Module 12 - Derived Classes

General Notes

Module Content

ZyBooks

Inheritance in Object-Oriented Programming

Introduction

- Inheritance is a fundamental Object-Oriented Programming (OOP) technique for organizing classes that are related in a hierarchy.
- A derived class is also called a subclass or child class, and a base class is also called a superclass or parent class.
- This technique allows derived classes to use the code already defined in their base class and inherit their properties.

Key Topics

- `protected` Access Modifier
- Constructors and `super`
- Overloading and Overriding methods
- Creating child class objects

Derived Class Concept

- One class is often similar to another class but with some additions or variations.
- Ex: A store inventory system might use a class called `GenericItem` that has `itemName` and `itemQuantity` data members. But for produce (fruits and vegetables), a `ProduceItem` class with data members `itemName`, `itemQuantity`, and `expirationDate` may be desired.

Inheritance

- A **derived class** (or **subclass**) is a class that is derived from another class, called a **base class** (or **superclass**).

- An object declared of a derived class type has access to all the public members of the derived class as well as the public members of the base class.
- A derived class is declared by placing the keyword `extends` after the derived class name, followed by the base class name.
- Ex: `class DerivedClass extends BaseClass { ... }`.

Inheritance Example:

GenericItem.java

```
public class GenericItem {
   private String itemName;
   private int itemQuantity;

   public void setName(String newName) {
      itemName = newName;
   }

   public void setQuantity(int newQty) {
      itemQuantity = newQty;
   }

   public void `printItem()` {
      System.out.println(itemName + " " + itemQuantity);
   }
}
```

ProduceItem.java

```
public class ProduceItem extends GenericItem {
   private String expirationDate;

   public void setExpiration(String newDate) {
      expirationDate = newDate;
   }

   public String `getExpiration()` {
      return expirationDate;
   }
}
```

Protected Access Modifier

- The `protected` access modifier allows the derived class to access the base class's members that are marked `protected`.
- The `protected` members can be accessed in the derived class and any other classes in the same package.

Constructors and Super

- The `super()` keyword is used to call the constructor of the base class.
- If the base class constructor requires arguments, the `super()` keyword must be passed those arguments.
- If the derived class does not explicitly call a base class constructor, the default constructor of the base class will be called automatically.

Overloading and Overriding Methods

- Inheritance allows the derived class to override or overload the base class's methods.
- Overriding a method allows the derived class to replace the base class's implementation with its own.
- Overloading a method allows the derived class to provide a different implementation of the same method.

Creating Child Class Objects

- To create an object of a child class, you can use the `new` operator followed by the child class name.
- The child class constructor is called automatically when an object is created.

Inheritance Scenarios

- A derived class can serve as a base class for another class.
- A class can serve as a base class for multiple derived classes.
- A class can only be derived from one base class directly.

Example: Business and Restaurant

- The `Restaurant` class is derived from the `Business` class and adds a `rating` private field with a getter and setter.
- Inheritance allows the `Restaurant` class to use the code already defined in the `Business` class.
- An object of the `Restaurant` class can access the public members of the `Business` class through inheritance.

Conclusion

- Inheritance is a powerful OOP technique that allows code reuse and efficient organization of related classes in a hierarchy.
- Derived classes can use the code already defined in their base class and inherit their properties.
- Inheritance also allows for polymorphism, which enables objects of different derived classes to be treated as objects of the same base class.
- Inheritance can make code more maintainable and easier to read by organizing classes into a logical hierarchy.
- The `protected` access modifier allows derived classes to access protected members of the base class, which can be useful for implementing certain features.
- Constructors and the `super` keyword are used to call the base class constructor and initialize base class members.
- Overriding and overloading methods allow derived classes to provide their own implementation of methods defined in the base class.
- Creating child class objects is straightforward, simply using the `new` operator followed by the child class name.
- Inheritance can be used in a variety of scenarios, such as when creating a hierarchy of related classes or when defining specialized versions of existing classes.

Further Reading

- Oracle's Java tutorials on inheritance

Access by Members of Derived Classes

In Java, the members of a derived class have access to the public members of the base class but not to the private members of the base class. Adding a member method that attempts to access private members of the base class yields a compiler error. Protected members, on the other hand, are accessible to derived classes and all classes in the same package but not to anyone else.

