Inheritance

Chapter 10

*JAVA: An Introduction to Problem Solving & Programming, 8*th Ed. By Walter Savitch
ISBN 0134462033

Objectives

- Describe inheritance in general
- Define and use derived classes in Java

Inheritance Basics: Outline

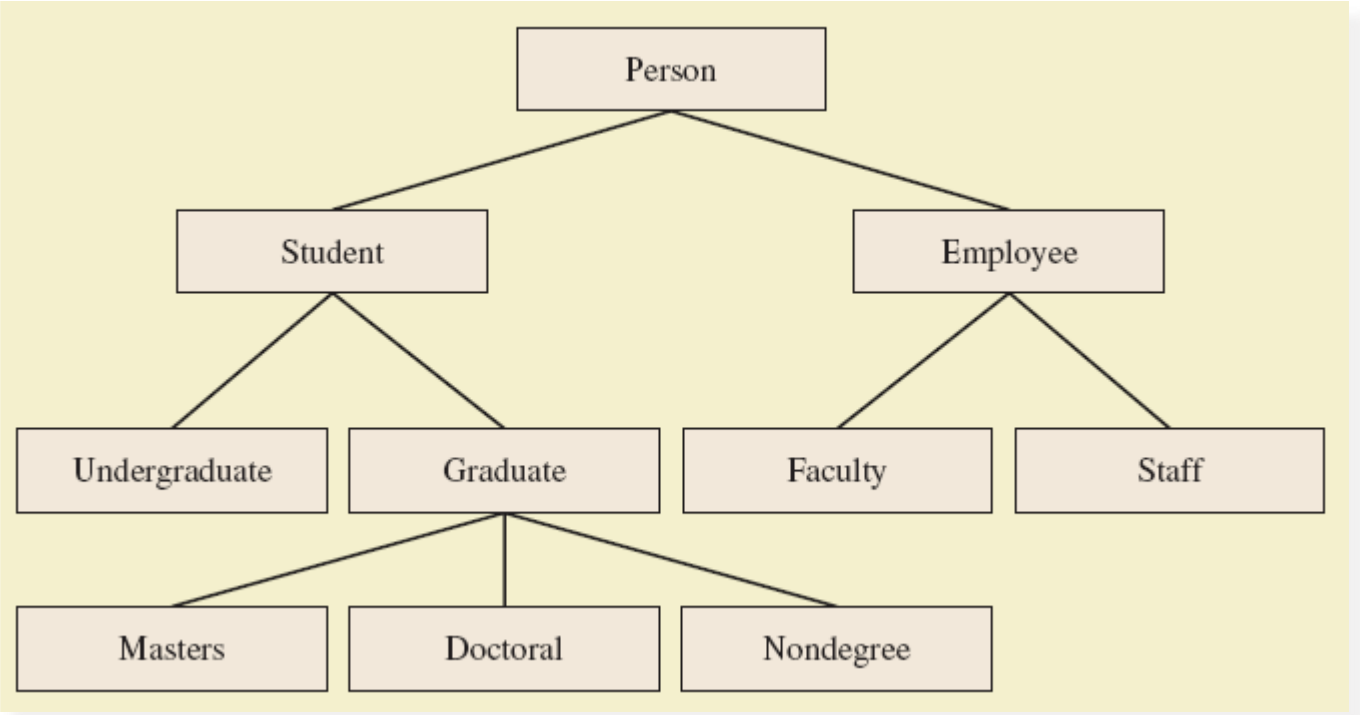
- Derived Classes
- Overriding Method Definitions
- Overriding Versus Overloading
- Private Instance Variables and Private Methods of a Base Class
- UML Inheritance Diagrams

Inheritance Basics

- Inheritance allows programmer to define a general class
- Later you define a more specific class
  - Adds new details to general definition
- New class inherits all properties of initial, general class
- View example class, listing 10.1
  `class Person`

