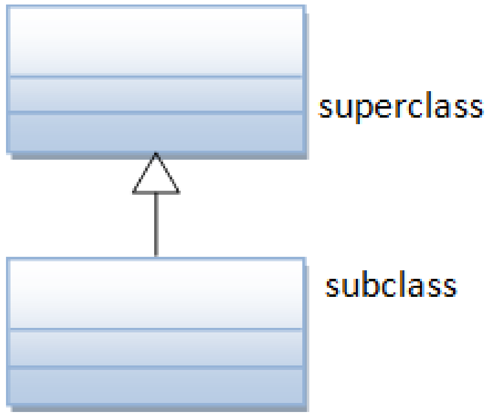
CMPS 251

Lecture 08

Inheritance

Read Chapter 9

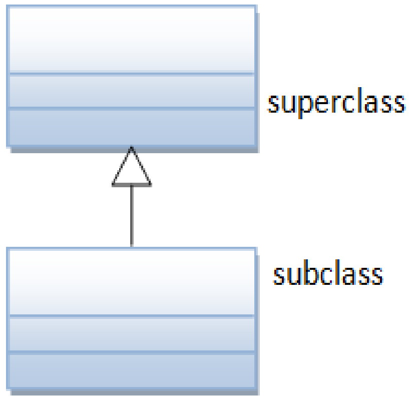


Outline

1. Inheritance Basics
2. Overriding

Inheritance

- **Motivation**
  - Supports the key OOP goal of **code reuse**. Allow us to design **class hierarchies** so that **shared behavior is placed in a super class** then inherited by subclasses (i.e., avoids writing the same code twice to ease maintenance)
- **Ideas**
  - Common attributes and methods are placed in a **superclass** (also called ***parent class*** or ***base class***)
  - You can create a subclass that **inherits** attributes and methods of the super class
    - Subclass also called ***child class*** or ***derived class***
  - Subclass has access to all **non-private** (i.e., *public* and *protected*) **attributes and methods of the superclass**
  - Subclass can also add new attributes/methods and/or overriding the superclass methods
- **Syntax**
  - public class SubClass ***extends*** SuperClass { … }

