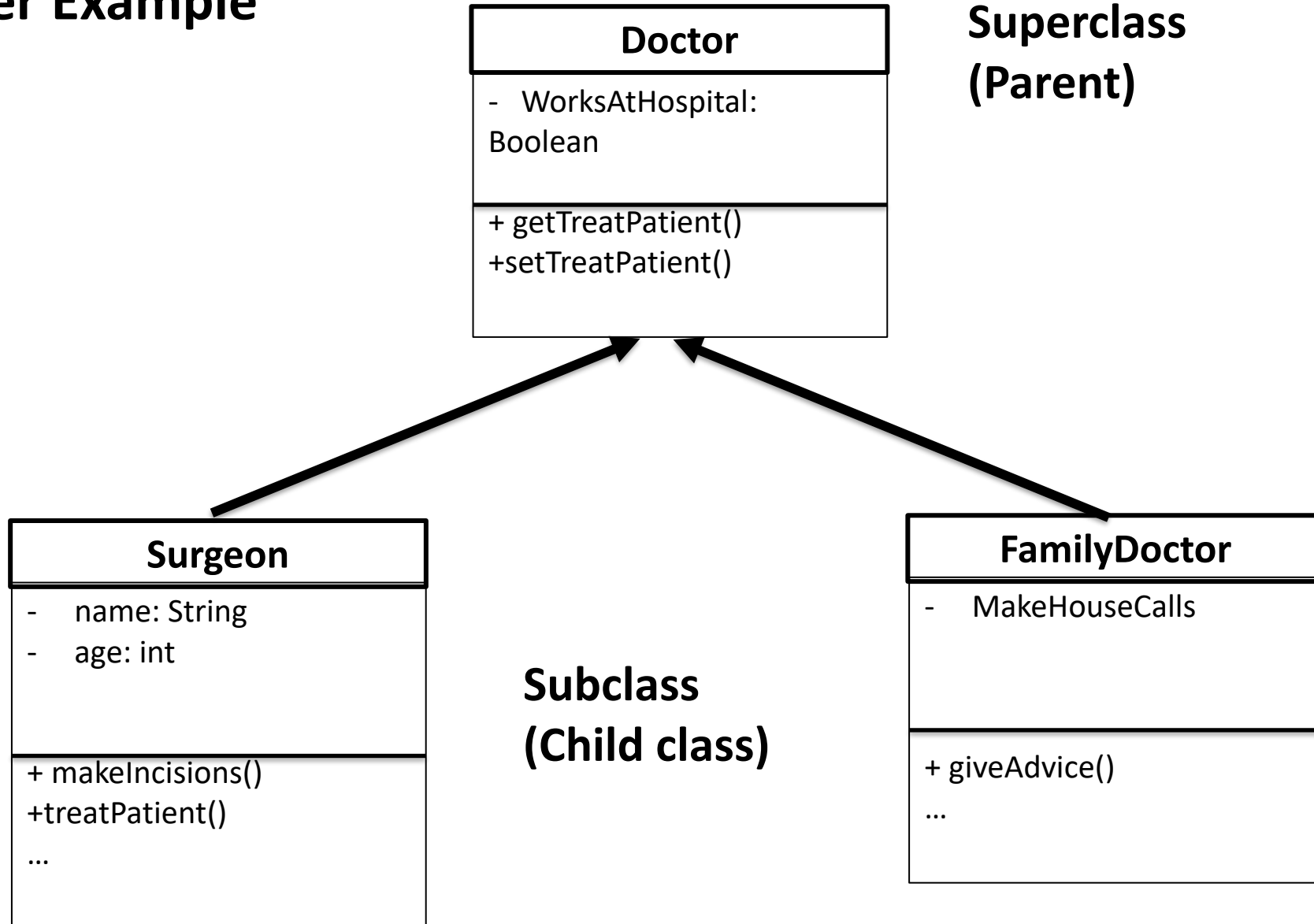
Mammals class inherits from the Animal class.

