Lecture 7: Inheritance and Polymorphism

CSC 1214: Object-Oriented Programming

Outline

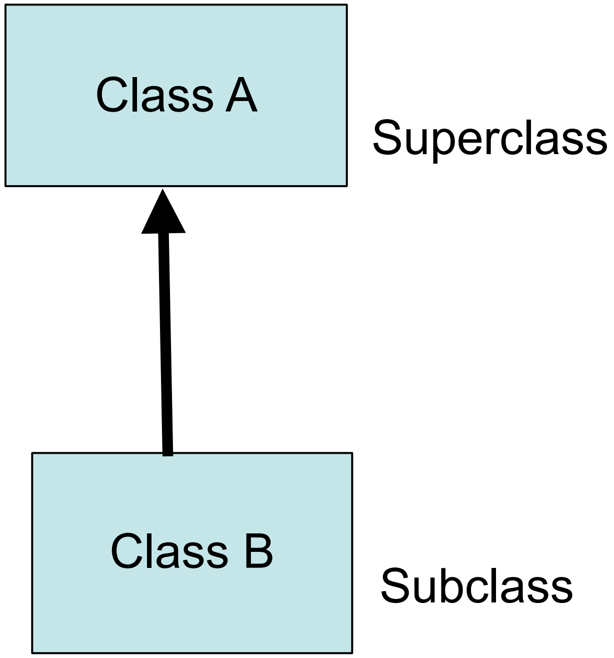
- Inheritance and **protected** visibility modifier
- Method overriding
- Method overloading
- Polymorphism

Outline

- Inheritance and **protected** visibility modifier
- Method overriding
- Method overloading
- Polymorphism

Inheritance: Introduction

- In object-oriented programming, **inheritance** allows one to derive a new class from an existing one
- The existing class is called the **parent class**, or **superclass**, or **base class**
- The derived class is called the **child class** or **subclass**.
- The child class inherits the methods and data defined for the parent class

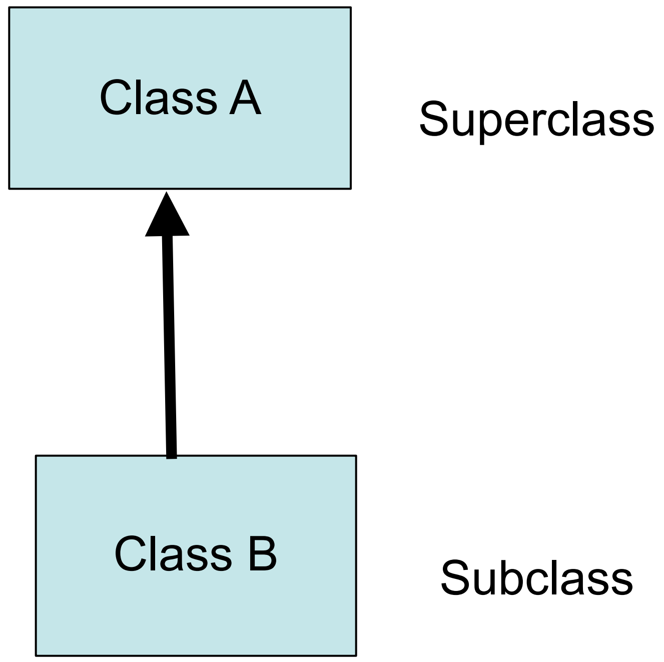


Benefits of Inheritance

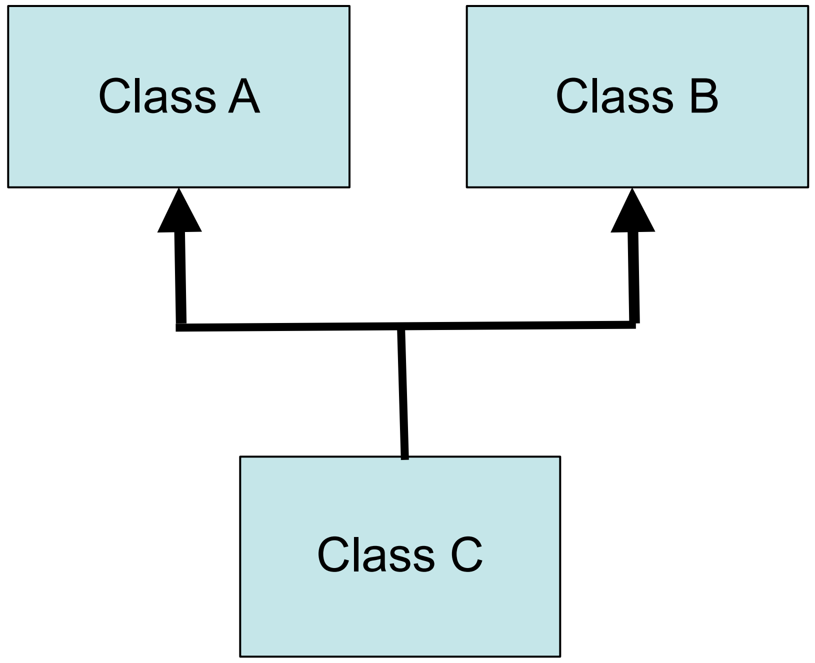
- What are the benefits of using inheritance?
  - **Reusability**: Inheritance increases the ability to reuse classes. Software can be extended by reusing previously defined classes and adding new methods to the subclasses.
  - **Clarity**: Inheritance avoids duplication and reduces redundancy
  - …*read more about other benefits of inheritance*