Member Access

- Members of a derived class can access public members of the base class but not private members of the base class.
- Attempting to access private members of the base class in a derived class member method results in a compiler error.
- Private members are accessible only by self, while protected members are accessible by self, derived classes, and other classes in the same package.

Protected Member Access

- The protected access specifier provides access to derived classes and all classes in the same package but not to anyone else.
- Protected members are private to everyone else.
- In the example provided, the member called `name` is specified as protected and is accessible anywhere in the derived class.

Protected Access Specifier Example:

Business.java

```
public class Business{
   protected String name;    // Member accessible by self and derived
classes
   private String address;   // Member accessible only by self

   public void `printMembers()` { // Member accessible by anyone
      // Print information ...
   }
}
```

Restaurant.java

```
public class Restaurant extends Business{
   private int rating;

   public void `displayRestaurant()` {
      // Attempted accesses
      `printMembers()`              // OK
      name = "Gyro Hero";          // OK    ("protected" above made this
possible)
      address = "5 Fifth St";     // ERROR
   }

   // Other class members ...
}
```

InheritanceAccessEx.java

```
public class InheritanceAccessEx {
   public static void main(String[] args) {
      Business business = new `Business()`
      Restaurant restaurant = new `Restaurant()`

      // Attempted accesses
      business.printMembers();          // OK
      business.name = "Gyro Hero";      // OK (protected also applies
to other classes in the same package)
      business.address = "5 Fifth St";  // ERROR

      restaurant.printMembers();        // OK
      restaurant.name = "Gyro Hero";    // OK (protected also applies
to other classes in the same package)
      restaurant.rating = 5; // ERROR

      // Other instructions ...
   }
}
```

Access specifiers

Figure 12.2.2 shows the access specifiers used in Java programming language. Protected access specifier allows access to the members by derived classes and classes in the same package but not by others.

Class Definitions

The keyword `public` in a class definition specifies a class's visibility in other classes in the program.

- A class defined as `public` can be used by every class in the program regardless of the package in which either is defined.
- A class with **no specifier** can be used only in other classes within the same package, known as **package-private**.

Access Specifiers for Class Members

Specifier	Description
`private`	Accessible by self.
`protected`	Accessible by self, derived classes, and other classes in the same package.
`public`	Accessible by self, derived classes, and everyone else.
no specifier	Accessible by self and other classes in the same package.

Further Reading

- More on access specifiers
  from Oracle's Java tutorials

Notes on Overriding Member Methods

Overriding

- When a derived class defines a member method that has the same name and parameters as a base class's method, the member method is said to override the

base class's method.

- The `@Override` annotation is placed above a method that overrides a base class method so the compiler verifies that an identical base class method exists.
- An **annotation** is an optional command beginning with the "@" symbol that can
  provide the compiler with information that helps the compiler detect errors better.
- The `@Override` annotation causes the compiler to produce an error when a programmer mistakenly specifies parameters that are different from the parameters of the method that should be overridden or misnames the overriding method.
- Good practice is to always include an `@Override` annotation with a method that is meant to override a base class method.

Overriding vs. Overloading

- Overloading is different from overriding.
- In overloading, methods with the same name must have different parameter types, number of parameters, or return values.
- In overriding, a derived class member method must have the same parameter types, number of parameters, and return value as the base class member method with the same name.
- Overloading is performed if derived and base member methods have different parameter types; the member method of the derived class does not hide the member method of the base class.

Calling Overridden Methods

- An overriding method can call the overridden method by using the `super` keyword. Ex: `super.getDescription()`.
- The `super` keyword is a reference variable used to call the parent class's methods or constructors.

Method Calling Overriden Method of Base Class

```
public class Restaurant extends Business{
   @Override
   public String `getDescription()` {
      return super.getDescription() + "\n  Rating: " + rating;
   }
}
```

- The above example shows how the Restaurant's `getDescription()` method overrides the Business's `getDescription()` method.
- The `super.getDescription()` is used to call the base class's `getDescription()` method before adding the `"\n Rating: " + rating` to it.