Terms used in Inheritance,

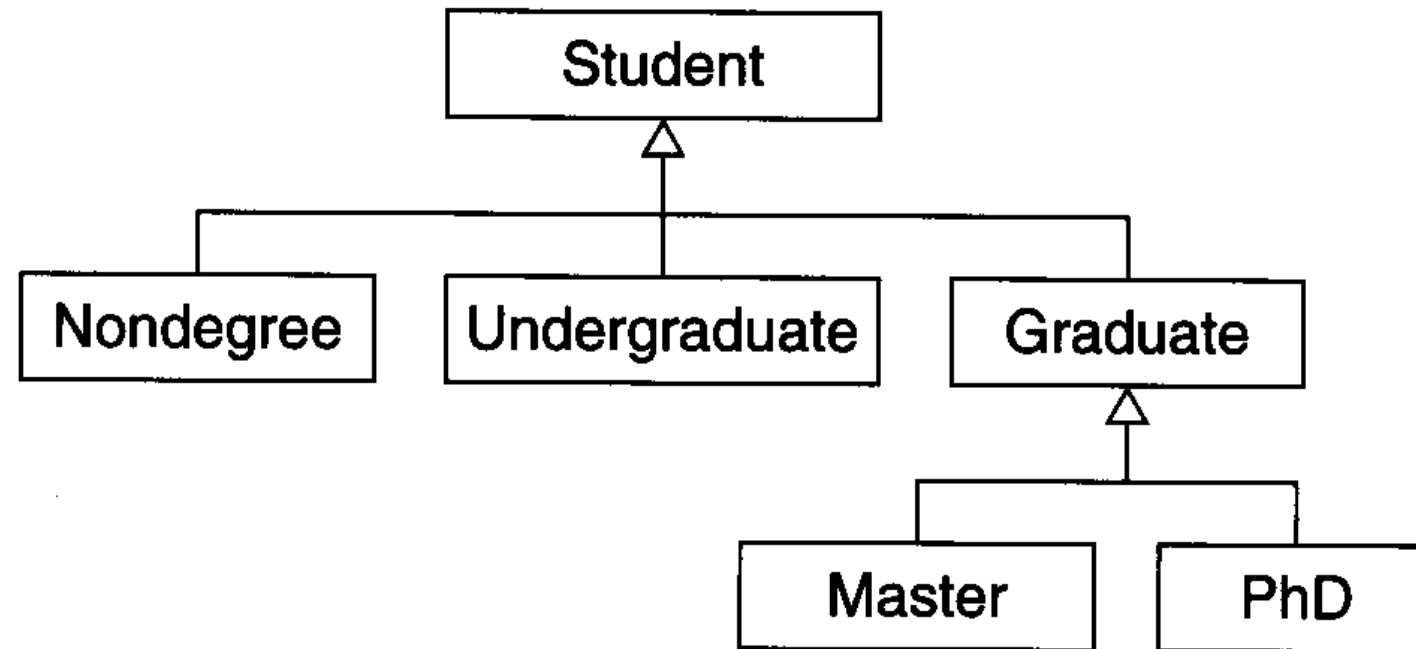
- Sub Class/Child Class/ Derived Class/Extended Class
- Super Class/Parent Class/Base Class



## Another Example



# Example



# Inheritance – “IS A” or “IS A KIND OF” Relationship

A Teacher **is a** Person

A Student **is a** Person

A Student **is a** Module

A Car **is a** Vehicle

A Motorbike **is a** Vehicle

A bus **is a** Car

A SavingAccount **is kind of** BankAccount

A Ferrari **is a** Car **is a** Vehicle

A Cat **is a** Mammal **is an** Animal



## Inheritance – Advantages

- Code Reusability
- Less redundancy
- Increased maintainability

# Class extension

Members of subclasses inherit variables and methods from their superclass(es)

➤ they **do not** inherit private variables & methods, or constructors

But they also can have their own special instance variables and methods that are not present in the superclass

```
public class Employee extends Person {  
... }
```

Employee is a **subclass** of Person

Person is the **superclass** of Employee

Employee **inherits** from Person

# Example – Class Person

```
public class Person {  
    private String name;  
    private String dob;  
  
    public Person(String n, String d) {  
        name = n;  
        dob = d;  
    }  
    Public Person (String n){  
        name = n;  
    }  
    public void setName(String newName) {  
        name = newName;  
    }  
    public String getName() {  
        return name;  
    }  
    public String getDoB() {  
        return dob;  
    }  
}
```

## Example – Class Employee

```
public class Employee extends Person {  
    private double salary;  
  
    public Employee(what goes here?) {  
        // What to write here?  
    }  
    public double getSalary() {  
        return salary;  
    }  
    public void setSalary(double newSalary) {  
        salary = newSalary;  
    }  
}
```

# Exercise 1

**Remember:** Employees are also Persons

What methods can we call on Employee objects?

