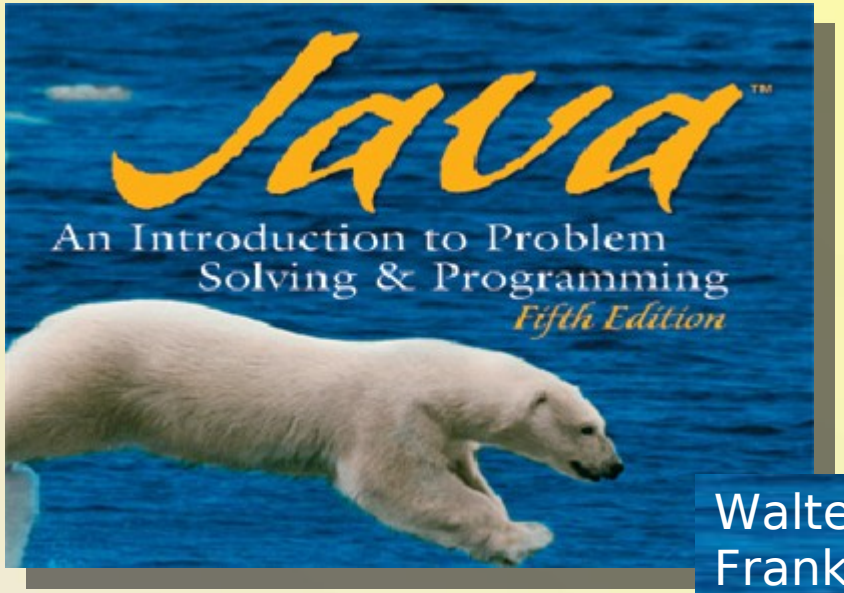


Walter Savitch
Frank M. Carrano

Polymorphism and Inheritance

Chapter 8

*JAVA: An Introduction to Problem Solving & Programming, 5*th Ed. By Walter Savitch and Frank Carrano.
ISBN 0136130887

Objectives

- Describe polymorphism and inheritance in general
- Define interfaces to specify methods
- Describe dynamic binding
- Define and use derived classes in Java

Polymorphism: Outline

- Class Interfaces
- Java Interfaces
- Implementing an Interface
- An Interface as a Type

Class Interfaces

- Consider a set of behaviors for pets
  - Be named
  - Eat
  - Respond to a command
- We could specify method headings for these behaviors
- These method headings can form a class interface

Class Interfaces

- Now consider different classes that implement this interface
  - They will each have the same behaviors
  - Nature of the behaviors will be different
- Each of the classes implements the behaviors/methods differently

Java Interfaces

- Download all source files from the SavitchSrc link:

  ch08/polymorphism

Java Interfaces

- A program component that contains headings for a number of public methods
  - Will include comments that describe the methods
- Interface can also define public named constants
- View Measurable.java

Java Interfaces

- Interface name begins with uppercase letter
- Stored in a file with suffix `.java`
- Interface does not include
  - Declarations of constructors
  - Instance variables
  - Method bodies

Implementing an Interface

- To implement an interface, a class must
  - Include the phrase
    `implements Interface_name`
  - Define each specified method
- View Rectangle.java
  `class Rectangle implements Measurable`
- View another class, Circle.java which also implements Measurable

An Inheritance as a Type

- Possible to write a method that has a parameter as an interface type
  - An interface is a reference type
- Program invokes the method passing it an object of any class which implements that interface
- See Driver.java, Driver2.java, Driver3.java
  - box has 2 types: Rectangle and Measurable
  - disc has 2 types: Circle and Measurable

An Inheritance as a Type

- The method can substitute one object for another
  - Called *polymorphism*
- This is made possible by
  - *dynamic binding*
  - ... also known as *late binding*

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

- Download from SavitchSrc link:
- ch08
  - InheritanceDemo.java
  - Person.java
  - Student.java
  - Undergraduate.java
  - UndergraduateDemo.java

Inheritance Basics

- Inheritance allows programmers 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 **`Person.java`**

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 **`Student.java`**

```
class Student extends Person
```