Common Error

- A common error is to leave off `super` when wanting to call a base class method.
- Without the use of the `super` keyword, the call to `getDescription()` refers to itself (a *recursive* call), so `getDescription()` would call itself, which would call itself, etc., never actually printing anything.

Conclusion

- Overriding is when a derived class defines a member method that has the same name and parameters as a base class's method.
- Overriding is different from overloading where methods with the same name must have different parameter types, number of parameters, or return values.
- An overriding method can call the overridden method by using the `super` keyword.
- A common error is to leave off `super` when wanting to call a base class method.

The Object class

The built-in **Object class** serves as the base class for all other classes and does not have a base class. All classes, including user-defined classes, are

derived from Object and implement Object's methods. In the following discussion, note the subtle distinction between the term "Object class" and the generic term "object", which can refer to the instance of any class. Two common methods defined within the Object class are `toString()` and `equals()`

- The `toString()` method returns a String representation of the Object. By default, `toString()` returns a String containing the object's class name followed by the object's hash code in hexadecimal form.
  - Ex: `java.lang.Object\@372f7a8d`.
- The `equals(otherObject)` method compares an `Object` to `otherObject` and returns `true` if both variables reference the same object. Otherwise, `equals()` returns `false`. By default, `equals()` tests the equality of the two `Object` references, not the equality of the Objects' contents.

Business class is derived from Object

```
public class Business {
   protected String name;
   protected String address;

   void setName(String busName) {
      name = busName;
   }

   void setAddress(String busAddress) {
      address = busAddress;
   }

   String getDescription() {
      return name + " -- " + address;
   }
}
```

Overriding `toString()` in the base class

1. The figure below shows a Business class that overrides Object's `toString()` method and returns a String containing the business name and address.
2. The Restaurant class derives from Business but does not override `toString()`.

3. So when a Restaurant object's `toString()` method is called, the Business class's `toString()` method executes.

Base class `Business` overrides `toString()`

Business.java

```
public class Business {
   protected String name;
   protected String address;

   void setName(String busName) {
      name = busName;
   }

   void setAddress(String busAddress) {
      address = busAddress;
   }

   @Override
   public String toString() {
      return name + " -- " + address;
   }
}
```

Restaurant.java

```
public class Restaurant extends Business {
   private int rating;

   public void setRating(int userRating) {
      rating = userRating;
   }

   public int getRating() {
      return rating;
   }
}
```

The `toString()` method is called automatically by the compiler when an object is concatenated to a string or when `print()` or `println()` is called.

- Ex: `System.out.println(someObj)` calls `someObj.toString()` automatically.

Overriding `toString()` in the derived class

1. Both the base class Business and derived class Restaurant override `toString()` in the figure below.
2. The Restaurant `toString()` uses the `super` keyword to call the base class `toString()` to get a string with the business name and address.
3. Then `toString()` concatenates the rating and returns a string containing the name, address, and rating.

Derived Class `Restaurant` overrides `toString()`

Business.java

```
public class Business {
   protected String name;
   protected String address;

   void setName(String busName) {
      name = busName;
   }

   void setAddress(String busAddress) {
      address = busAddress;
   }

   @Override
   public String toString() {
      return name + " -- " + address;
   }
}
```

Restaurant.java

```
public class Restaurant extends Business {
   private int rating;

   public void setRating(int userRating) {
      rating = userRating;
   }

   public int getRating() {
      return rating;
   }

   @Override
   public String toString() {
      return super.toString() + ", Rating: " + rating;
   }
}
```

Exploring further

- Oracle's Java Object class specification
- Oracle's Java class hierarchy

Polymorphism

Definition and Forms

- **Polymorphism:** determining program behavior based on data types.
- **Compile-time polymorphism:** Method overloading. The compiler determines which method to call based on method's arguments.
- **Runtime polymorphism:** determination made while program is running, compiler cannot make the determination.

Runtime Polymorphism Example

This is a runtime polymorphism scenario involving derived classes where programmers create a collection of objects of both base and derived class types.