- setSalary
- getSalary
- setName (inherited)
- getName (inherited)
- getDoB (inherited)

If **emp** is an Employee object:

```
String n = emp.getName();  
emp.setName("John Smith");  
emp.setSalary(25000.0);
```

## Exercise 2

if **p** is a **Person** class object, we can have:

```
p.setName("Dr Who");  
String n = p.getName();
```

Can we have `p.setSalary(25000);` ?

**NO! Because the method is declared in the subclass!**

# Overview so far

Employees are also Persons, so they inherit methods from Persons

- Java looks for a method first in the class to which the calling object belongs
- If it does not find it there, Java looks in the class's superclass, ...

Persons are not always Employees, so Persons will not have access to the methods that are defined only within Employees.

what about variables?

# Inheritance & instance variables

An instance of a subclass **stores all** the instance variables of the superclass (even private ones), plus all the instance variables defined in the subclass.

**Be careful** though, **private instance variables** of the superclass are **NOT INHERITED**



# Instance Variable

**Problem:** superclass variables are likely to be private

- Why private?
- Why is this a problem?

# Subclass Constructor

A subclass does not inherit constructors from the superclass.

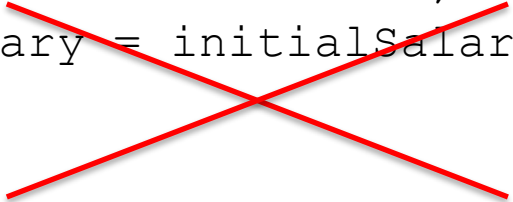
The subclass constructor will need to initialise instance variables that belong to the subclass and to the superclass (see previous slide)

# Exercise: which is the problem?

```

public class Employee extends Person {
    ...
    public Employee(String initialName, double initialSalary) {
        name = initialName;
        salary = initialSalary; }
    ...
}

```



# Solution: Super keyword

We invoke superclass constructors by using the **super** keyword

```
public Employee(String initialName, double initialSalary)
{
    super(initialName);
    salary = initialSalary;
}
```

```
Employee e = new Employee("Ted White", 25000.0);
```

# Usage of Java super Keyword

- 1.super can be used to refer immediate parent class instance variable.
- 2.super can be used to invoke immediate parent class method.
- 3.super() can be used to invoke immediate parent class constructor.

# A new access modifier: protected

For making access to methods and variables easier between classes in an inheritance relationship, the **protected** access modifier is available

**private**: can be accessed only in same class

**protected**: can be accessed in the same class, or in a subclass, or in the same package

**public**: can be accessed in any class

# Access modifier: protected

Protected variables and methods can be accessed by subclasses, subclasses of subclasses, etc.

## **protected vs. private**

- declaring variables as protected exposes them to all subclasses
- best to declare variables as private (even in inheritance relationships) and write getter and setter methods to provide access to variables

# Polymorphism

'polymorphism' from the ancient greek poly (many) morph (shapes)

In OOP, it describes the capability to use the “same code” to process objects of various types and classes, as long they have a common super class

There are two kinds of polymorphism:

- Overloading

- Two or more methods with different signatures in the same class

- Overriding

- Replacing an inherited method with another having the same signature



# Method Overloading

1. Number Of Parameters
2. Difference in data type of arguments
3. Sequence of data type of arguments

# Number Of Parameters

```
class Overloading_Example1 {  
    public void display(char c) {  
        System.out.println(c);  
    }  
  
    public void display(char c, int num) {  
        System.out.println(c + " " + num);  
    }  
}  
  
public class Sample1 {  
    public static void main(String args[]) {  
        Overloading_Example1 myobj = new Overloading_Example1();  
        myobj.display('a');  
        myobj.display('a', 50);  
    }  
}
```