Different Forms of Inheritance

Single Inheritance

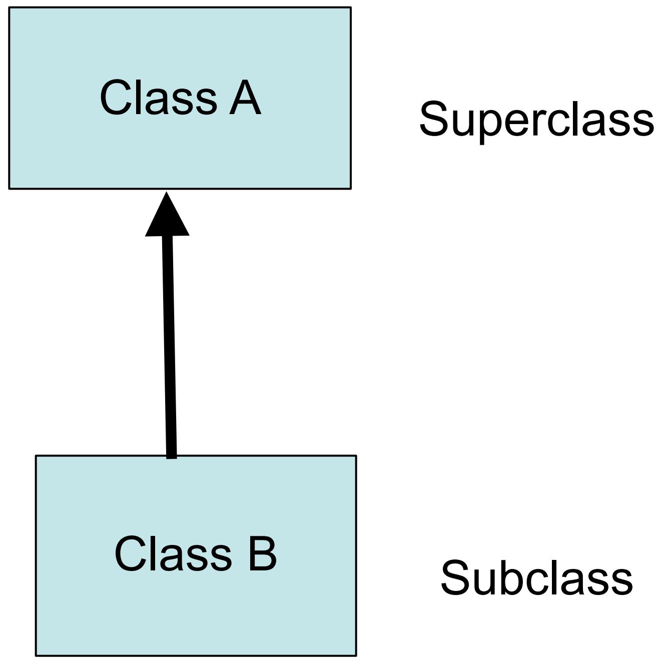


Multiple Inheritance

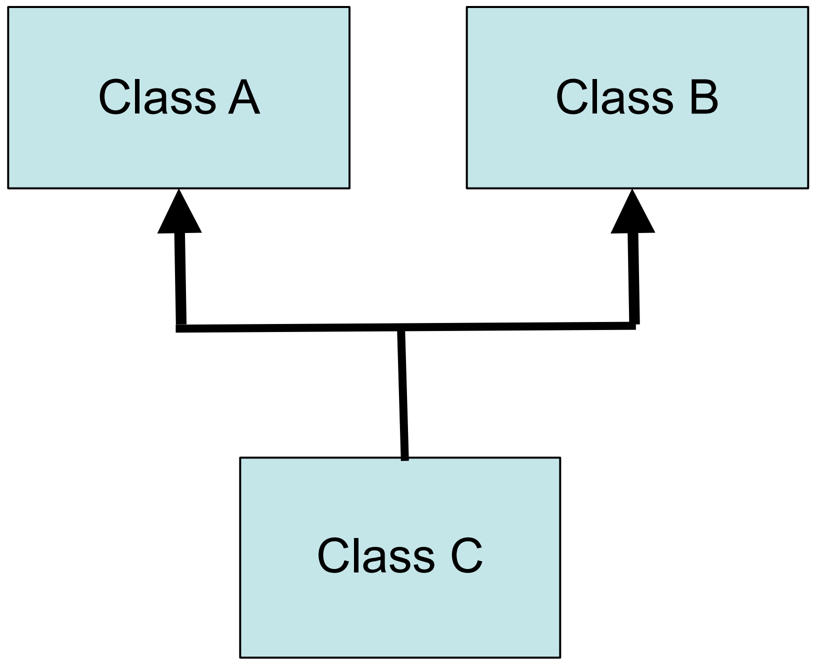


Different Forms of Inheritance

Single Inheritance



Multiple Inheritance



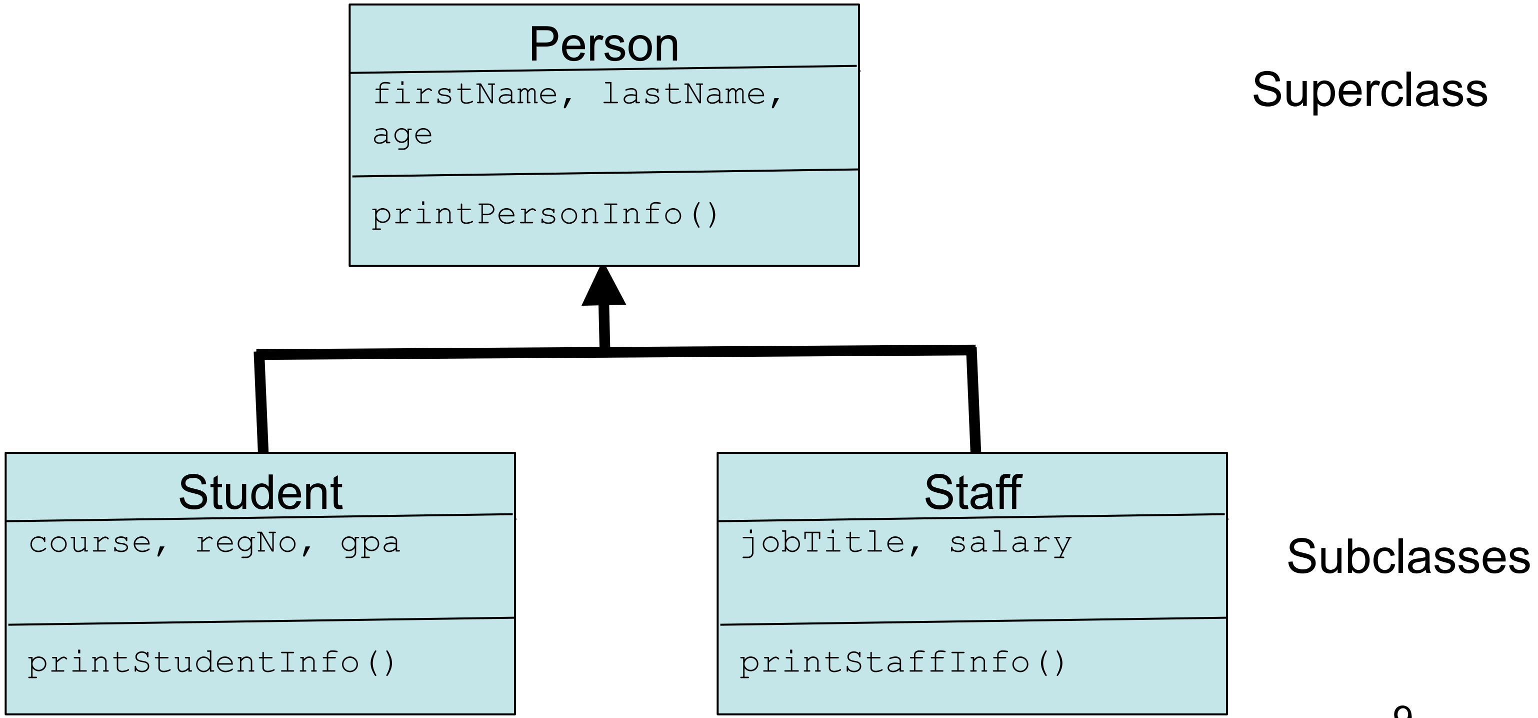
➡Java does not support multiple inheritance. OO languages that support multiple inheritance include C++, Python, Common Lisp