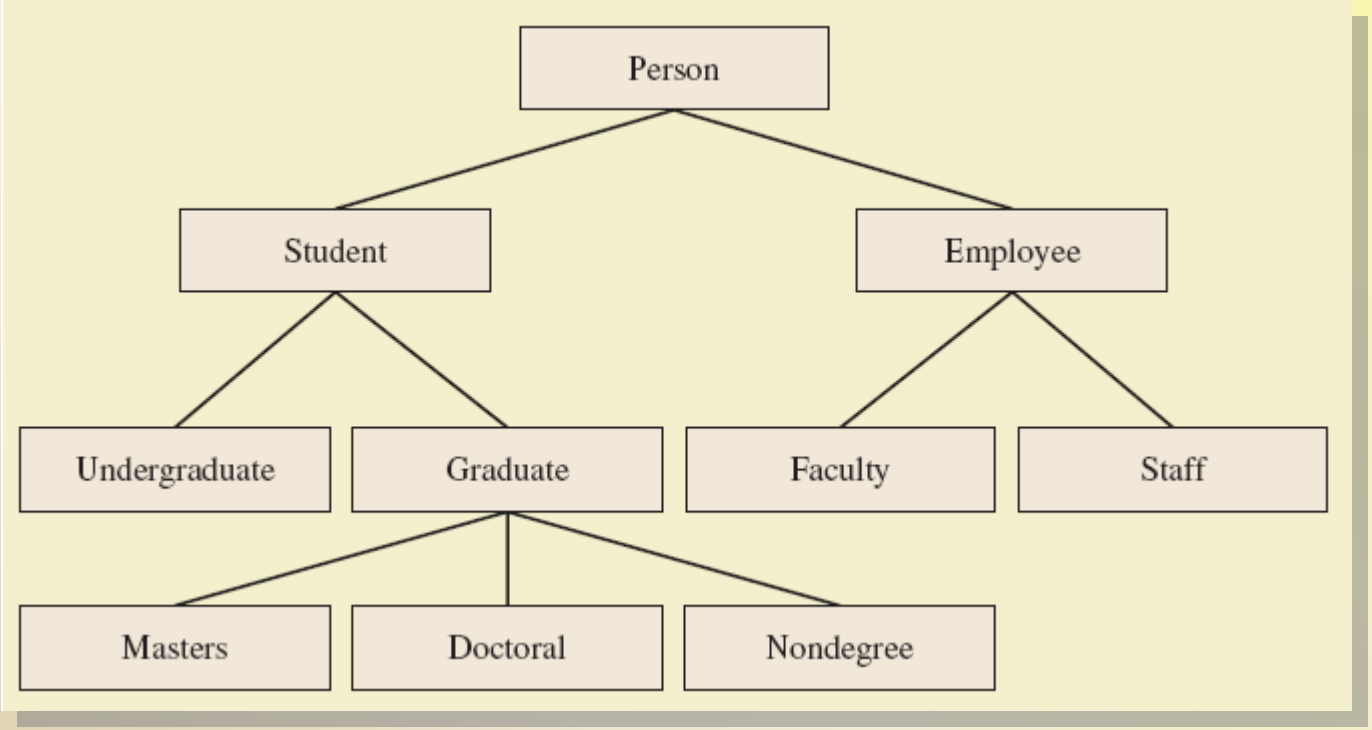
- View **`InheritanceDemo.java`**

Sample screen output

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

Derived Classes

- Figure 8.1 A class hierarchy



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

