Chapter 7

Inheritance

- Inheritance Basics
- Programming with Inheritance
- Dynamic Binding and Polymorphism

Principles of OOP

- OOP - Object-Oriented Programming
- Principles discussed in previous chapters:
  - » Information Hiding
  - » Encapsulation
  - » Polymorphism
- In this chapter
  - » Inheritance

Why OOP?

- To try to deal with the complexity of programs
- To apply principles of abstraction to simplify the tasks of writing, testing, maintaining and understanding complex programs
- To increase code reuse
  - » to reuse classes developed for one application in other applications instead of writing new programs from scratch ("Why reinvent the wheel?")
- Inheritance is a major technique for realizing these objectives

Inheritance Overview

- Inheritance allows you to define a very general class then later define more specialized classes by adding new detail
  - » the general class is called the *base* or *parent class*
- The specialized classes *inherit* all the properties of the general class
  - » specialized classes are *derived* from the base class
  - » they are called *derived* or *child* classes
- After the general class is developed you only have to write the "difference" or "specialization" code for each derived class
- A *class hierarchy:* classes can be derived from derived classes (child classes can be parent classes)
  - » any class higher in the hierarchy is an *ancestor class*
  - » any class lower in the hierarchy is a *descendent class*

An Example of Inheritance: a **`Person`** Class

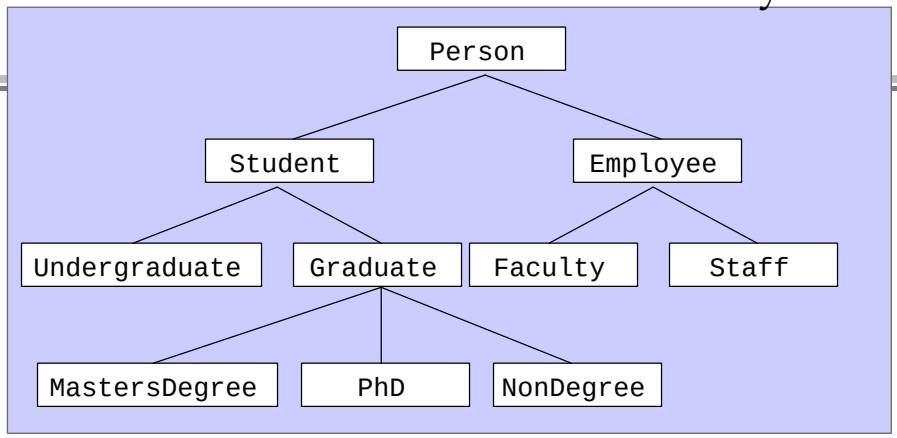
The base class: Display 7.1

- Constructors:
  - » a default constructor
  - » one that initializes the `name` attribute (instance variable)
- Accessor methods:
  - » `setName` to change the value of the `name` attribute
  - » `getName` to read the value of the `name` attribute
  - » `writeOutput` to display the value of the `name` attribute
- One other class method:
  - » `sameName` to compare the values of the `name` attributes for objects of the class
- Note: the methods are `public` and the `name` attribute `private`

A **`Person`** Base Class
Display 7.1

```
public class Person
{
    private String name;
    public Person()
    {
        name = "No name yet.";
    }
    public Person(String initialName)
    {
        name = initialName;
    }
    public void setName(String newName)
    {
        name = newName;
    }
    public String getName()
    {
        return name;
    }
   public void writeOutput()
    {
        System.out.println("Name: " + name);
    }
    public boolean sameName(Person otherPerson)
    {
        return (this.name.equalsIgnoreCase(otherPerson.name));
    }
}
```

Derived Classes: a Class Hierarchy



- The base class can be used to implement specialized classes
  - » For example: student, employee, faculty, and staff
- Classes can be derived from the classes derived from the base class, etc., resulting in a *class hierarchy*

Example of Adding Constructor in a Derived Class: **`Student`**

```
public class Student extends Person
{
    private int studentNumber;
    public Student()
    {
        super();
        studentNumber = 0;
    }
...
```

The first few lines of `Student` class (Display 7.3):