Example statement:

```
ArrayList<GenericItem> inventoryList = new ArrayList<GenericItem>();
```

- Declares an `ArrayList` that can contain references to objects of type `GenericItem` or `ProduceItem`.
- `ProduceItem` derives from `GenericItem`.
- `ProduceItem` is a specialized version of `GenericItem`, any object that is a `ProduceItem` is a `GenericItem`.

Runtime polymorphism.

The JVM can dynamically determine the correct method to call based on the object's type.

GenericItem.java:

```
public class GenericItem {
   public void setName(String newName) {
      itemName = newName;
   }

   public void setQuantity(int newQty) {
      itemQuantity = newQty;
   }

   public void printItem() {
      System.out.println(itemName + " " + itemQuantity);
   }

   protected String itemName;
   protected int itemQuantity;
}
```

ProduceItem.java:

```
public class ProduceItem extends GenericItem { // ProduceItem derived
from GenericItem
   public void setExpiration(String newDate) {
      expirationDate = newDate;
   }

   public String getExpiration() {
      return expirationDate;
   }

   @Override
   public void printItem() {
      System.out.println(itemName + " " + itemQuantity
                                    + " (Expires: " + expirationDate +
")");
   }

   private String expirationDate;
}
```

ItemInventory.java:

```
import java.util.ArrayList;

public class ItemInventory {
   public static void main(String[] args) {
      GenericItem genericItem1;
      ProduceItem produceItem1;
      ArrayList<GenericItem> inventoryList = new ArrayList<GenericItem>
(); // Collection of "Items"
      int i;
// Loop index

      genericItem1 = new GenericItem();
      genericItem1.setName("Smith Cereal");
      genericItem1.setQuantity(9);

      produceItem1 = new ProduceItem();
      produceItem1.setName("Apple");
      produceItem1.setQuantity(40);
      produceItem1.setExpiration("May 5, 2012");

      genericItem1.printItem();
      produceItem1.printItem();

      // More common: Collection (e.g., ArrayList) of objs
      // Polymorphism -- Correct  printItem()  called
      inventoryList.add(genericItem1);
      inventoryList.add(produceItem1);
      System.out.println("\nInventory: ");
      for (i = 0; i < inventoryList.size(); ++i) {
         inventoryList.get(i).printItem(); // Calls correct printItem()
      }
   }
}
```

- Uses a Java feature relating to **derived/base class reference conversion** wherein a reference to a derived class can be converted to a reference to the base class (without explicit casting).
    - Unlike converting a `double` to an `int` which produces an error unless explicitly cast.

- `inventoryList` is an `ArrayList` of `GenericItem` references.
- The Java virtual machine automatically performs runtime polymorphism to determine the correct method to call based on the actual object type to which the variable (or element) refers.

Conclusion

- Polymorphism is the determination of program behavior based on data types.
- Method overloading is a form of compile-time polymorphism and runtime polymorphism is the determination made while the program is running.
- Runtime polymorphism scenario involves derived classes, where programmers create a collection of objects of both base and derived class types.
- When printing the ArrayList's contents, the program knows which printItem() to call based on the actual object type to which the variable (or element) refers, using Java's feature of runtime polymorphism.
- The Arraylist called itemList can hold BaseItem objects and any objects that are created from subclasses of BaseItem, such as DerivedItem.
- A way to think about this is that DerivedItem is a specialized version of BaseItem.

Additional Information:

- Polymorphism is a fundamental concept in object-oriented programming that allows code to be more flexible and reusable.
- In Java, polymorphism can be achieved through method overloading, method overriding, and using abstract classes and interfaces.
- Method overloading is a form of compile-time polymorphism, while method overriding is a form of runtime polymorphism.
- By using polymorphism, programmers can write code that can work with objects of different types without knowing the exact type of the object at compile time.

Further Reading

- More on Polymorphism
  from Oracle's Java tutorials.

ArrayLists of Objects

Overview

This section discusses how to create an `ArrayList` to hold **unrelated objects**, although from a design perspective, this is **not** recommended.

While it is possible to create an `ArrayList` of objects of different class types, managing such a collection could be very difficult.