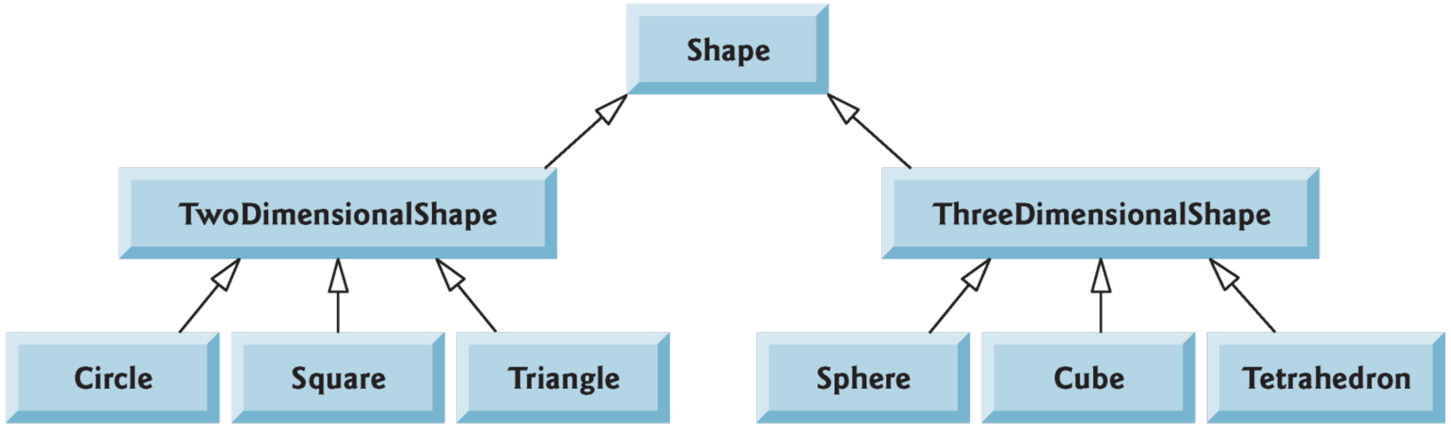


Benefits of Inheritance

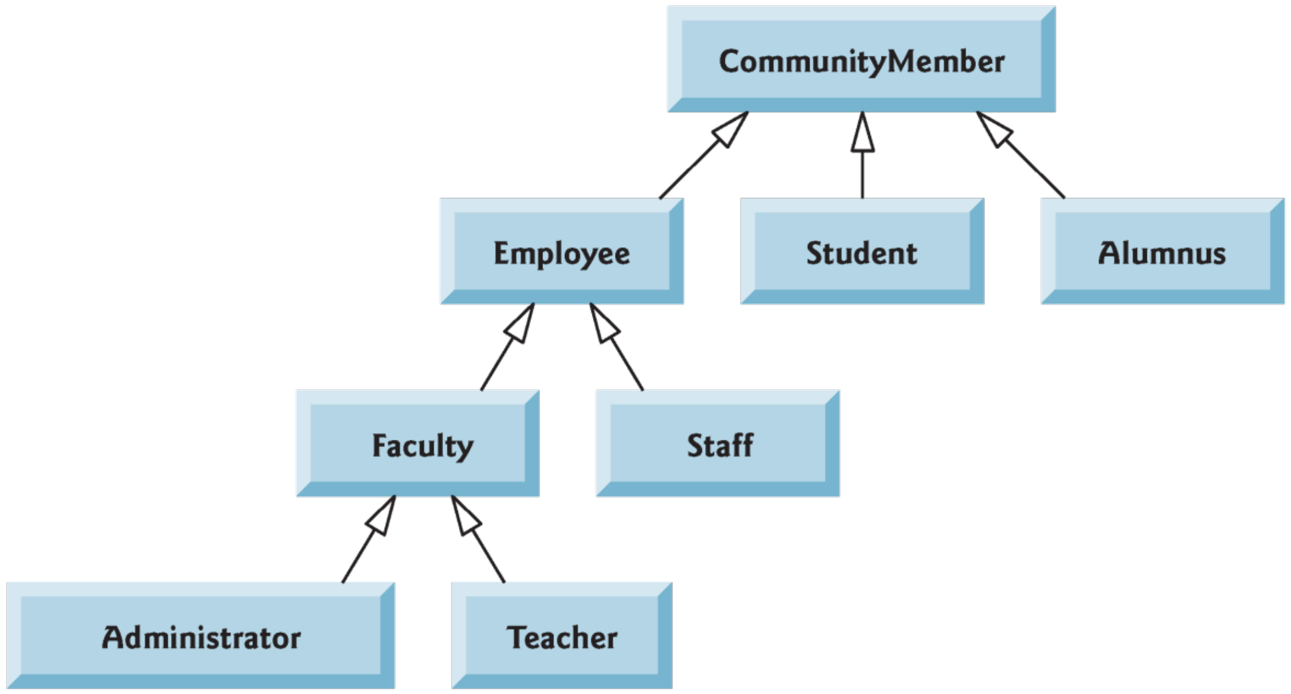
- Benefits of inheritance
  - Can save time during program development by basing new classes on existing tested and quality classes.
  - Increases the likelihood that a system will be implemented and maintained effectively.
  - Reduces duplication => eases maintainability of the code

- **Limitation:**
  - Java supports only **single inheritance**, in which each class is derived from exactly one direct superclass