- Keyword *`extends`* in first line
  - » creates derived class from base class
  - » this is inheritance
- Two new constructors (one on next slide)
  - » default initializes attribute `studentNumber` to 0
- *`super`* must be first action in a constructor definition
  - » Included automatically by Java if it is not there
  - » *`super()`*calls the parent default constructor

Example of Adding Constructor in a Derived Class: **`Student`**

- Passes parameter `newName` to constructor of parent class
- Uses second parameter to initialize instance variable that is not in parent class.

```
public class Student extends Person
{
. . .
    public Student(String newName, int newStudentNumber)
    {
        super(newName);
        studentNumber = newStudentNumber;
    }
. . .
```

More lines of `Student` class (Display 7.3):

More about Constructors in a Derived Class

- Constructors can call other constructors
- Use `super` to invoke a constructor in parent class
  - » as shown on the previous slide
- Use `this` to invoke a constructor within the class
  - » shown on the next slide
- Whichever is used must be the first action taken by the constructor
- Only one of them can be first, so if you want to invoke both:
  - » Use a call with `this` to call a constructor with `super`

Example of a constructor using **`this`**

`Student` class has a constructor with two parameters: `String` for the `name` attribute and `int` for the `studentNumber` attribute

```
public Student(String newName, int newStudentNumber)
 {
      super(newName);
      studentNumber = newStudentNumber;
 }
```

Another constructor within `Student` takes just a `String` argument and initializes the `studentNumber` attribute to a value of 0:

» calls the constructor with two arguments, initialName (`String`) and 0 (`int`), within the same class

```
public Student(String initialName)
{
    this(initialName, 0);
}
```


Example of Adding an Attribute in a Derived Class: **`Student`**

A line from the `Student` class:

```
private int studentNumber;
```

- Note that an attribute for the student number has been added
  - » `Student` has this attribute in addition to `name`, which is inherited from `Person`

Example of Overriding a Method in a Derived Class: **`Student`**

- Both parent and derived classes have a `writeOutput` method
- Both methods have the same parameters (none)
  - » they have the same *signature*
- The method from the derived class *overrides* (replaces) the parent's
- It will not override the parent if the parameters are different (since they would have different signatures)
- This is *overriding*, not overloading

```
public void writeOutput()
{
   System.out.println("Name: " + getName());
   System.out.println("Student Number : "
                                    studentNumber);
}
```


Call to an Overridden Method

- Use `super` to call a method in the parent class that was overridden (redefined) in the derived class
- Example: `Student` redefined the method `writeOutput` of its parent class, `Person`
- Could use `super.writeOutput()` to invoke the overridden (parent) method

```
public void writeOutput()
{
   super.writeOutput();
   System.out.println("Student Number : "
                                    studentNumber);
}
```

Overriding Verses Overloading

Overriding

- Same method name

- Same signature
- One method in ancestor, one in descendant

Overloading

- Same method name

- Different signature
- Both methods can be in same class

The `final` Modifier

- Specifies that a method definition cannot be overridden with a new definition in a derived class
- Example:

```
public final void specialMethod()
{
. . .
```

- Used in specification of some methods in standard libraries
- Allows the compiler to generate more efficient code
- Can also declare an entire class to be final, which means it cannot be used as a base class to derive another class

`private` & `public` Instance Variables and Methods

- `private` instance variables from the parent class are not available by name in derived classes
  - » "Information Hiding" says they should not be
  - » use accessor methods to change them, e.g. `reset` for a `Student` object to change the `name` attribute
- `private` methods are **not** inherited!
  - » use `public` to allow methods to be inherited
  - » only helper methods should be declared `private`

What is the "Type" of a Derived class?

- Derived classes have more than one type
- Of course they have the type of the derived class (the class they define)
- They also have the type of every ancestor class
  - » all the way to the top of the class hierarchy
- *All* classes derive from the original, predefined class `Object`
- `Object` is called the *Eve* class since it is the original class for all other classes

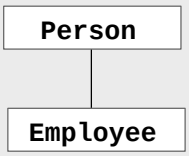
Assignment Compatibility

- **Can** assign an object of a derived class to a variable of any ancestor type

```
Person josephine;
Employee boss = new Employee();
josephine = boss;      OK
```