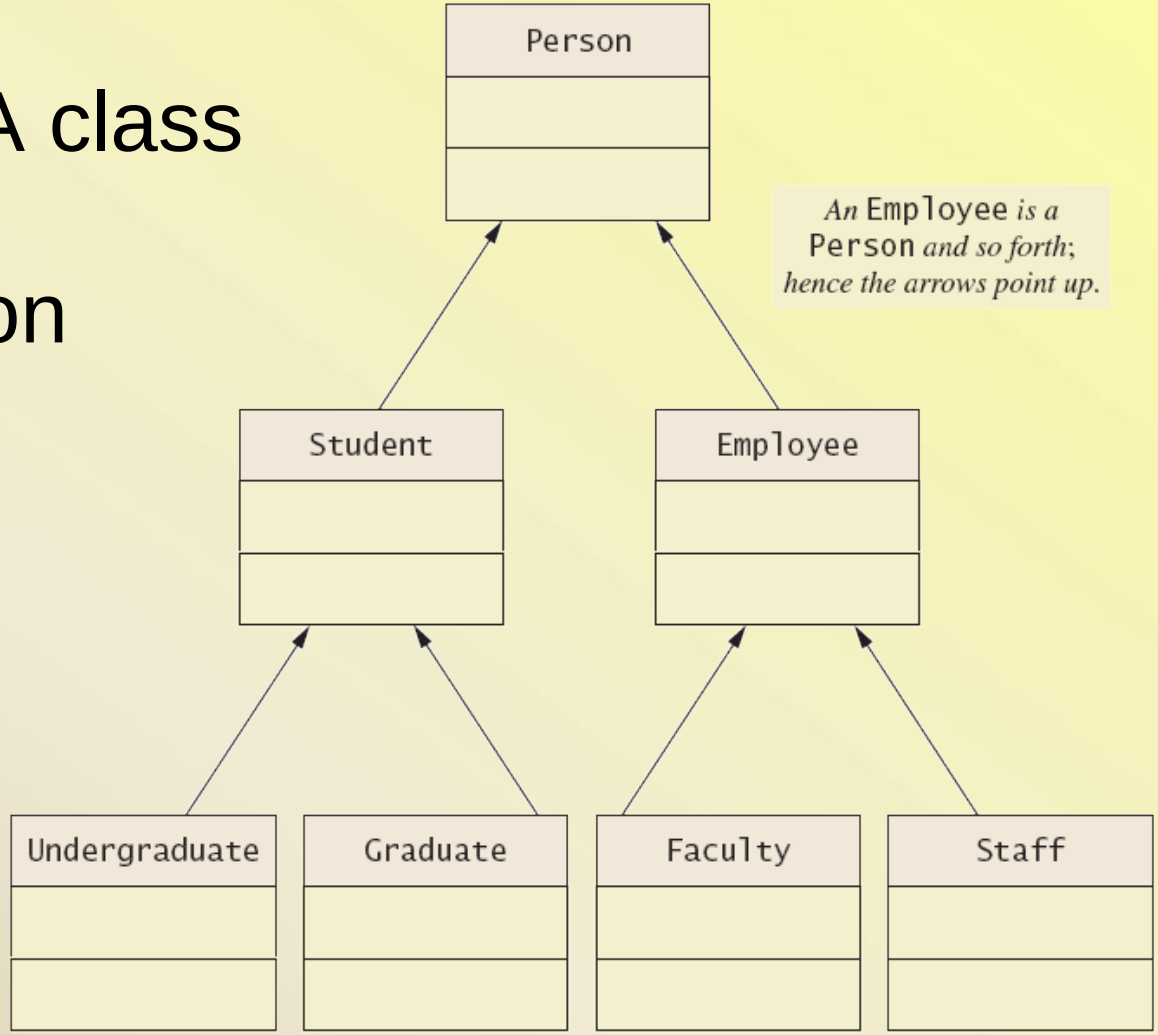
- Do not confuse overriding with overloading
  - Overriding takes place in subclass – new method with **same signature**
- Overloading
  - New method in same class with different signature