Derived Classes

- Figure 10.1 A class hierarchy



Derived Classes

- Class **`Person`** used as a *base* class
  - Also called *superclass*
- Now we declare *derived* class **`Student`**
  - Also called *subclass*
  - Inherits methods from the superclass
- View derived class, listing 10.2
  `class Student extends Person`
- View demo program, listing 10.3
  `class InheritanceDemo`

Sample screen output

```
Name: Warren Peace
Student Number: 1234
```

Overriding Method Definitions

- Note method **`writeOutput`** in class **`Student`**
  - Class Person also has method with that name
- Method in subclass with same signature overrides method from base class
  - Overriding method is the one used for objects of the derived class
- Overriding method must return same type of value

Overriding Versus Overloading

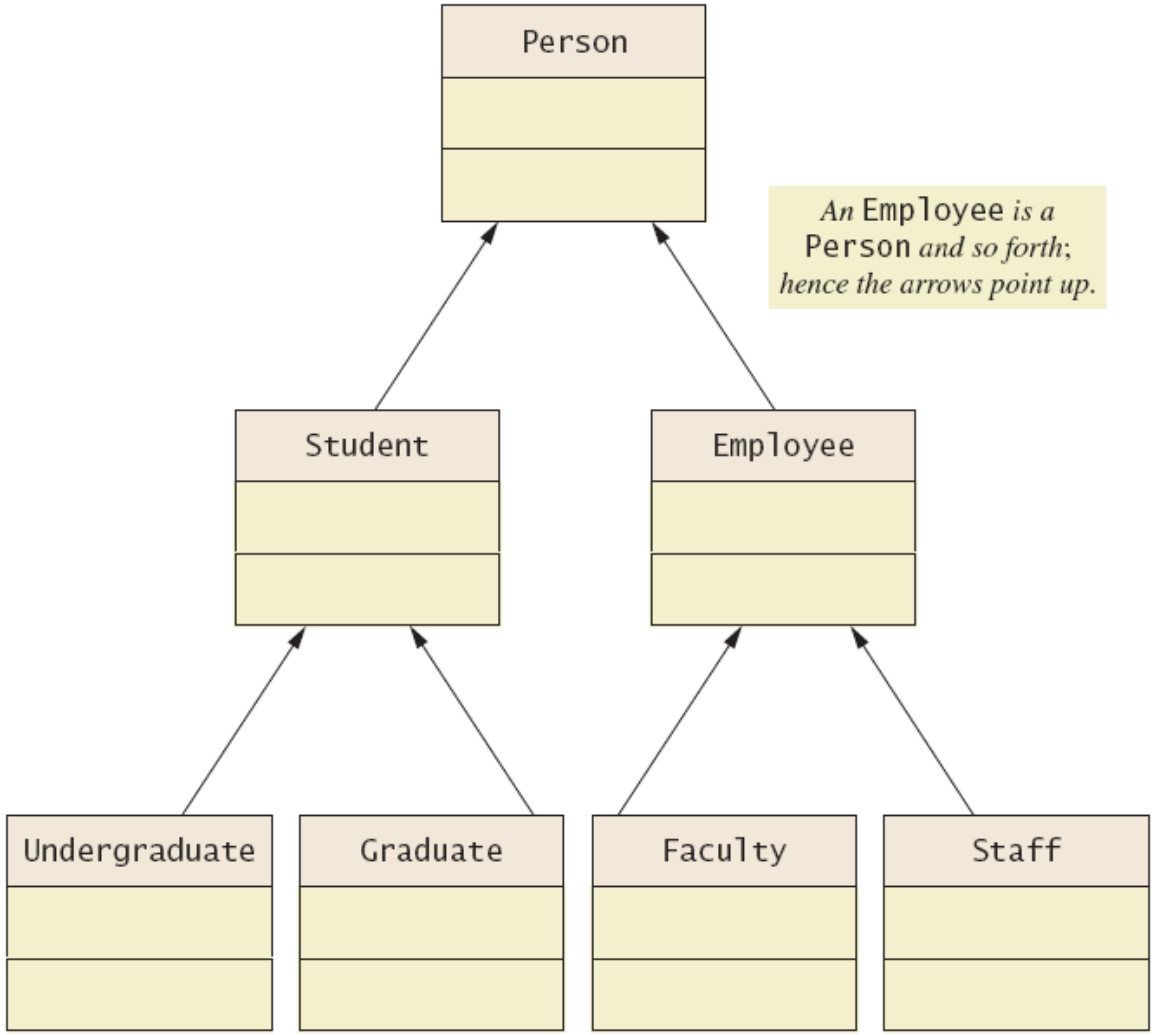
- Do not confuse overriding with overloading
  - Overriding takes place in subclass – new method with same signature
- Overloading
  - New method in same class with different signature