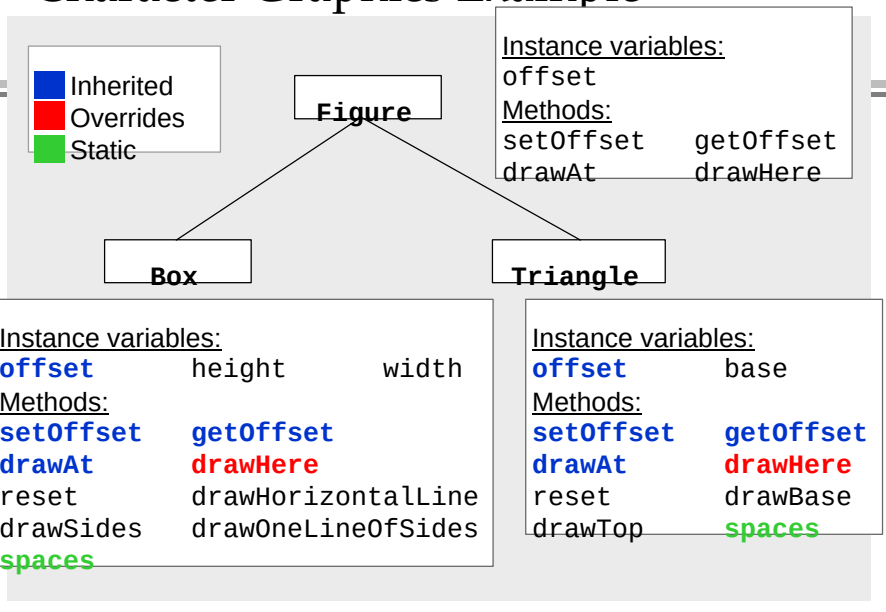
- Can **not** assign an object of an ancestor class to a variable of a derived class type

```
Person josephine = new Person();
Employee boss;
boss = josephine;      Not allowed
```



`Person` is the parent class of `Employee` in this example.

Character Graphics Example



How do Programs Know Where to Go Next?

- Programs normally execute in sequence
- Non-sequential execution occurs with:
  - » selection (if/if-else/switch) and repetition (while/do-while/for) (depending on the test it may not go in sequence)
  - » method calls, which jump to the location in memory that contains the method's instructions and returns to the calling program when the method is finished executing
- One job of the compiler is to try to figure out the memory addresses for these jumps
- The compiler cannot always know the address
  - » sometimes it needs to be determined at run time

Static and Dynamic Binding

- *Binding*: determining the memory addresses for jumps
- *Static*: done at compile time
  - » also called *offline*
- *Dynamic*: done at run time
- Compilation is done *offline*
  - » it is a separate operation done before running a program
- Binding done at compile time is, therefore, static, and
- Binding done at run time is dynamic
  - » also called *late binding*(←→ *early binding*)

Example of Dynamic Binding: General Description

- Derived classes call a method in their parent class which calls a method that is overridden (defined) in each of the derived classes
  - » the parent class is compiled separately and before the derived classes are even written
  - » the compiler cannot possibly know which address to use
  - » therefore the address must be determined (bound) at run time

Dynamic Binding: Specific Example

Parent class: `Figure`
- » Defines methods: `drawAt` and `drawHere`
- » `drawAt` calls `drawHere`

Derived class: `Box extends Figure`
- » Inherits `drawAt`
- » redefines (overrides) `drawHere`
- » Calls `drawAt`
  - – uses the parent's `drawAt` method
  - – which must call this, the derived class's, `drawHere` method

- `Figure` is compiled before `Box` is even written, so the address of `drawHere`(in the derived class `Box`) cannot be known then
  - » it must be determined during run time, i.e. dynamically

Polymorphism

- Using the process of dynamic binding to allow different objects to use different method actions for the same method name
- Originally overloading was considered to be polymorphism
- Now the term usually refers to use of dynamic binding

Summary

- A derived inherits the instance variables & methods of the base class
- A derived class can create additional instance variables and methods
- The first thing a constructor in a derived class normally does is call a constructor in the base class
- If a derived class redefines a method defined in the base class, the version in the derived class *overrides* that in the base class
- Private instance variables and methods of a base class cannot be accessed directly in the derived class
- If A is a derived class of class B, than A is both a member of both classes, A and B
  - » the type of A is both A and B