# Week 10: Topics

- Inheritance
- Polymorphism
- Abstraction

# Inheritance Hierarchy

Blackboard: Week9/BasPlusCommissionEmployee_V2

Blackboard: Week9/BasPlusCommissionEmployee_V2

# `protected` Members

- A class's `public` members are accessible whenever the program has a *reference* to an object of the class or one of its subclasses

  Private members are accessible only within the class itself

- The **`protected`** access modifier offers an intermediate level of access between `public` and `private`

  A superclass' protected members can be accessed by its subclasses and by members of other classes in the same package

# BasePlusCommissionEmployee V3

- To enable class `BasePlusCommissionEmployee` to directly access superclass instance variables

  We declare those members as protected in the superclass (i.e., class CommissionEmployee)

```
protected final String fistName;
protected final String lastName;
protected final String socialSecurityNumber;
protected double grossSales;
protected double commissionRate;
```

# Notes Using `protected` Instance Variables

- Inheriting `protected` instance variables

  Enables direct access to the variables by the subclass

- Better to use `private` instance variables to encourage proper software engineering

  Code will be easier to maintain, modify and debug

# Notes Using `protected` Instance Variables

- A subclass object can set the inherited variable's value without using a *set* method

  Could assign an invalid value to the variable

- Subclass methods are more likely to be written so that they depend on the superclass's data implementation

  Subclasses should depend only on the superclass services

# BasePlusCommissionEmployee V4

Blackboard: Week10/BasPlusCommissionEmployee_V4

Blackboard: Week10/BasPlusCommissionEmployee_V4

# BasePlusCommissionEmployee V4

- Class `CommissionEmployee` declares its instance variables as `private` and provides a number of `public` methods

- Methods `earnings` and `toString` use

The class' get methods to obtain the values of its instance variables

- If we decide to change the variable names only the *get* and *set* methods will need to change

Changes occur solely within the superclass

# BasePlusCommissionEmployee V4

Blackboard: Week10/BasPlusCommissionEmployee_V4

Blackboard: Week10/BasPlusCommissionEmployee_V4

# `BasePlusCommissionEmployee` V4 Notes

- Methods `earnings` and `toString` each invoke their superclass versions

  Do not access instance variables directly

- Method `earnings` **overrides** the superclass's `earnings` method

- The new version calls CommissionEmployee's `earnings` method with `super.earnings()`

  Obtains the earnings based on commission alone

# Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
- If the superclass is derived from another class

  The superclass constructor invokes the constructor of the next class up the hierarchy, and so on