Private Instance Variables, Methods

- Consider private instance variable in a base class
  - It is not inherited in subclass
  - It can be manipulated only by public accessor, modifier methods
- Similarly, private methods in a superclass not inherited by subclass

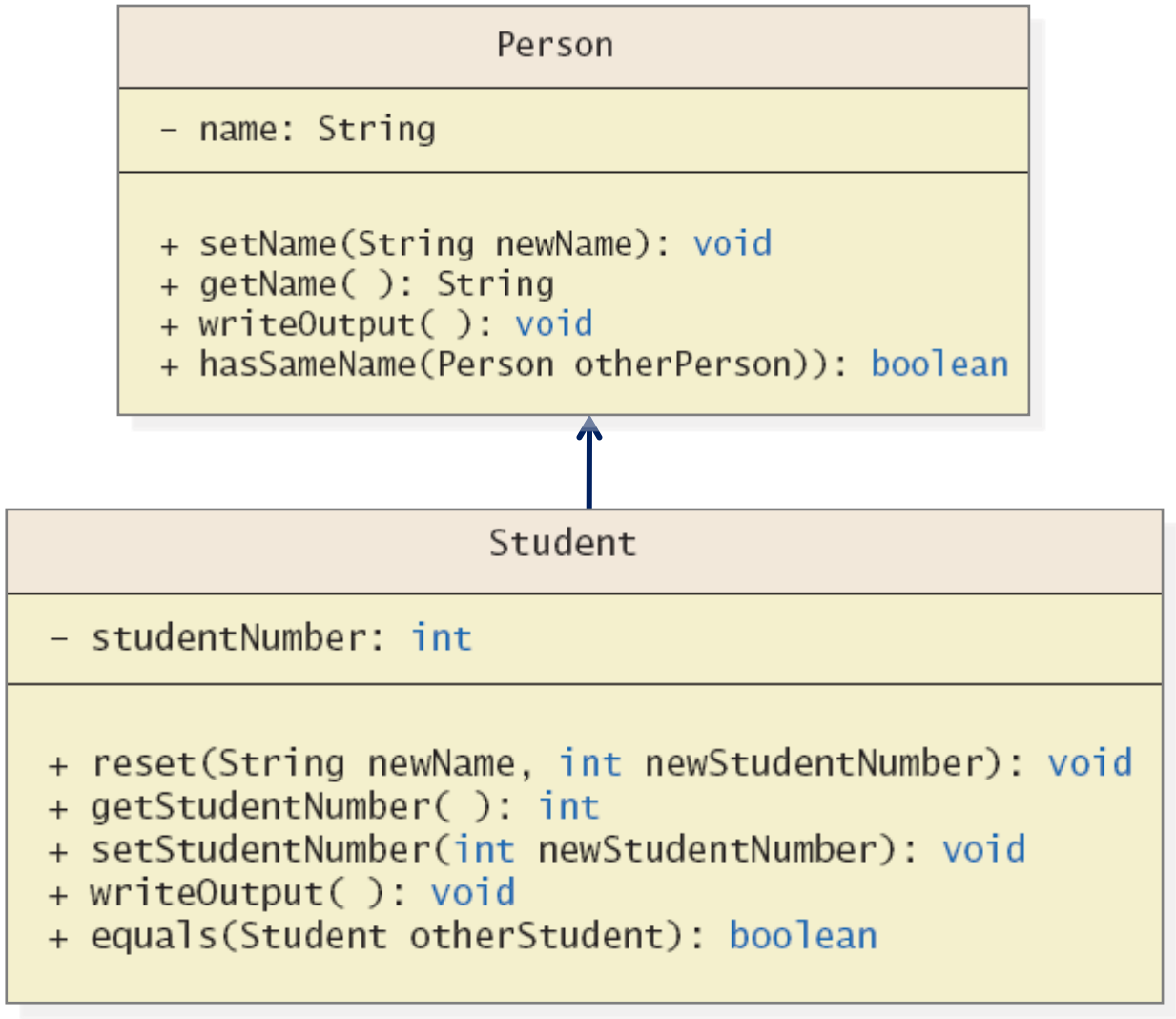
UML Inheritance Diagrams

- Figure 10.2 A class hierarchy in UML notation



UML Inheritance Diagrams

- Figure 10.3 Some details of UML class hierarchy from figure 10.2



Constructors in Derived Classes

- A derived class does not inherit constructors from base class
  - Constructor in a subclass must invoke constructor from base class
- Use the reserve word **`super`**

  - Must be first action in the constructor

```