Inheritance: Examples

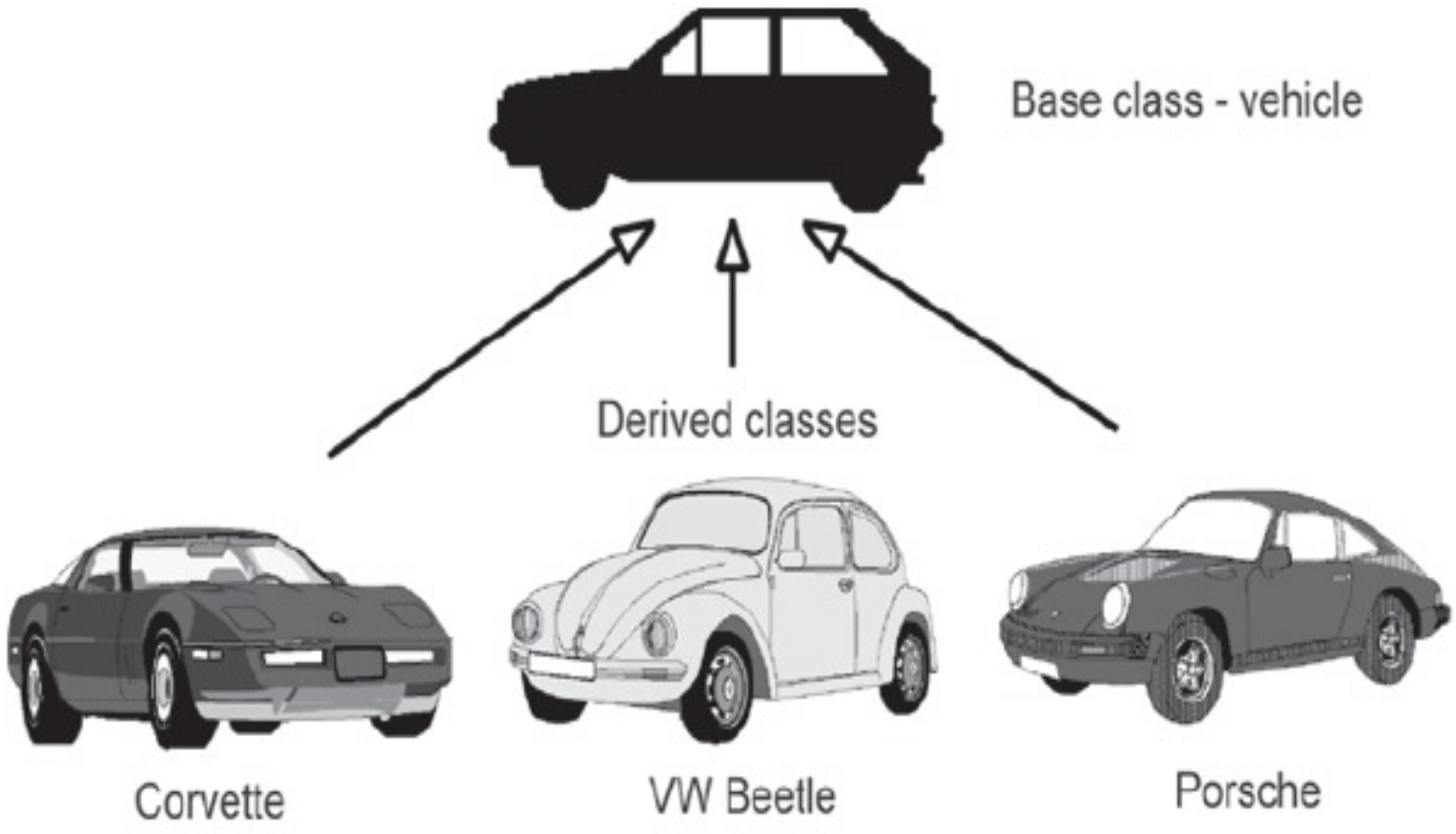
- Suppose that you are asked to develop a University Management System using an OO language like Java. You will need classes to represent students, staff members
- Different objects often have a certain amount in common with each other. Students, and staff members, for example, all share the characteristics of a person (first name, last name, age, …). Yet each also has additional features that make them different: a student has a course and a reg No., a staff member has salary, etc.