- The last constructor called in the chain

  Is always Object's constructor

- Original subclass constructor’s body finishes executing last

# Polymorphism

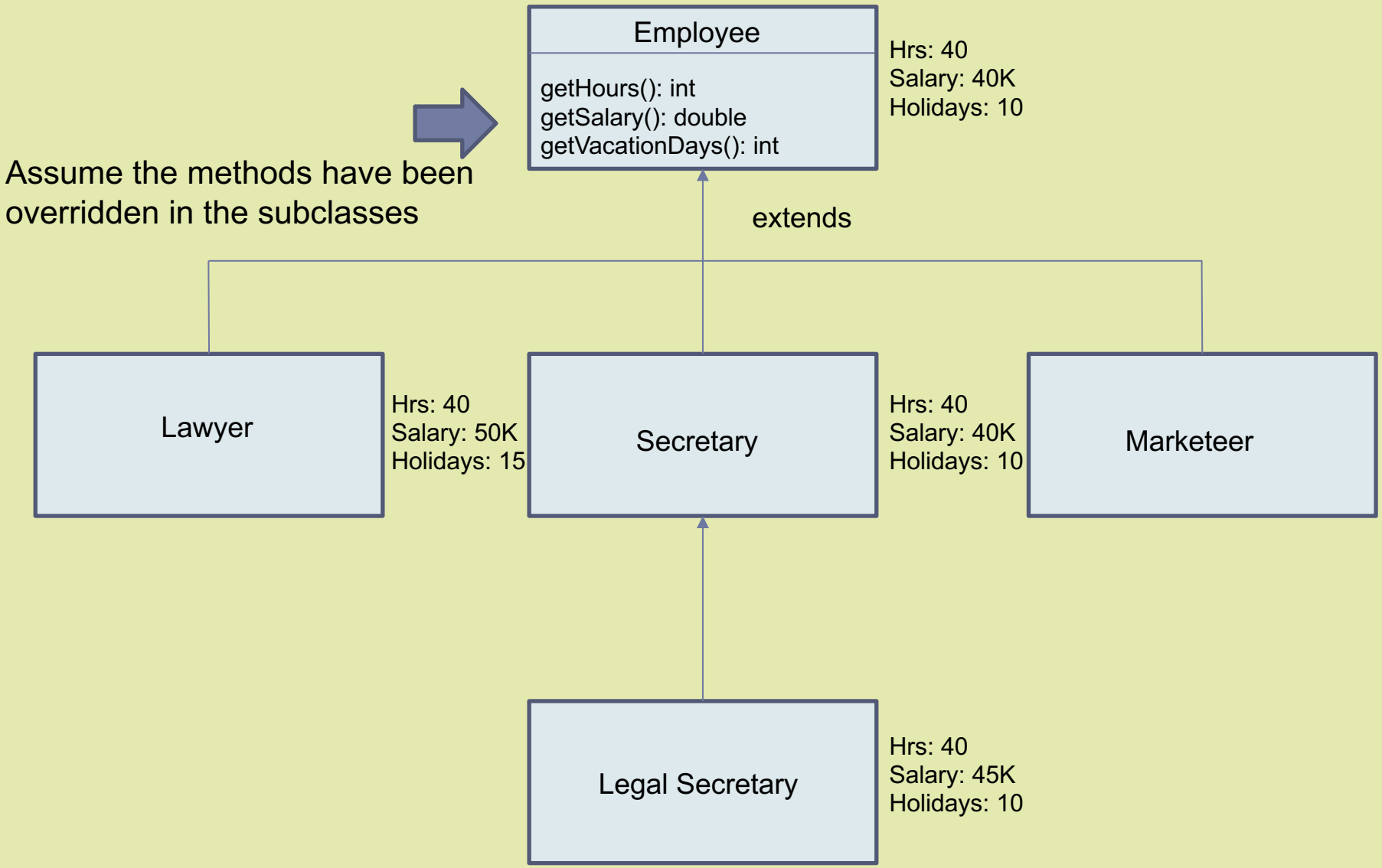
- Enables you to “program in the general” rather than “program in the specific”

- If Rectangle is derived from `Quadrilateral` then a `Rectangle` object is a more specific version of a `Quadrilateral`

  Any operation that can be performed on a Quadrilateral can also be performed on a Rectangle

- Polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable
  - At execution time, the correct subclass version of the method is called based on the type of the reference stored in the superclass variable

# Motivating Polymorphism:



Blackboard: Week10/Polymorphism_Employee

# Motivating Polymorphism (cont.)

- `printInfo` method lets us pass many different types of `Employee` as parameters

  Produces different behavior depending on the type that is passed

- Program does not know which `getSalary` or `getVacationForm` method to call until it's actually running

  This concept is known as late binding

# Problem:

**Upcasting** is the typecasting of a child object to a parent object. Create two classes `Bike` and `Scooter`. `Scooter` extends `Bike` and overrides its `run()` method. Call the `run()` method of the *subclass* by the reference variable of the *parent* class.

Blackboard: Week10/Polymorphism_Upcasting

# More Complex Polymorphism Example

Blackboard: Week10/Polymorphism_FlyingMachines

Blackboard: Week10/Polymorphism_FlyingMachines

# Abstract Classes and Methods

- An abstract class provides a superclass from which other classes can inherit, and thus share a common design

  Cannot be used to instantiate objects

- Classes that can be used to instantiate objects are called *concrete classes*

  Such classes provide implementations of every method they declare. Some of the implementations can be inherited.

# Abstract Classes and Methods

- An abstract method is an instance method with keyword `abstract` in its declaration

  public abstract void draw();

- A class that contains abstract methods must be an abstract class

  -even if that class contains some concrete (non-abstract) methods

# Abstract Superclass Employee

Blackboard: Week10/AbstractClasses_Test

# Concrete Subclass `SalariedEmployee`

Blackboard: Week10/AbstractClasses_Test