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
- 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
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
Derived Classes

The keyword `extends` in first line indicates inheritance.

- Creates derived class `Student` from base class `Person`.

A derived class has the instance variables and methods of the base class that it extends.

- The `Person` class has a `name` instance variable so the `Student` class will also have a `name` instance variable.
- Can call the `setName` method with a `Student` object even though `setName` is defined in `Person` and not in `Student`:

```java
public class Student extends Person {
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Warren Peace");
    }
}
```
Extending the Base Class

- A derived class can add instance variables and/or methods to those it inherits from its base class.
- Note that an instance variable for the student number has been added:
  - `Student` has this attribute in addition to `name`, which is inherited from `Person`

```java
private int studentNumber;
```

- Student also adds several methods that are not in Person:
  - `reset`, `getStudentNumber`, `setStudentNumber`, `writeOutput`, `equals`, and some constructors
Overriding Methods

- When a child class has a method that with the same signature as the parent class, the method in the child class *overrides* the one in the parent class.
- This is *overriding*, not overloading.
- Example:
  - Both Person and Student have a writeOutput method with no parameters (same signature).
  - When writeOutput is called with a Student calling object, the writeOutput in Student will be used, not the one in Person.
Overriding Versus 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:
  ```java
  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
Example of Adding Constructor in a Derived Class: Student

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

- 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.

```java
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

```java
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

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

```java
public void writeOutput()
{
    super.writeOutput();
    System.out.println("Student Number : "
                        + studentNumber);
}
```
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
  
  ```java
  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
  
  ```java
  Person josephine = new Person();
  Employee boss;
  boss = josephine;  // Not allowed
  ```

Person is the parent class of Employee in this example.
"Is a" and "Has a" Relationships

- Inheritance is useful for "is a" relationships.
  - A student "is a" person.
  - Student inherits from Person.

- Inheritance is usually **not** useful for "has a" relationships.
  - A student "has a(n)" enrollment date.
  - Add a Date object as an instance variable of Student instead of having Student inherit from Date.

- If it makes sense to say that an object of Class1 "is a(n)" object of Class2, then consider using inheritance.
Character Graphics Example

**Instance variables:**
- offset
- height
- width

**Methods:**
- setOffset
- getOffset
- drawAt
- drawHere
- reset
- drawHorizontalLine
- drawSides
- drawOneLineOfSides
- spaces

**Instance variables:**
- offset
- base

**Methods:**
- setOffset
- getOffset
- drawAt
- drawHere
- reset
- drawBase
- drawTop
- spaces
Abstract Classes

- Cannot create objects of an **abstract class**
  - Example: `Figure` class in character graphics program
  - An abstract class is used as a base for inheritance instead of being used to create objects.

- Abstract classes simplify program design by not requiring you to supply methods that would always be overridden.
  - Example: `drawHere` method is overridden in all classes derived from `Figure`.

- Specify that a method is abstract if you don't want to implement it:
  ```java
  public abstract void drawHere();
  ```

- Any class that has an abstract method must be declared as an abstract class:
  ```java
  public abstract class Figure
  ```
Interfaces

- An **interface** is a type that specifies method headings.
  
  Example:

  ```java
  public interface Writeable
  {
    public String toString();
    public void writeOutput();
  }
  ```

- You can make a method more general by using an interface as a type for a parameter.
  
  » An object of any class that implements the interface (see the next slide) can be passed as the parameter.

  ```java
  public void display(Writeable displayObj)
  {
    displayObj.writeOutput();
  }
  ```
Implementing an Interface

- A class that implements an interface must
  - contain complete definitions for all of the methods specified in the interface
  - be declared as implementing the interface
    
    `implements Interface_Name`

- Any class that implements the `Writeable` interface must have complete definitions of `toString` and `writeOutput`.

- There can be many different classes that implement an interface.
How do Programs Know Where to Go Next?

- Programs normally execute in sequence
- Non-sequential execution occurs with:
  - selection (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*
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