public Student(String initialName, int initialStudentNumber)
{
    super(initialName);
    studentNumber = initialStudentNumber;
}
```

The `this` Method – Again

- Also possible to use the **`this`** keyword
  - Use to call any constructor in the class

```
public Person()
{
    this("No name yet");
}
```

- When used in a constructor, this calls constructor in same class
  - Contrast use of **`super`** which invokes constructor of base class

Calling an Overridden Method

- Reserved word **`super`** can also be used to call method in overridden method

```
public void writeOutput()
{
    super.writeOutput(); //Display the name
    System.out.println("Student Number: " + studentNumber);
}
```

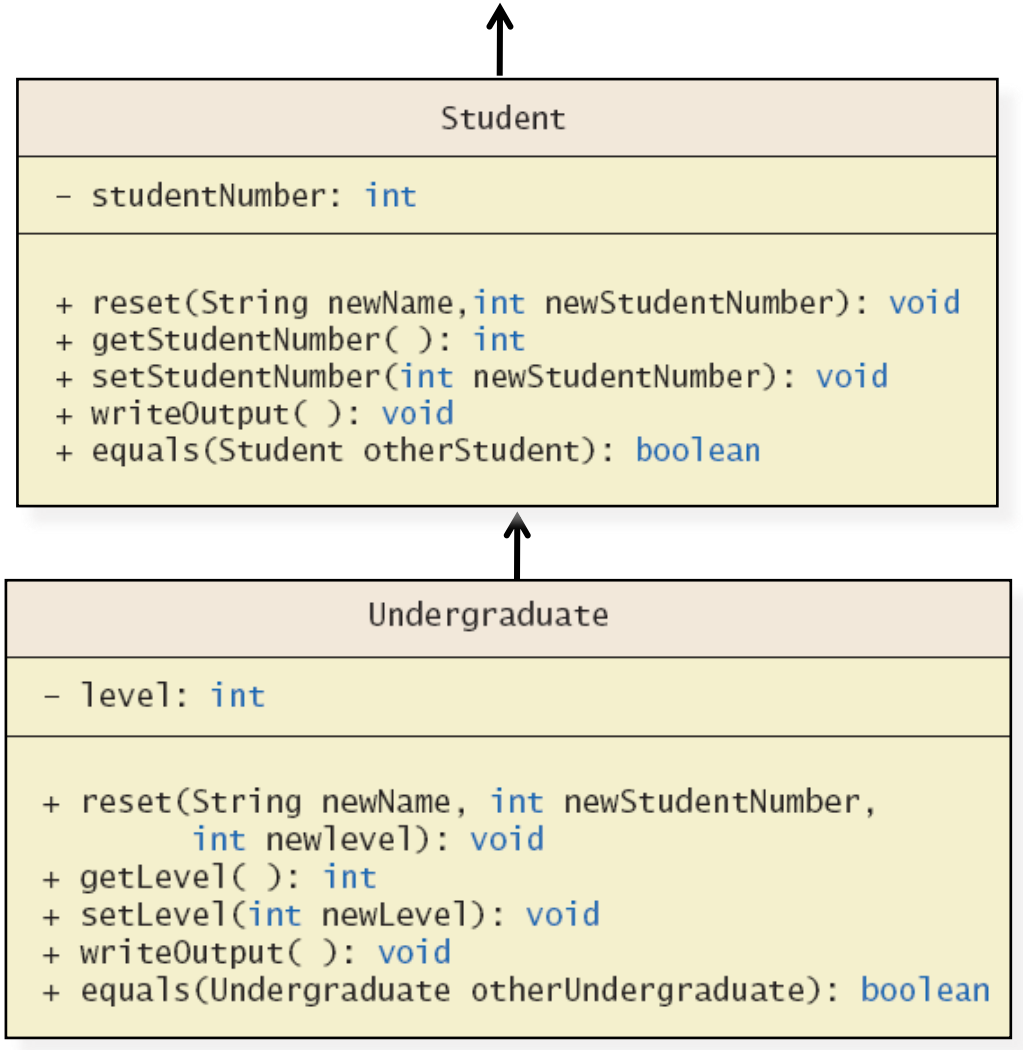
- Calls method by same name in base class