# Difference in data type of arguments

```
class Overloading_Example2 {  
    public void display(char c) {  
        System.out.println(c);  
    }  
    public void display(int c) {  
        System.out.println(c);  
    }  
}  
  
public class Sample2 {  
    public static void main(String args[]) {  
        Overloading_Example2 myobj = new Overloading_Example2();  
        myobj.display('a');  
        myobj.display(50);  
    }  
}
```

# Sequence of data type of arguments

```
class Overloading_Example3 {  
    public void display(char c, int num) {  
        System.out.println("In the first definition of method display");  
    }  
  
    public void display(int num, char c) {  
        System.out.println("In the second definition of method display");  
    }  
}  
  
public class Sample3 {  
    public static void main(String[] args) {  
        Overloading_Example3 myobj = new Overloading_Example3();  
        myobj.display('a', 5);  
        myobj.display(50, 'a');  
    }  
}
```

# Substitution Principle

- In order to allow polymorphism, Java introduces the **Substitution Principle**, defined as:

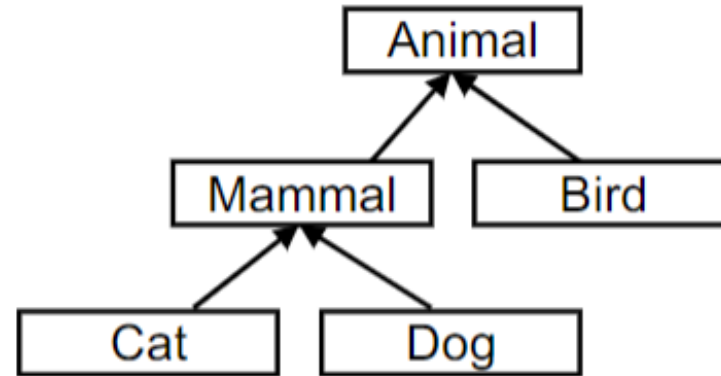
**An instance of a subclass can take the place of an instance of any of its superclasses**

Vehicle v = new Car();

Vehicle v = new Bicycle();

- Variables holding object types are polymorphic variables - they can hold objects of different acceptable types
- acceptable types: the declared type, or any subtype of the declared type

## Exercise: Which is correct?



- 1) `Animal myBird = new Bird();`
- 2) `Mammal felix = new Cat();`
- 3) `myBird tweetie = new Bird();` NO
- 4) `Dog snoopy = new Mammal();` NO
- 5) `Bird littleBird = new Animal();` NO
- 6) `Mammal m = new felix;` NO
- 7) `Animal snoopy= new Dog();`

# Method polymorphism

A Java version of the “algorithm” for operating a vehicle:

```
v.start();  
v.releaseBreak();  
v.accelerate();  
v.applyBrake();  
v.stop();
```

- It does not matter what exactly v is, as long as it is Vehicle or any of its subtypes
- This algorithm is polymorphic, it works for a variety of vehicle types, not only for a single type

# Dynamic method binding

- How can `v.start()` work if the compiler at compile time does not know what type `v` refers to?
- We do not really know which version of `start()` is being called
- `v` will have to be tested during program execution each time it calls an instance method
- This process is known as dynamic binding
  - ∅ the exact method called will not be known until the program is actually run



# Dynamic vs Static binding

- static binding - what method to call is resolved at compile time (e.g. overloaded methods)
- dynamic binding - what method to call is resolved at run time (most overridden methods)

# Dynamic binding

If different objects are assigned to `v` during execution, different versions of `start()` may be called:

```
v = new Car();  
v.start(); // Calls start method in Car class
```

```
v = new Bicycle();  
v.start(); //Calls start method in Bicycle class
```

# Method Overriding

Method overriding is a concept in object-oriented programming where a subclass provides a specific implementation for a method that exists in its superclass. The method in the subclass has the same name, return type, and parameters as the method in the superclass.

# Rules...

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.
- A subclass within the same package as the superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- Constructors cannot be overridden

# Example

```
class Animal {  
    public void move() {  
        System.out.println("Animals are cute");  
    }  
}
```

```
class Cat extends Animal {  
    public void move() {  
        System.out.println("Cats can run");  
    }  
}
```

```
public class TestCat {  
    public static void main(String[] args) {  
        Animal animal = new Animal();  
        Animal cat = new Cat();  
  
        animal.move();  
        cat.move();  
    }  
}
```

---

# Thank you