CSI1102: Introduction to Software Design

Chapter 7: Inheritance

Learning objectives: Inheritance

- Another fundamental object-oriented technique is called inheritance, for organizing and creating classes and for promoting reuse
- Understand what the following entails:
  - Deriving new classes from existing classes
  - Creating class hierarchies: a parent and children
  - The `protected` modifier
  - Polymorphism via inheritance
  - Inheritance hierarchies for interfaces

What is Inheritance?

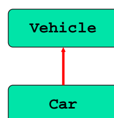
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the ***parent class***, or *superclass*, or *base class*
- The derived class is called the ***child class*** or *subclass*.
- As the name implies, the child inherits characteristics of the parent
  - That is, the child class inherits
    - the **methods** and **data defined** for the parent class

Inheritance: The main Idea

- To tailor a derived class, the programmer can
  - **add** new variables or methods, or
  - **modify** the inherited ones
- *Software reuse* is at the heart of inheritance
  - By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Visualizing Inheritance

- Inheritance relationships often are shown graphically in a *class diagram*, with the arrow pointing to the parent class



Inheritance creates an *is-a relationship*, meaning the child *is a* more specific version of the parent

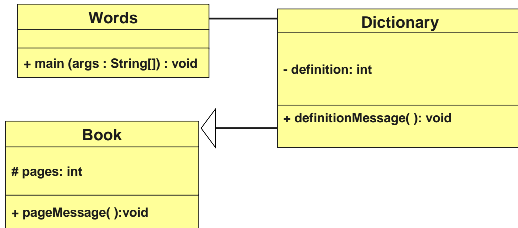
Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Car extends Vehicle
{
   // class contents
}
```

- See `Words.java`, `Book.java` and `Dictionary.java` (pp.384++)

UML diagram showing inheritance



Driver program: Words.java

```
public class Words
{
    //Instantiates a derived class
    public static void main (String[] args)
    {
          Dictionary webster = new Dictionary();

          webster.pageMessage();
          webster.definitionMessage();
    }
}
```


Book.java: The Parent class

```
public class Book
{
  protected int pages = 1000;

  // Prints a message about the pages of this book.
  public void pageMessage ()
  {
   System.out.println ("Number of pages: " + pages);
  }
}
```


Dictionary.java: The Child class

```
public class Dictionary extends Book
  {
      private int definitions = 52500;

      public void definitionMessage()
      {  System.out.println("Number of definitions: "
                             + definitions);
         System.out.println("Definitions per page: "
                             + definitions/pages);
      }
  }
```

```
Number of pages: 1000
Number of definitions: 52500
Definitions per page: 52
```


Visibility modification: The `protected` Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The `protected` Modifier

- The `protected` visibility modifier allows a **member of a parent class to be inherited by a child**
- `protected` visibility provides more encapsulation than `public` does
- However, `protected` visibility is not as tightly encapsulated as `private` visibility
- The details of each modifier are given in Appendix F

Referring to a parent: The **`super`** Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor
- See Word2 example, pp.388-393

The driver program: Words2.java

```
public class Words2
{
    //Instantiates a derived class
    public static void main (String[] args)
    {
     Dictionary2 webster = new Dictionary2(1500, 52500);

     webster.pageMessage();
     webster.definitionMessage();
    }
}
```

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35
```


The parent class: Book2.java

```
public class Book2
{
   protected int pages;

   public Book2(int numPages)
   {
        pages = numPages;
   }

   // print a message
   public void pageMessage()
   {
        System.out.println("Number of pages: " + pages);
   }
}
```


The child class: Dictionary2.java

```
public class Dictionary2 extends Book2
{
 private int definitions;

 public Dictionary2(int numPages, int numDefinitions)
 {
   super(numPages);

   definitions = numDefinitions;
 }

 public void definitionMessage()
 {
   System.out.println("Number of definitions: " + definitions);
   System.out.println("Definitions per page:" + definitions/pages);
 }
}
```


More about The **`super`** Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can be used to reference other variables and methods defined in the parent's class

Single vs. Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can **have only one parent class**
- *Multiple inheritance,* in some other languages, allows a class to be derived from two or more classes, inheriting the members of all parents
  - Collisions, such as the same variable name in two parents, have to be resolved
  - In most cases, the use of **interfaces** gives us **aspects** of multiple inheritance without the overhead

Overriding Methods: Redefining

- A child class can *override* the definition of an inherited method in favor of its own
  - That is, a child can redefine a method that it inherits from its parent
- The new method **must** have the **same signature** as the parent's method, but **can have a different body**
- The **type** of the object executing the method determines which version of the method is invoked

Overriding methods: Messages.java (p.392)

```
public class Messages
    {
        public static void main (String[] args)
        {
            Thought parked = new Thought();
            Advice dates = new Advice();

            parked.message();
            dates.message();
        }
    }
```

```
I feel fine

Warning: Time is shrinking
```