Programming Example

- A derived class of a derived class
- View sample class, listing 10.4
  `class Undergraduate`
- Has all public members of both
  - **`Person`**
  - **`Student`**
- This reuses the code in superclasses

Programming Example

- Figure 10.4 More details of the UML class hierarchy



Type Compatibility

- In the class hierarchy
  - Each **`Undergraduate`** is also a **`Student`**
  - Each **`Student`** is also a **`Person`**
- An object of a derived class can serve as an object of the base class
  - Note this is not typecasting
- An object of a class can be referenced by a variable of an ancestor type

Type Compatibility

- Be aware of the "is-a" relationship
  - A **`Student`** *is a* **`Person`**
- Another relationship is the "has-a"
  - A class can contain (as an instance variable) an object of another type
  - If we specify a date of birth variable for **`Person`** – it "has-a" **`Date`** object

The Class `Object`

- Java has a class that is the ultimate ancestor of every class
  - The class `Object`
- Thus possible to write a method with parameter of type `Object`
  - Actual parameter in the call can be object of any type
- Example: method
  `println(Object theObject)`

The Class `Object`

- Class Object has some methods that every Java class inherits
- Examples
  - Method **`equals`**
  - Method **`toString`**
- Method **`toString`** called when **`println(theObject)`** invoked
  - Best to define your own **`toString`** to handle this

A Better `equals` Method

- Programmer of a class should override method equals from `Object`
- View code of sample override, listing 10.5

```
public boolean equals
        (Object theObject)
```

Summary

- Derived class obtained from base class by adding instance variables and methods
  - Derived class inherits all public elements of base class
- Constructor of derived class must first call a constructor of base class
  - If not explicitly called, Java automatically calls default constructor

Summary

- Within constructor
  - **`this`** calls constructor of same class
  - **`super`** invokes constructor of base class
- Method from base class can be overridden
  - Must have same signature
- If signature is different, method is overloaded

Summary

- Overridden method can be called with preface of **`super`**
- Private elements of base class cannot be accessed directly by name in derived class
- Object of derived class has type of both base and derived classes
- Legal to assign object of derived class to variable of any ancestor type
- Every class is descendant of class **`Object`**