Creating an ArrayList of Objects

The Object class is the top-level superclass for all Java classes. Therefore, to create an ArrayList that holds unrelated objects, you could create an ArrayList to hold Object objects and then add any other type of object to the list. This approach should be avoided, however, as it is difficult to manage.

Using Polymorphism to Create a Collection of Objects

Since all classes are derived from the Object class, programmers can use runtime polymorphism to create a collection (such as an ArrayList) of objects of various class types and perform operations on the elements.

Example: Business Class

The example program below uses the Business class and other built-in classes to create and output a single ArrayList of differing types. The Business class has two protected instance variables, `name` and `address`, and two constructors - a default constructor and a constructor that takes two parameters.
The `toString()` method is also overridden to display the business name and address.

Business.java

```java
public class Business {
   protected String name;
   protected String address;

   public Business() {}

   public Business(String busName, String busAddress) {
      name = busName;
      address = busAddress;
   }

   @Override
   public String toString() {
      return name + " -- " + address;
   }
}
```

Another Example

```
import java.util.ArrayList;

public class ArrayListPrinter {

   // Method prints an ArrayList of Objects
   public static void printArrayList(ArrayList<Object> objList) {
      int i;

      for (i = 0; i < objList.size(); ++i) {
         System.out.println(objList.get(i));
      }
   }

   public static void main(String[] args) {
      ArrayList<Object> objList = new ArrayList<Object>();

      // Add new instances of various classes to objList
      objList.add(new Object());
      objList.add(12);
      objList.add(3.14);
      objList.add(new String("Hello!"));
      objList.add(new Business("ACME", "5 Main St"));

      // Print list of Objects
      printArrayList(objList);
   }
```

Method Invocation on Collection of Object Elements

When operating on a collection of Object elements, a method may only invoke the methods defined by the base class (i.e., the Object class).

- Thus, calling the `toString()` method on an element of an ArrayList of Objects called `objList`, such as `objList.get(i).toString()`, is valid because the Object class defines the `toString()` method.
- However, calling a method like the Integer class's `intValue()` method on the same element (i.e., `objList.get(i).intValue()`) would result in a compiler error even if that particular element is an Integer object.

Resources

- Oracle's Java Object class specification
- More on Polymorphism
  from Oracle's Java tutorials

Is-a versus Has-a Relationships

Composition

Composition is the idea that one object may be made up of other objects.

- For example, a `MotherInfo` class may be made up of objects like `firstName`, `childrenData`, etc.
- Defining the `MotherInfo` class does not involve inheritance, but rather just composing the sub-objects in the class.
- The ***has-a*** relationship describes this type of composition.
- A `MotherInfo` object ***has a*** `String` object and ***has a*** `ArrayList` of `ChildInfo` objects.

Composition Example

```
public class ChildInfo {
   public String firstName;
   public String birthDate;
   public String schoolName;
}

public class MotherInfo {
   public String firstName;
   public String birthDate;
   public String spouseName;
   public ArrayList<ChildInfo> childrenData;
}
```

Inheritance

Inheritance refers to the idea that a subclass can inherit the characteristics of a superclass.

- For example, a `MotherInfo` class may inherit the characteristics of a `PersonInfo` superclass, which has the attributes of `firstName` and `birthdate`.
- The ***is-a*** relationship describes this type of inheritance.
- A `MotherInfo` object ***is a*** kind of `PersonInfo`.
- The `MotherInfo` class thus inherits from the `PersonInfo` class.

Inheritance Example

```
public class PersonInfo {
   public String firstName;
   public String birthdate;
}

public class ChildInfo extends PersonInfo {
   public String schoolName;
}

public class MotherInfo extends PersonInfo {
   public String spousename;
   public ArrayList<ChildInfo> childrenData;
}
```

UML Diagrams

- Programmers commonly draw class inheritance relationships using **Unified Modeling Language (UML)** notation.
- UML diagrams help to visualize and design class hierarchies and relationships.

Conclusion

- In summary, inheritance and composition are two fundamental concepts in object-oriented programming.
- Understanding the differences between the 'is-a' and 'has-a' relationships can help to improve the design and organization of programs.
- UML diagrams provide a useful tool for visualizing and communicating class hierarchies and relationships.

Further Readings

- IBM: UML basics