Thought.java

```
public class Thought
    {
        public void message()
        {
            System.out.println("I feel fine");
            System.out.println();
        }
    }
```


Advice.java

```
public class Advice
{
    public void message()
    {
        System.out.println("Warning: Time is shrinking");
        System.out.println();
    }
}
```

```
public class Advice extends Thought
{
    public void message()
    {
        System.out.println("Warning: Time is shrinking");
        System.out.println();
        super.message();
    }
}
```


Overriding Methods and Variables

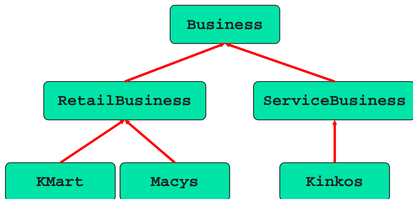
- Note that a parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data (called *shadowing variables*), but generally **it should be avoided**

Overloading vs. Overriding: Not the same

- Overloading deals with **multiple methods** in the same class with the **same name** but **different signatures**
- Overriding deals with **two methods**, one in a **parent** class and one in a **child** class, that have the **same signature**
- Overloading lets you define a similar operation in different ways for different **data**
- Overriding lets you define a similar operation in different ways for different **object types**

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies: Some definitions

- Two children of the same parent are called *siblings*
- Good class design puts all common features as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
  - The inheritance mechanism is *transitive.*
- That is, a **child class inherits from all its ancestor classes**
- There is no single class hierarchy that is appropriate for all situations
- Class hierarchies often need to be extended and modified to keep up with changes

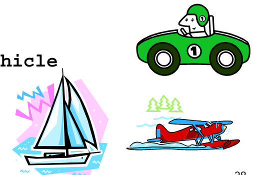
The `Object` Class: Included in `java.lang`

- All classes are derived from the `Object` class
- The `Object` class is the ultimate root of all class hierarchies
- The `Object` class contains a few useful methods, which are inherited by all classes
  - For example, the `toString` method is defined in the `Object` class
  - That's why the `println` method can call `toString` for any object that is passed to it – all objects are guaranteed to have a `toString` method via inheritance

Abstract Classes

- An abstract class is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract

```
abstract public class vehicle
```



What are Abstract Classes?

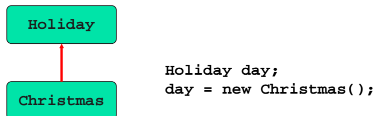
- An abstract class often contains abstract methods with no definitions (like an interface does), though it doesn't need to
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- An abstract class typically contains non-abstract methods with method bodies, further distinguishing abstract classes from interfaces
- A class declared as abstract does not need to contain abstract methods

What are Abstract Classes?

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as
  - final (because it must be overridden) or
  - static (because it has no definition yet)
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is too general to instantiate
  - ***E.g. Vehicle, FuelConsumption***
  - ***E.g. Employee, BenefitsCalculation***

Indirect use of class members: References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance
- For example, if the `Holiday` class is used to derive a child class called `Christmas`, then a `Holiday` reference could be used to point to a `Christmas` object




```
Holiday day;
day = new Christmas();
```


References and Inheritance: Widening versus narrowing

- Widening conversion:
  - Assigning a predecessor object to an ancestor reference
  - Performed by simple assignment
- Narrowing conversion:
  - Assigning an ancestor object to a predecessor reference
  - Performed with a cast

Indirect Use of Non-inherited Members

- An inherited member can be referenced directly by name in the child class, as if it were declared in the child class
- But even if a method or variable is not inherited by a child, it can still be accessed indirectly through parent methods
- See `FoodAnalysis.java` (page 403)
- See `FoodItem.java` (page 404)
- See `Pizza.java` (page 405)

FoodAnalysis.java

```
public class FoodAnalysis
{
        public static void main (String[] args)
        {
                Pizza special = new Pizza(275);

                System.out.println("Calories per serving: " +
                        special.caloriesPerServing());
        }
    }
```

```
Calories per serving: 309
```


FoodItem.java

```
public class FoodItem
    {
        final private int CAL_PER_GRAM = 9;
        private int fatGrams;
        protected int servings;

        public FoodItem(int numFatGrams, int numServings)
        {
                fatGrams = numFatGrams;
                servings = numServings;
        }

        private int calories()
        {
                return fatGrams*CAL_PER_GRAM;
        }

        public int caloriesPerServing()
        {
                return (calories()/servings);
        }
    }
```


Pizza.java

```
public class Pizza extends FoodItem
    {
        public Pizza(int fatGrams)
        {
                super(fatGrams, 8);
        }
    }
```