Inheritance: Examples

- Inheritance is useful because you can create a superclass that contains variables and methods that will be used by a number of different subclasses. This saves you from having to rewrite common variables and methods in each different class.



Inheritance: Examples



Inheritance in Java: Deriving Subclasses

- In Java, we use the reserved keyword **`extends`** to establish an inheritance relationship between classes

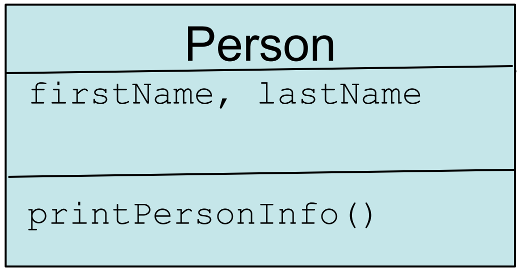
Subclass

```
class Student extends Person
{
    // class contents
}
```

Superclass

This gives `Student` all the same fields and methods as `Person`

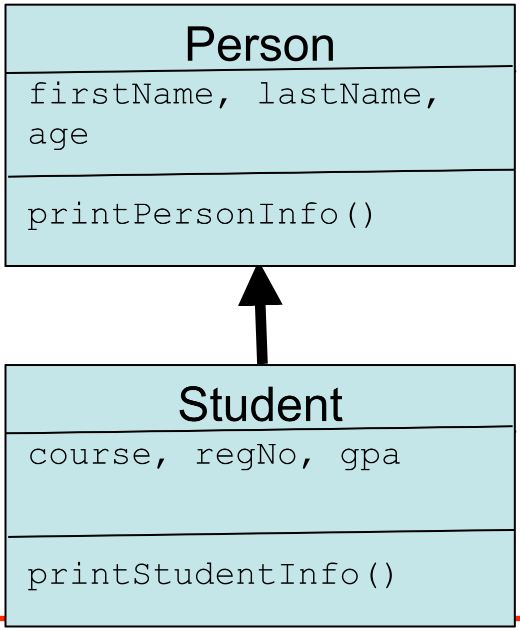
Person Class: Superclass



```
class Person {
 protected String firstName;
 protected String lastName;

 public Person(String fName, String lName){
   this.firstName = fName;
   this.lastName = lName;
 }

 public void printPersonInfo (){
   System.out.println("Full Name.: "+ firstName+" "+lastName);
 }
}
```

Student Class: Subclass

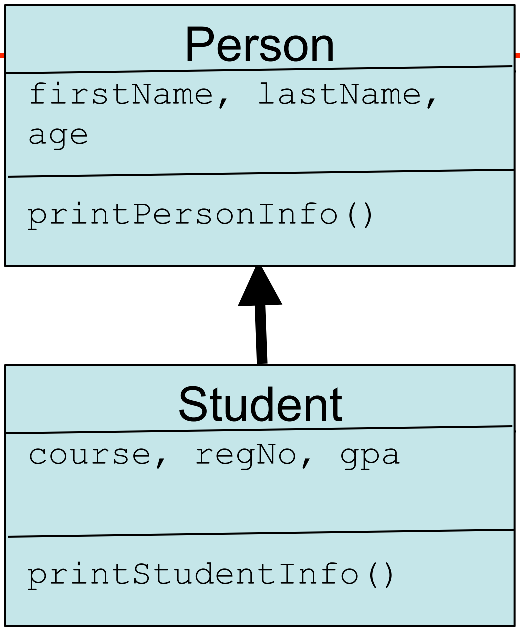
```
class Student extends Person{
 private String course;
 private String regNo;
 private double gpa;

 public Student(String fName,String lName,String course,String regNo,double gpa){
   super(fName, lName);
   this.course = course;
   this.regNo = regNo;
   this.gpa = gpa;
 }

 public void printStudentInfo (){
   System.out.println("Reg No.: "+ regNo);
   System.out.println("Course: "+ course);
   System.out.println("GPA: "+ gpa);
 }
}
```



Driver Class

```
class InheritanceDemo1{
 public static void main(String args[]){
 Student john = new Student("John","Okot", "CSC","12/U/002", 4.60);
   john.printPersonInfo();
   john.printStudentInfo();
  }
}
```

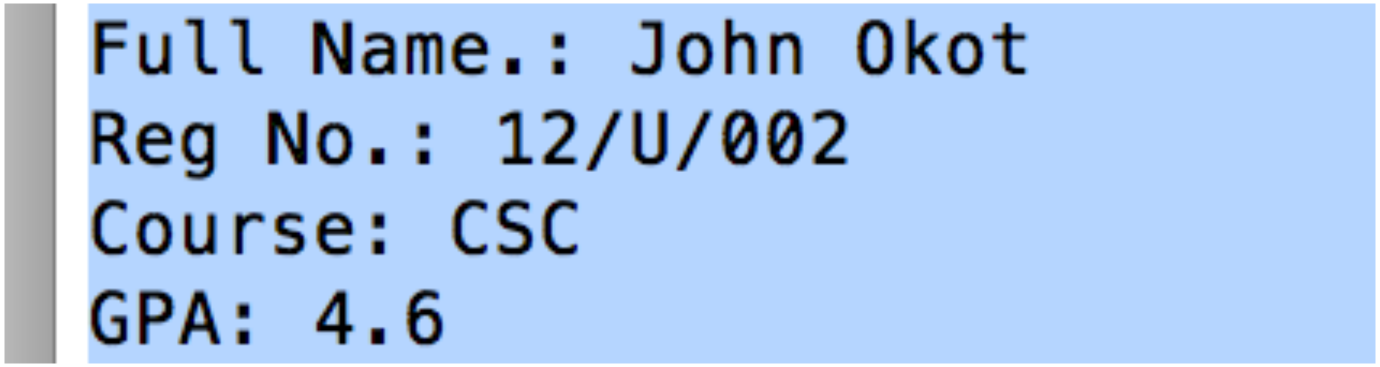
Student class inherits all the methods of Person class. Hence `john.printPersonInfo()` results in the invocation of the `printPersonInfo()` of the Person class