Basic Graphical Example

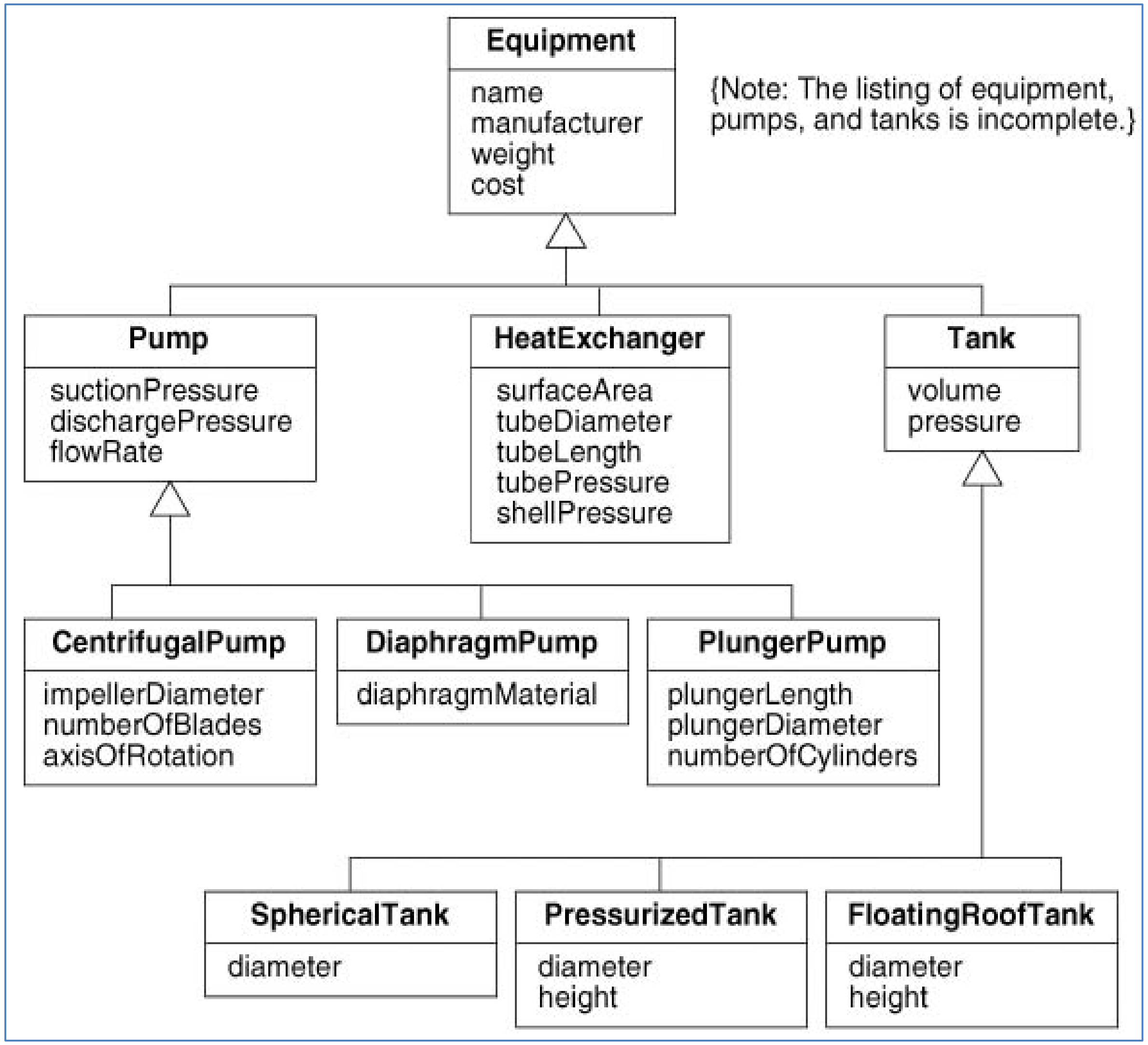


Basic Graphical Example

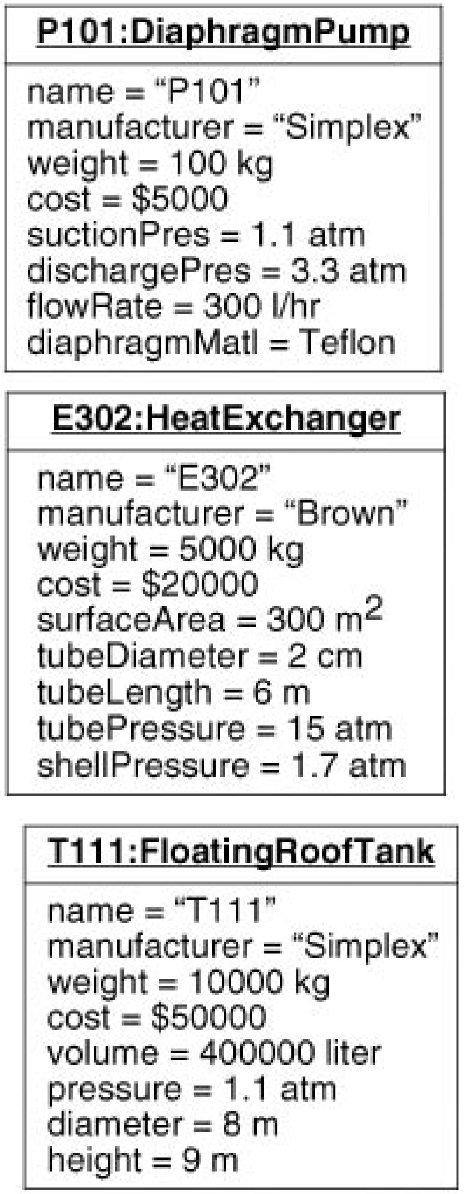


Multilevel Inheritance Hierarchy and Objects

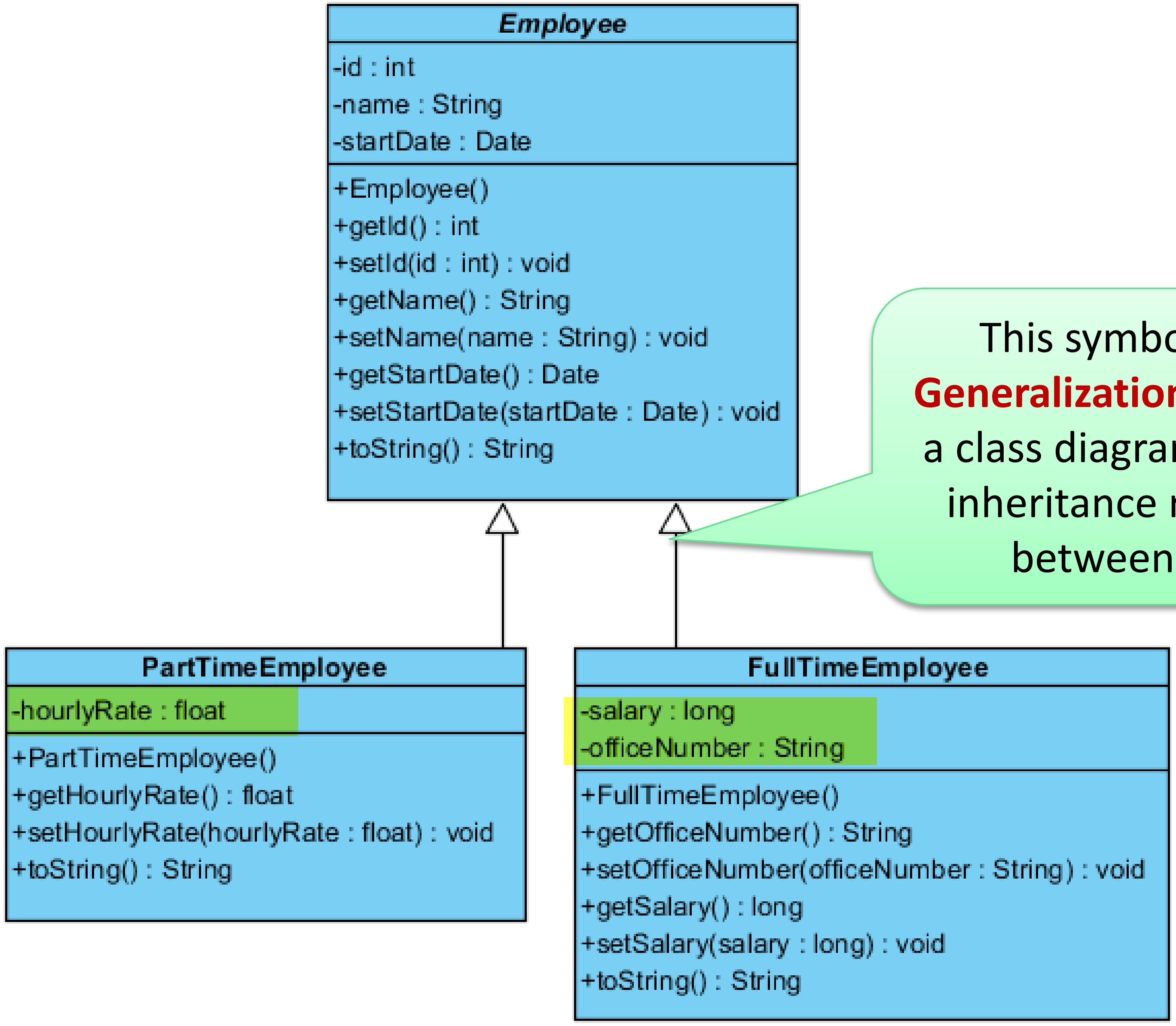
Classes



Objects



Inheritance Example - Employee Hierarchy



This symbol is called **Generalization**. It is used in a class diagram to indicate inheritance relationship between classes.

Superclass - Subclass

- In Java, it is possible to inherit attributes and methods from one class to another.
- We group the "inheritance concept" into two categories:
  - **subclass (**child) - the class that inherits from another class
  - su**perclass (par**ent) - the class being inherited from
- To inherit from a class, use the extends keyword.
- In the example below, the Car class (subclass) inherits the attributes and methods from the Vehicle class (superclass):

```
Class Vehicle {
  protected String brand = "Ford";         // Vehicle attribute