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

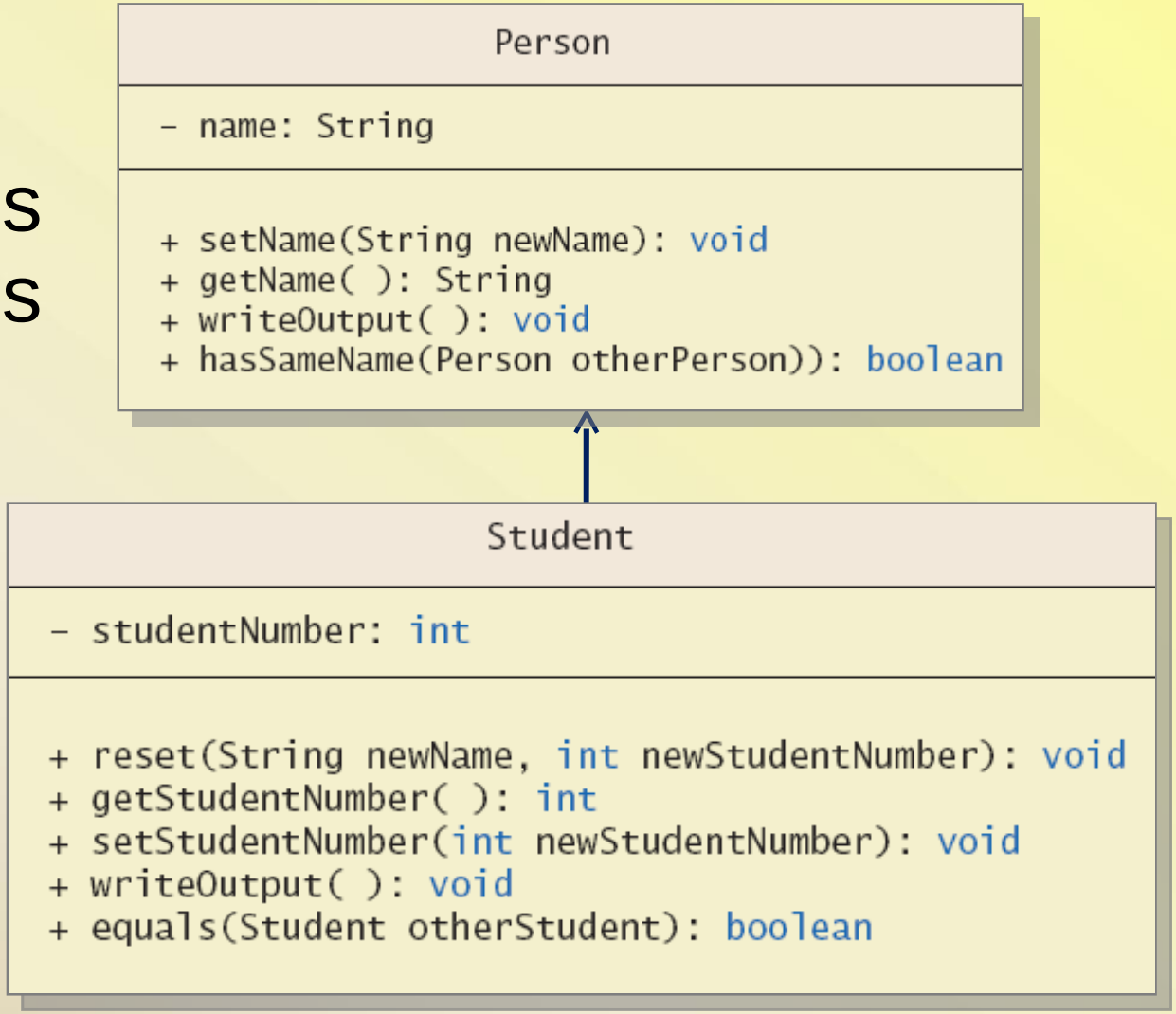
UML Inheritance Diagrams

- Figure 8.2 A class hierarchy in UML notation



UML Inheritance Diagrams

- Figure 8.3 Some details of UML class hierarchy from Figure 8.2



Programming with Inheritance: Outline

- Constructors in Derived Classes
- The `this` Method – Again
- Calling an Overidden Method
- Derived Class of a Derived Class
- Type Compatibility

Programming with Inheritance: Outline

- The class **`Object`**
- A Better **`equals`** Method
- Case Study: Character Graphics
- Abstract Classes
- Dynamic Binding and Inheritance