Driver Class

```
class InheritanceDemo1{
 public static void main(String args[]){
 Student john = new Student("John","Okot", "CSC","12/U/002", 4.60);
 john.printPersonInfo();
 john.printStudentInfo();
  }
}
```

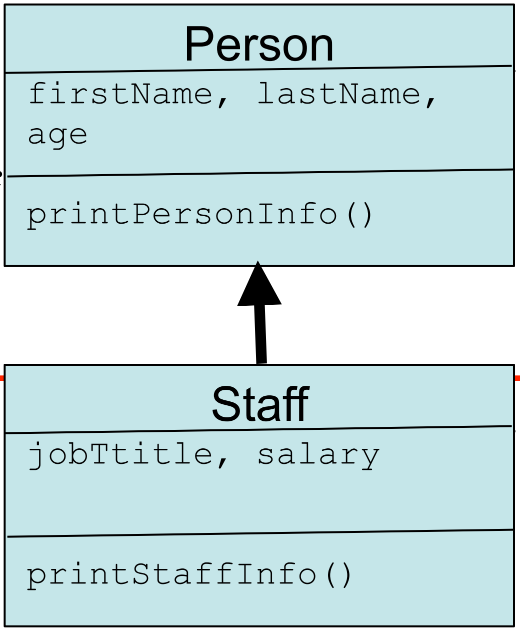
Output



```
Full Name.: John Okot
Reg No.: 12/U/002
Course: CSC
GPA: 4.6
```

Staff Class: Subclass

```
class Staff extends Person{
 private String jobTitle;
 private double salary;

 public Staff(String fName, String lName, String jobTitle, double salary
  super(fName, lName);
  this.jobTitle = jobTitle;
  this.salary = salary;
 }
 public void printStaffInfo (){
  System.out.println("Job title: "+ jobTitle);
  System.out.println("Salary: "+ salary);
 }
}
```



Driver Class

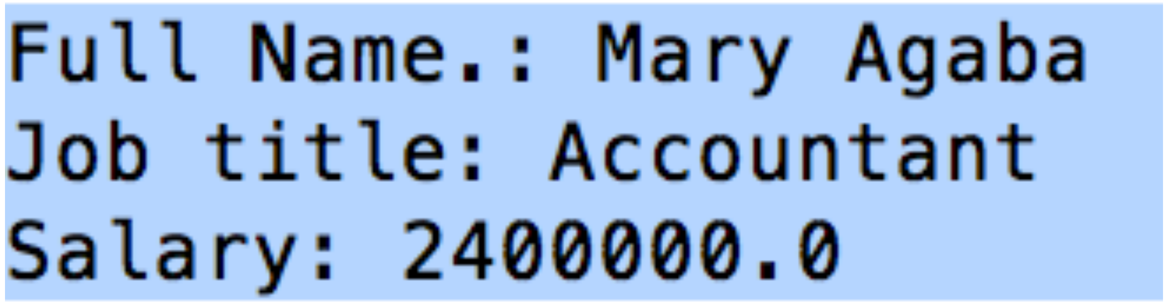
```
class InheritanceDemo1{
 public static void main(String args[]){
   Staff mary = new Staff("Mary","Agaba", "Accountant", 2400000.00);
   mary.printPersonInfo();
   mary.printStaffInfo();
 }
}
```

Similarly, Staff class inherits all the methods of Person class. Hence `mary.printPersonInfo()` results in the invocation of the `printPersonInfo()` of the Person class

Driver Class

```java
class InheritanceDemo1{
 public static void main(String args[]){
   Staff mary = new Staff("Mary","Agaba", "Accountant", 2400000.00);
   mary.printPersonInfo();
   mary.printStaffInfo();
 }
}
```

Output



```
Full Name.: Mary Agaba
Job title: Accountant
Salary: 2400000.0
```

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with **public** visibility are inherited; those with **private** visibility are not
- But remember that **public** variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: **protected**

The protected Modifier

```
class Person {
 protected String firstName;
 protected String lastName;

 public Person(String fName, String lName){
   this.firstName = fName;
   this.lastName = lName;
 }

 public void printPersonInfo (){
   System.out.println("Full Name.: "+ firstName+" "+lastName)
 }
}
```

- The **protected** modifier allows a member of a base class to be inherited into a child
- **Protected** visibility provides more encapsulation than public visibility does. However, protected visibility is not as tightly encapsulated as private visibility