  public void honk() {                     // Vehicle method
    System.out.println("Tuut, tuut!");
  }
}
class Car extends Vehicle {
  private String modelName = "Mustang";     // Car attribute
  public static void main(String[] args) {

    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (from the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand attribute (from the Vehicle class) and the value of
the modelName from the Car class
    System.out.println(myCar.brand + " " + myCar.modelName);
  }
}
```

When to Use Inheritance

- Why protected access modifier is used for the attribute in Vehicle class?
- We set the **brand** attribute in **Vehicle** to a protected access modifier.
- If it was set to private, the Car class would not be able to access it.
- Why and When To Use "Inheritance"?
- It is useful for code reusability:
  - reuse attributes and methods of an existing class when you create a new class.
- Tip: Also take a look at the next chapter, Polymorphism, which uses inherited methods to perform different tasks.
- If you don't allow other classes to inherit from a class, use the final keyword

```
final class Vehicle {
  ...
}
class Car extends Vehicle {
  ...
}
```

- If you try to access a final class, Java will generate an error.

Inheritance and Constructors

- **Constructors are not inherited**
- But a subclass constructor **can call its direct superclass's constructor** to initialize the instance variables inherited from the superclass.
- Syntax—keyword **super**, followed by a set of parentheses containing the superclass constructor arguments. Must be the first statement in the subclass constructor's body.
- If a subclass constructor does not include an explicit call to the superclass constructor, Java implicitly calls the superclass's default constructor.
- You can explicitly use **super()** to call the superclass's default constructor, but this is rarely done.

Example: Inheritance and Constructors

Inheritance!