Polymorphism: Having many forms

- A reference can be *polymorphic*, which can be defined as "having many forms"
  - A polymorphic reference variable can refer to different types of objects during execution
  - Polymorphic references are resolved at run time; this is called *dynamic binding*
- Careful use of polymorphic references can lead to elegant, robust software designs

```
Mammal pet;
Horse myhorse = new Horse(); // Horse derived from Mammal
                             // Horse is-a Mammal
pet = myhorse;
```


Polymorphism via Inheritance

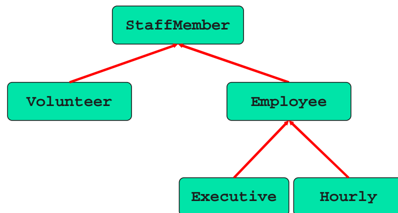
- Suppose the `Holiday` class has a method called `celebrate`, and the `Christmas` class overrides it
- Now consider the following invocation:

```
day.celebrate();
```

- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Christmas` object, it invokes the `Christmas` version

Polymorphism via Inheritance

- Consider the following class hierarchy:



- **Now consider the task of paying all employees**

Firm.java

```
public class Firm
   {
         public static void main (String[] args)
         {
                  Staff personnel = new Staff();
                  personnel.payday();
         }
   }
```

```
Name: Sam
Phone: 555-3456
Paid: 3341.07
Name: Joe
Phone: 555-1432
Paid: 1000.0
Name: Sue
Phone: 555-6567
Thanks!
Name: Ann
Phone: 555-7876
Thanks!
```


Staff.java

```
public class Staff
   {
         private StaffMember[] staffList;

         public Staff()
         {
                  staffList = new StaffMember[4];

                  staffList[0] = new Executive("Sam", "555-3456", 2341.07);
                  staffList[1] = new Employee("Joe", "555-1432", 1000.00);
                  staffList[2] = new Volunteer("Sue", "555-6567");
                  staffList[3] = new Volunteer("Ann", "555-7876");
         }
```

Continued...

Staff.java (cont.)

```
...
   public void payday()
         {
                  double amount;

                  for (int count = 0; count < staffList.length; count++)
                  {
                           System.out.println(staffList[count]);
                           amount = staffList[count].pay();
                           if (amount == 0)
                                    System.out.println("Thanks!");
                           else
                                    System.out.println("Paid: " + amount);
                  }
         }
   }
```

StaffMember.java

```
abstract public class StaffMember
    {
        protected String name;
        protected String phone;

        public StaffMember(String eName, String ePhone)
        {
            name = eName;
            phone = ePhone;
        }

        public String toString()
        {
            String result = "Name: " + name + "\n";
            result += "Phone: " + phone;

            return result;
        }

        public abstract double pay();

    }
```


Volunteer.java

```
public class Volunteer extends StaffMember
    {
        public Volunteer (String eName, String ePhone)
        {
            super(eName, ePhone);
        }

        public double pay()
        {
            return 0.0;
        }
    }
```


Employee.java

```
public class Employee extends StaffMember
{
    protected double payRate;

    public Employee(String eName, String ePhone, double rate)
    {
        super(eName, ePhone);
        payRate = rate;
    }


    public double pay()
    {    return payRate;   }

    public String toString()
    {
        String result = super.toString();

        return result;
    }
}
```


Executive.java

```
public class Executive extends Employee
    {
        private double bonus;

        public Executive(String eName, String ePhone, double rate)
        {
            super(eName, ePhone, rate);
            bonus = 1000;
        }

        public double pay()
        {
            double payment = super.pay() + bonus;

            return payment;
        }
    }
```

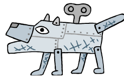

Interface Hierarchies

- **Inheritance can be applied to interfaces as well as to classes**
  - One interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the ancestor and child interfaces
- All members of an interface are public
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

Polymorphism via Interfaces

- An interface name can be used to declare an object reference variable
- Interfaces allow polymorphic references in which
  - the method that is invoked is determined by the object being referenced

Speakers, Philosophers and Dogs



```
public interface Speaker
{
   public void speak();
   public void announce (String str);
}
```

Assume Classes `Philosopher` and `Dog` both implement the Speaker interface:

```
Speaker guest;
guest = new Philosopher();
guest.speak();              speak method in Philosopher class
guest = new Dog();
guest.speak;                speak method in Dog class
```


Inheritance and GUIs: More in Chapter 9

- An applet is an excellent example of inheritance
- Recall that when we define an applet, we extend the `Applet` class or the `JApplet` class
- The `Applet` and `JApplet` classes already handle all the details about applet creation and execution, including the interaction with a Web browser
- When we define certain methods, such as the `paint` method of an applet, we are actually overriding a method defined in the `Component` class, which is ultimately inherited into the `Applet` class or the `JApplet` class

Summary: Chapter 7

- Understand what the following entails:
  - Inheritance
  - Deriving new classes from existing classes
  - Creating class hierarchies: a parent and children
  - The `protected` modifier
  - Polymorphism via inheritance
  - Inheritance hierarchies for interfaces