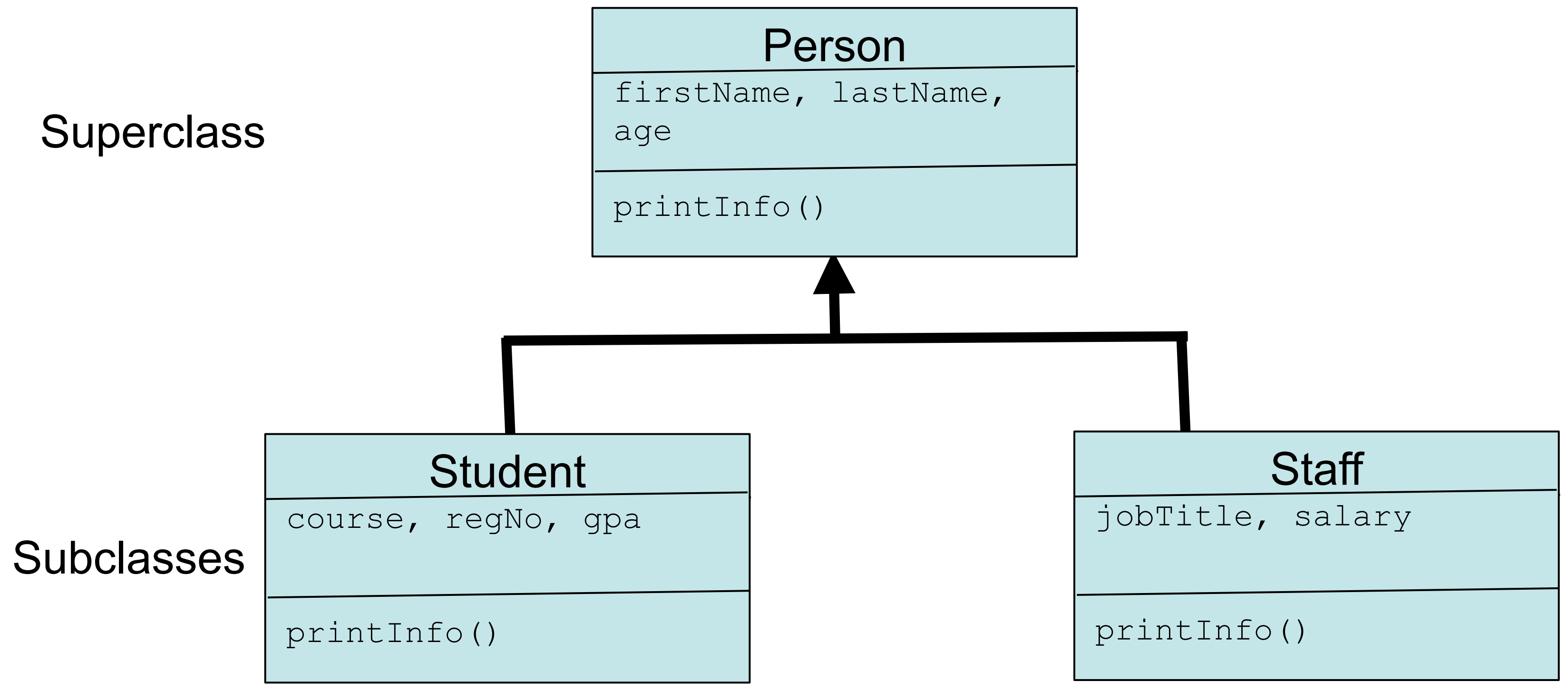
Outline

- Inheritance and **protected** visibility modifier
- Method overriding
- Method overloading
- Polymorphism

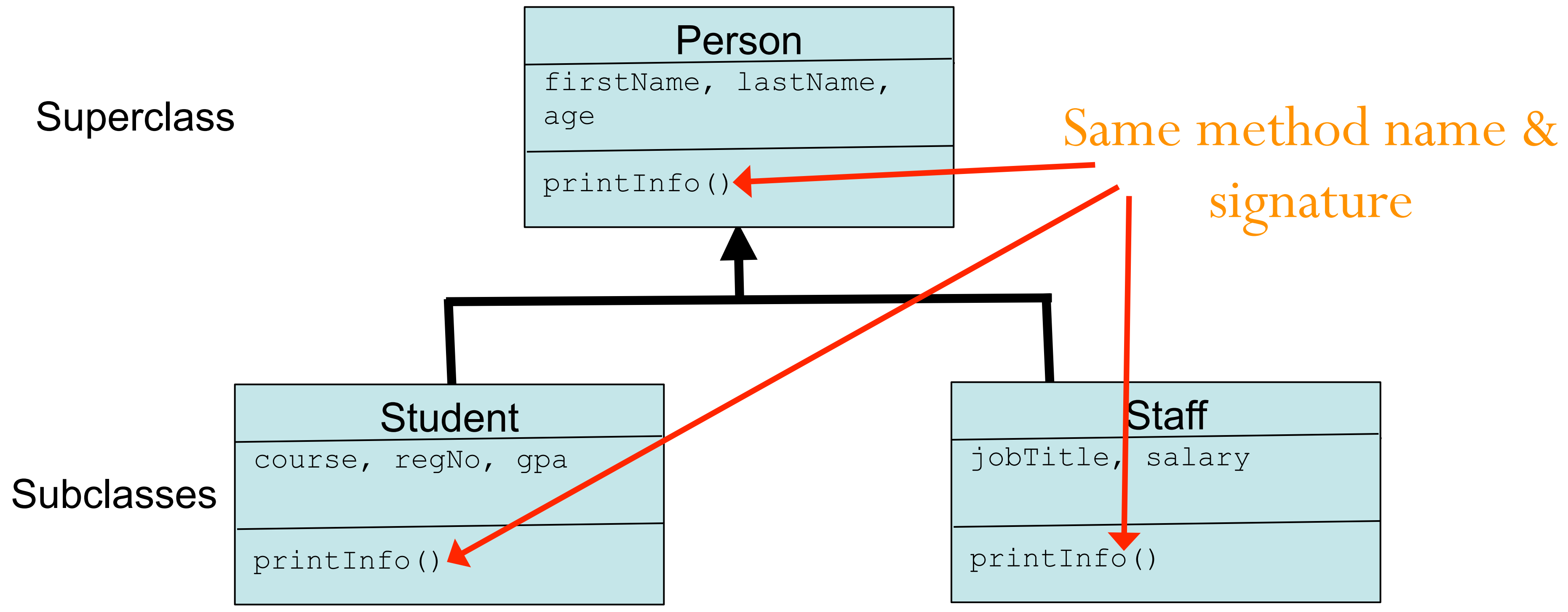
Inheritance: Overriding Methods

- A child class can override the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked

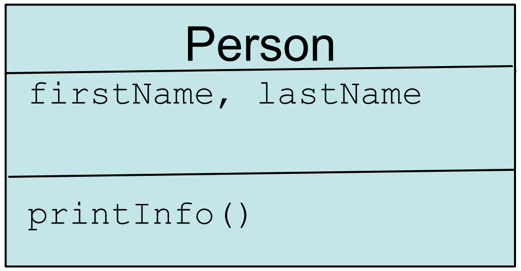
Inheritance: Overriding Methods



Inheritance: Overriding Methods



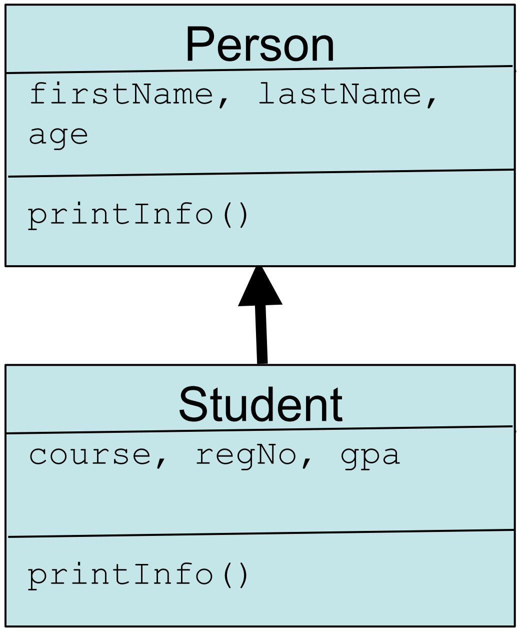
Person Class: Superclass



```
class Person {
 protected String firstName;
 protected String lastName;

 public Person(String fName, String lName){
   this.firstName = fName;
   this.lastName = lName;
 }

 public void printInfo (){
   System.out.println("Full Name.: "+ firstName+" "+lastName);
 }
}
```

Student Class: Subclass

```
class Student extends Person{
 private String course;
 private String regNo;
 private double gpa;

 public Student(String fName,String lName,String course,String regNo,double gpa){
   super(fName, lName);
   this.course = course;
   this.regNo = regNo;
   this.gpa = gpa;
 }

 public void printInfo (){
   super.printInfo();
   System.out.println("Reg No.: "+ regNo);
   System.out.println("Course: "+ course);
   System.out.println("GPA: "+ gpa);
 }
}
```



Student Class: Subclass

```java
class Student extends Person{
 private String course;
 private String regNo;
 private double gpa;

 public Student(String fName,String lName,String course,String regNo,double gpa){
   super(fName, lName);
   this.course = course;
   this.regNo = regNo;
   this.gpa = gpa;
 }

 public void printInfo (){
   super.printInfo();
   System.out.println("Reg No.: "+ regNo);
   System.out.println("Course: "+ course);
   System.out.println("GPA: "+ gpa);
 }
}
```

A parent method can be invoked explicitly using the **super** keyword

Student Class: Subclass

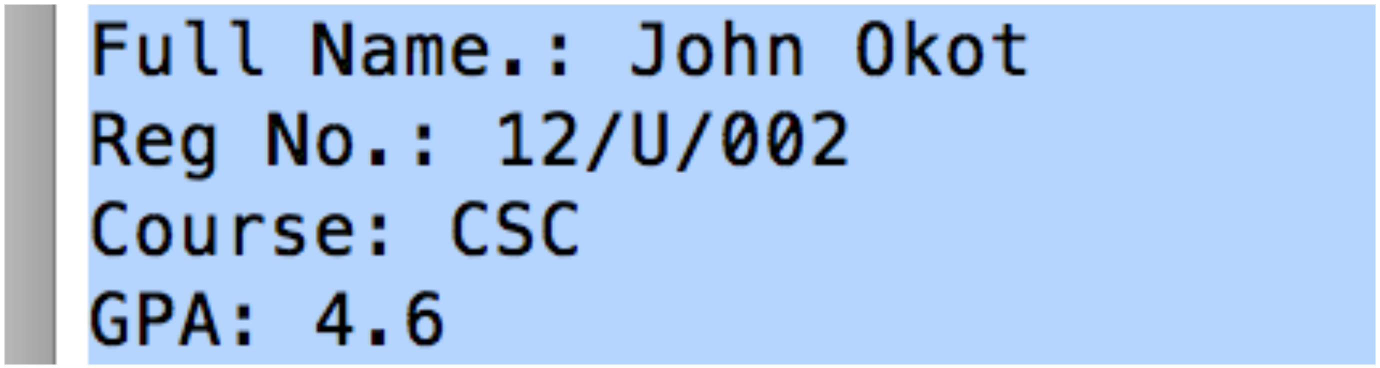
```
class Student extends Person{
 private String course;
 private String regNo;
 private double gpa;

 public Student(String fName,String lName,String course,String regNo,double gpa){
   super(fName, lName);
   this.course = course;
   this.regNo = regNo;
   this.gpa = gpa;
 }

 public void printInfo (){
   super.printInfo();
   System.out.println("Reg No.: "+ regNo);
   System.out.println("Course: "+ course);
   System.out.println("GPA: "+ gpa);
 }
}
```