Customer

```java
public class Customer extends Person {
    public String phone;

    public Customer(String n, String p) {
        super(n);
        this.phone = p;
    }
}
```

Person.java

```java
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }
}
```

Employee.java

```java
public class Employee extends Person {
    public double salary;

    public Employee(String n, double s) {
        super(n);
        this.salary = s;
    }
}
```

Constructors in Subclasses

- Instantiating a subclass object begins a **chain of constructor calls**
  - The subclass constructor, before performing its own tasks, invokes its direct superclass's constructor
- If the superclass is derived from another class, the superclass constructor invokes the constructor of the next class up the hierarchy all the way back to the Constructor of Object
- The last constructor called in the chain is always class Object's constructor.
- Each superclass's constructor initialized the superclass attributes that the subclass object inherits.

Example: Constructors in Subclasses

Employee.java

```java
public class Employee extends Person {
    public double salary;

    public Employee(String n, double s) {
        super(n);
        this.salary = s;
    }

    public double getPayAmount() {
        return this.salary;
    }
}
```

CommissionEmployee.java

```java
public class CommissionEmployee extends Employee {
    double commission;
    int sales;

    public CommissionEmployee(String n, double salary,
                              double commission, int sales) {
        super(n, salary);
        this.commission = commission;
        this.sales = sales;
    }
}
```

Person.java

```java
public class Person {
    private String name;

    public Person(String name)
    {
        this.name = name;
    }
}
```

Example: ArrayList for Superclass

- Objects of a subclass can be stored in references to a superclass
  - This is commonly done in lists and arrays

```
ArrayList<Person> pList = new ArrayList<Person>();

Customer c = new Customer(“Lionel Messi", “123456789");
pList.add(c);

Employee e = new Employee(“Christiano Ronaldo", 5000);
pList.add(e);

CommissionEmployee ce = new CommissionEmployee(“Donald
Duck", 5000, 0.05, 10000);
pList.add(ce);
```

is-a relationship vs. has-a relationship

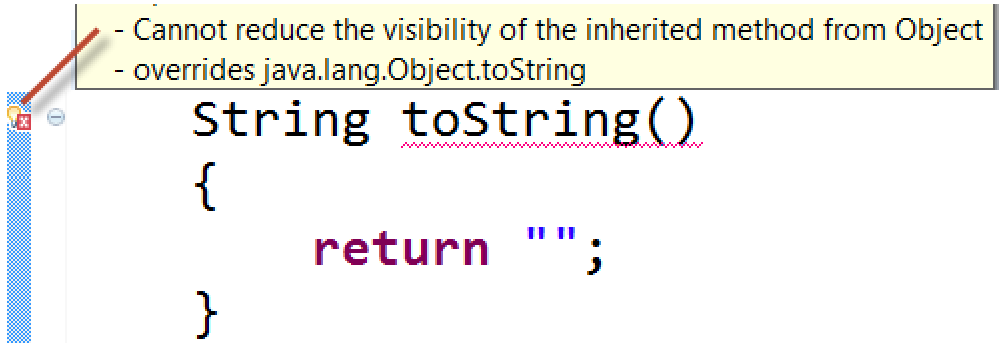
- We distinguish between the is-a relationship and the has-a relationship
- ***Is-a*** represents inheritance
  - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass
  - e.g., Student is a Person
- ***Has-a*** represents composition
  - In a *has-a* relationship, an object contains as attributes references to other objects
  - E.g., Student has a list of courses

The Object Class

- Object is the **root** class of all classes in Java
- All other classes are descendents of Object
- **Object** is part of the java.lang package
- Useful Object methods:
  - toString - returns a **string representation** of the object (by default, its class name and id, but this can be overridden).
  - equals - tests for equality of value of two different objects
  - getClass - returns the class to which an object belongs

Overriding

- Overriding = child class redefines the behavior of the parent
- To override a superclass method, a subclass must **declare a method with the same signature as the superclass method**
  - Same access modifier should be used. E.g. if the superclass method is public the overridden method should also be public.



```
- Cannot reduce the visibility of the inherited method from Object
- overrides java.lang.Object.toString
```

```
String toString()
{
        return "";
}
```

- **@Override is an optional annotation**
  - Declare overridden method with the *@Override* annotation to ensure at compilation time that you defined their signatures correctly.