Constructors in Derived Classes

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

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

  - Must be first action in the constructor

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);
}
```

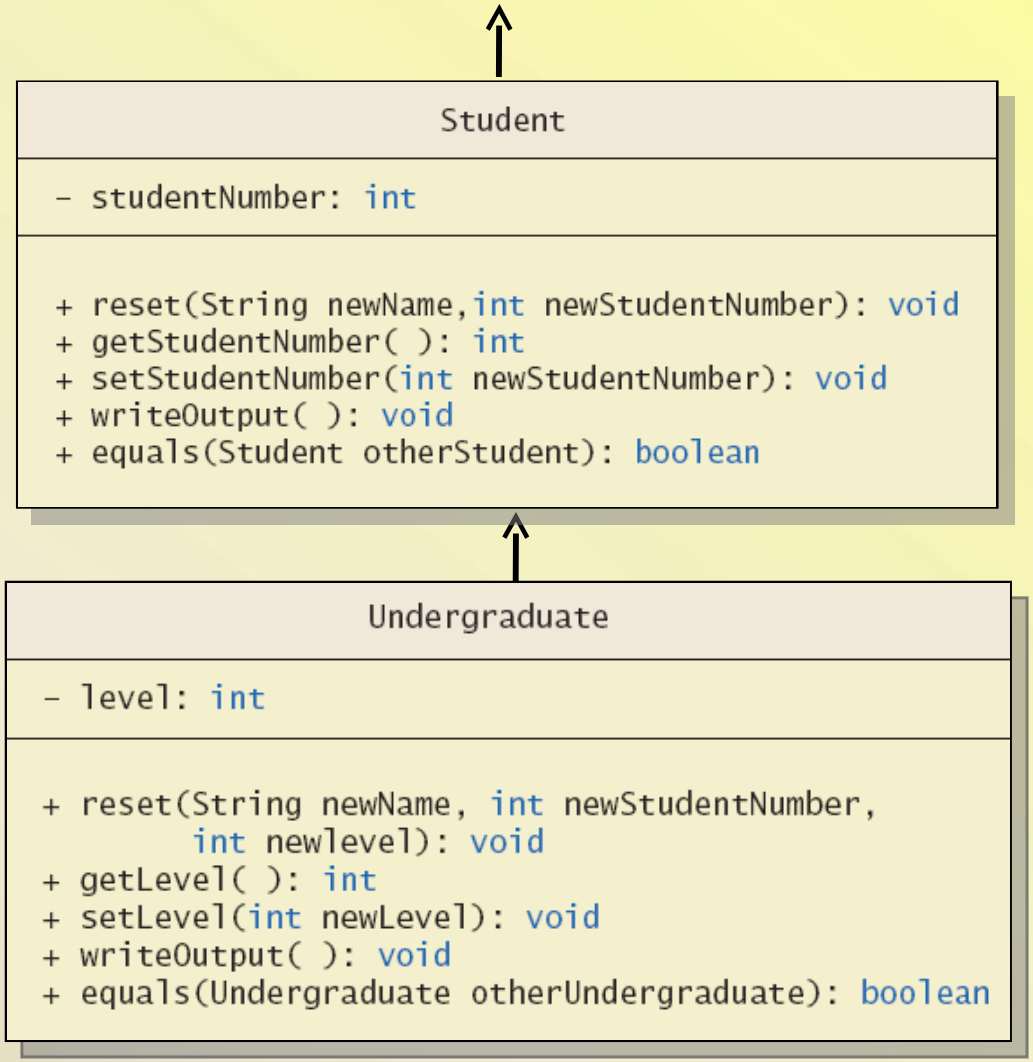
- Calls method by same name in base class

Programming Example

- A derived class of a derived class
- View **`Undergraduate.java`**
- Has all public members of both
  - **`Person`**
  - **`Student`**
- This reuses the code in superclasses

Programming Example

- Figure 8.4 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

Type Compatibility

- An object can have more than one type
- In an assignment statement where left and right are object references:

```
left = right;  // ok if right "is-a" left
```

- Example:

```
Student s = new Student();
Person p = new Person();
p = s; // ok – a Student "is-a" Person
s = p; // illegal – a Person is not a Student
```

The Class `Object`

- Java has a class that is the ultimate ancestor of every class ("Eve 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

- Download Examples: Parent.java Child.java
- Programmer of a class should override method equals from **`Object`**
- Use equals method in Student.java as a model for writing your own.
- View Student.java equals method

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

Abstract Classes

- Classes can be designed to be a base class for other classes
  - Some methods must be redefined for each subclass
  - These methods should be declared *abstract* – a method that has no body
- This makes the class abstract
- You cannot create an object of an abstract class – thus its role as base class

Abstract Classes

- Not all methods of an abstract class are abstract methods
- Abstract class makes it easier to define a base class
  - Specifies the obligation of designer to override the abstract methods for each subclass

Abstract Classes

- Cannot have an instance of an abstract class
  - But OK to have a parameter of that type
- Think of an abstract class as something between an **interface** (no methods implemented) and a **complete class definition** (all methods implemented)

Summary

- An interface contains
  - Headings of public methods
  - Definitions of named constants
  - No constructors, no private instance variables
- Class which implements an interface must
  - Define a body for every interface method specified
- Interface enables designer to specify methods for another programmer

Summary

- Interface is a reference type
  - Can be used as variable or parameter type
- Dynamic (late) binding enables objects of different classes to substitute for one another
  - Called polymorphism

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