The **super** keyword is also used to refer to the constructor of the parent class

Driver Class

```
class InheritanceDemo2{
 public static void main(String args[]){
 Student john = new Student("John","Okot", "CSC","12/U/002", 4.60);
 john.printInfo();
  }
}
```

Output



```
Full Name.: John Okot
Reg No.: 12/U/002
Course: CSC
GPA: 4.6
```

Inheritance: Overriding Methods

- **final** keyword:
  - If a method is declared with the **final** modifier, it cannot be overridden.
  - If a class is declared with the **final** keyword, it cannot be subclassed.
- The concept of overriding can be applied to data and is called **shadowing variables**
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Overloading vs. Overriding Methods

- Don't confuse the concepts of **overloading** and **overriding**
- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature

The Object Class

- A class called **`Object`** is defined in the **`java.lang`** package of the Java standard class library
- All classes are derived from the **`Object`** class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the **`Object`** class
- Therefore, the **`Object`** class is the ultimate root (“grandparent”) of all class hierarchies

Outline

- Inheritance and **protected** visibility modifier
- Method overriding
- Method overloading
- Polymorphism

Polymorphism

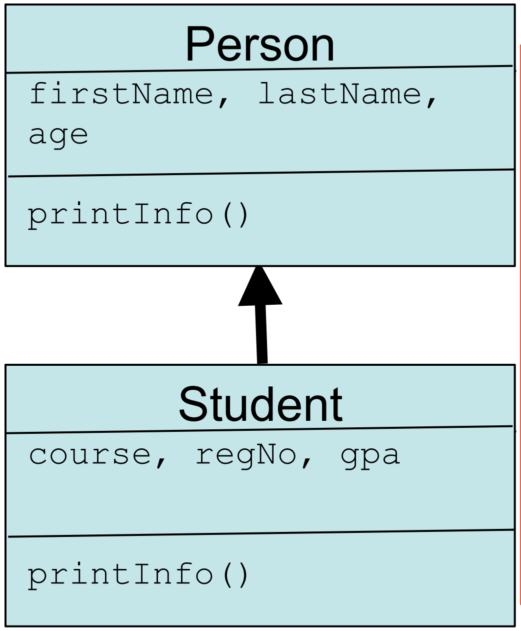
- In object-oriented programming, a reference can be polymorphic, which can be defined as "having many forms"

obj.doIt();

- This line of code might execute different methods at different times if the object that **obj** points to changes
- Polymorphic references are resolved at run time; this is called **dynamic binding**
- Careful use of polymorphic references can lead to elegant, robust software designs
- Polymorphism can be accomplished using inheritance or using interfaces

Polymorphism: References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the **`Person`** class is a superclass of the **`Student`** class, then a **`Person`** reference could be used to point to a **`Student`** object

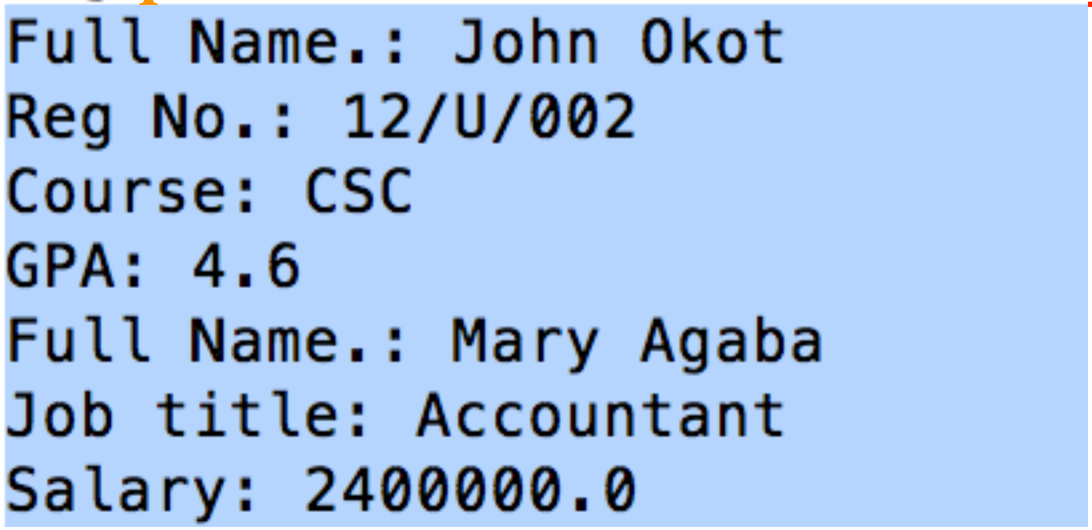


```
class InheritanceDemo2{
 public static void main(String args[]){
 Person p1;
 p1 = new Student("John","Okot", "CSC","12/U/002", 4.60);
 p1.printInfo();
 p1 = new Staff("Mary","Agaba", "Accountant", 2400000.00);
 p1.printInfo();
  }
}
```

Polymorphism: References and Inheritance

```
class InheritanceDemo2{
 public static void main(String args[]){
 Person p1;
 p1 = new Student("John","Okot", "CSC","12/U/002", 4.60);
 p1.printInfo();
 p1 = new Staff("Mary","Agaba", "Accountant", 2400000.00);
 p1.printInfo();
  }
}
```

Output



```
Full Name.: John Okot
Reg No.: 12/U/002
Course: CSC
GPA: 4.6
Full Name.: Mary Agaba
Job title: Accountant
Salary: 2400000.0
```

Exercise

- Read about **method overloading** and outline the differences between overriding and overloading
- Read about ***ad hoc*** **polymorphism**