Example: method override

Employee.java

```
public class Employee extends Person {
    public double salary;

    public Employee(String n, double s) {
        super(n);
        this.salary = s;
    }

    public double getPayAmount() {
       return this.salary;
    }
}
```

Person.java

```
public class Person {
    private String name;

    public Person(String name)
    {
        this.name = name;
    }
}
```

CommissionEmployee.java

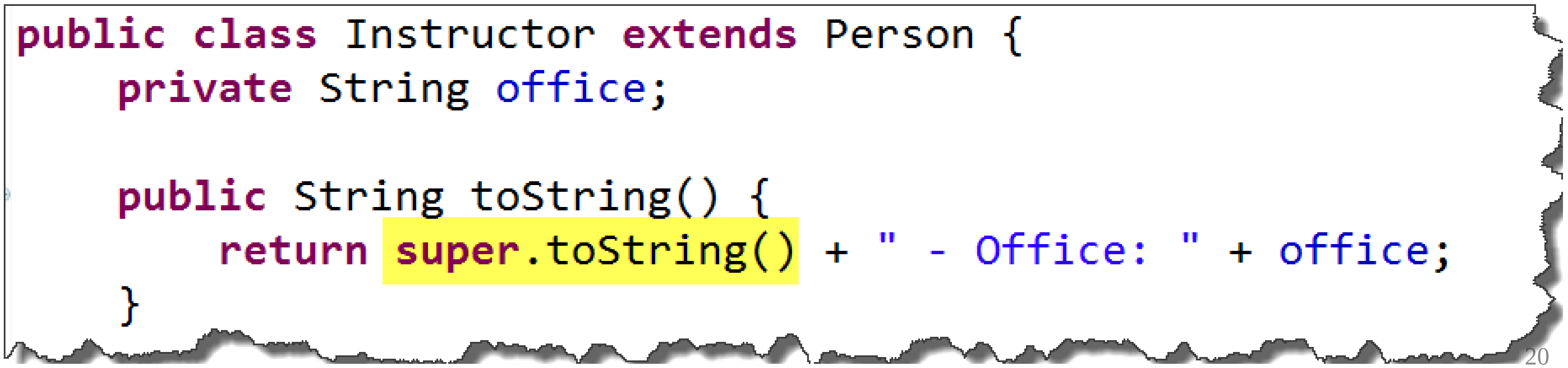
```
public class CommissionEmployee extends Employee {
    double commission;
    int sales;

    public CommissionEmployee(String n, double salary,
                              double commission, int sales) {
        super(n, salary);
        this.commission = commission;
        this.sales = sales;
    }

    @Override
    public double getPayAmount() {
        return salary + commission * sales;
    }
}
```

Overriding

- Overriding allow the subclass to replace/extend the behavior of the superclass.
- When a subclass method overrides an inherited superclass method, the superclass method can be accessed from the subclass by preceding the superclass method name with keyword **super** and a dot (.) separator.



```java
public class Instructor extends Person {
    private String office;

    public String toString() {
        return super.toString() + " - Office: " + office;
    }
```

Important notes

- It is a **syntax error** to **override** a method with a more restricted access modifier.
  - ✓ A **public** method of the **superclass** cannot become **protected** or **private** in the **subclass**.
- A **final** class **cannot be extended** (cannot be a **superclass**)
- A **final** method **cannot be overridden** (cannot change how it works in a **subclass**)!
- When you **extend** a class, the new class inherits the **superclass's** members—though the private **superclass** members are hidden in the new class.
- To enable a subclass to directly access superclass instance variables, we can declare those members as **protected** in the superclass.
- You can check to see if an object is an instance of a specified class using the **instanceof** operator. It returns either true or false.

Summary

- Inheritance = placing common attributes and methods in a superclass so that subclasses can reuse them
- Subclass extends a superclass:
  - inherit the superclass's members, though the `private` superclass members are hidden from the subclass
  - can define their own additional specialized methods / attributes
  - can override an inherited method
- **Constructors are not inherited** but a subclass constructor **can call its direct superclass's constructor** to initialize the superclass attributes