Chapter 12 – Aggregation, Composition, and Inheritance

- Composition
- Aggregation
- UML Class Diagram for Composition and Aggregation
- Car Dealership Program
- Inheritance Overview
- Inheritance Example - People in a Department Store
- Inheritance Terminology
- UML Class Diagrams for Inheritance Hierarchies
- Benefits of Inheritance
- Inheritance For a Superclass's private Instance Variables
- Using super to Call Superclass Constructor
- Calling a Superclass's Method from Within a Subclass
- Default Call to Superclass Constructor
- Method Overriding
- The final Access Modifier
- Aggregation, Composition, and Inheritance Compared
- Aggregation, Composition, and Inheritance Combined
- Card Game Program

Composition

- Prior to this chapter, all of our objects have been relatively simple, so we've been able to describe each object with just a single class.

- But for an object that's more complex, you should consider breaking up the object into its constituent parts and defining one class as the whole and other classes as parts of the whole. When the whole class is the exclusive owner of the parts classes, then that class organization is called a composition.
The concept of composition is not new; that's what we do to describe complex objects in the real world:

- Every living creature and most manufactured products are made up of parts. Often, each part is a subsystem that is itself made up of its own set of subparts. Together, the whole system forms a composition hierarchy.
- Note the human body composition hierarchy on the next slide.
- Remember that with a composition relationship, a component part is limited to just one owner at a time. For example, a heart can be in only one body at a time.

A partial composition hierarchy for the human body:
Aggregation

- In a composition hierarchy, the relationship between a containing class and one of its part classes is known as a *has-a* relationship. For example, each human body *has* a brain and *has* a heart.
- There's another has-a relationship, called *aggregation*, which is a weaker form of composition. With aggregation, one class is the whole and other classes are parts of the whole (as with composition), but there is no additional constraint that requires parts to be exclusively owned by the whole.
- An aggregation example where the parts are not exclusively owned by the whole –
  - You can implement a school as an aggregation by creating a whole class for the school and part classes for the different types of people who work and study at the school.
  - The people aren’t exclusively owned by the school because a person can be part of more than one aggregation.
  - For example, a person can attend classes at two different schools and be part of two school aggregations. The same person might even be part of a third aggregation, of a different type, like a household aggregation.

Car Dealership Program

- Suppose you're trying to model a car dealership with a computer program. Since the car dealership is made from several distinct non-trivial parts, it's a good candidate for being implemented with composition and aggregation relationships.
- The car dealership's "non-trivial parts" are a group of cars, a sales manager, and a group of sales people.
- In implementing the program, define four classes:
  - The *Car*, *Manager*, and *SalesPerson* classes implement the car dealership's non-trivial parts.
  - The *Dealership* class contains the three parts classes.
- For each of the three class relationships, *Dealership-Car*, *Dealership-SalesPerson*, and *Dealership-Manager*, is the relationship composition or aggregation?
Universal Modeling Language (UML) class diagrams show the relationships between a program's classes:

- A solid line between two classes represents an **association** – a relationship between classes.
- On an **association line**, a solid diamond indicates a composition relationship, and a hollow diamond indicates an aggregation relationship. The diamond goes next to the container class.
- The labels on the association lines are called **multiplicity values**. They indicate the number of object instances for each of the two connected classes.
- The * multiplicity value represents any size number, zero through infinity.

Car Dealership Program

- To implement a program that uses aggregation and composition:
  - Define one class for the whole and define separate classes for each of the parts.
  - For a class that contains another class, declare an instance variable inside the containing class such that the instance variable holds a reference to one or more of the contained class's objects.
  - Typically, for association lines with * multiplicity values, use an **ArrayList** to implement the instance variable associated with the asterisked class.
  - If two classes have an aggregation relationship with non-exclusive ownership, then store the contained class's object in an instance variable in the containing class, but also store it in another variable outside of the containing class, so the object can be added to another aggregation and have two different "owners."
  - If two classes have a composition relationship, then store the contained class's object in an instance variable in the containing class, but do not store it elsewhere. That way, the object can have only one "owner."
import java.util.ArrayList;

public class Dealership
{
    private String company;
    private Manager manager;
    private ArrayList<SalesPerson> people = new ArrayList<SalesPerson>();
    private ArrayList<Car> cars = new ArrayList<Car>();

    public Dealership(String company, Manager manager)
    {
        this.company = company;
        this.manager = manager;
    }

    public void addCar(Car car)
    {
        cars.add(car);
    }

    public void addPerson(SalesPerson person)
    {
        people.add(person);
    }

    public void printStatus()
    {
        System.out.println(company + "\t" + manager.getName());
        for (SalesPerson person : people)
            System.out.println(person.getName());
        for (Car car : cars)
            System.out.println(car.getMake());
    }
} // end Dealership class
Car Dealership Program

/***************************************************************************/
/* Car.java */
/* Dean & Dean */
/* This class implements a car. */
/***************************************************************************/

public class Car {
    private String make;

    public Car(String make) {
        this.make = make;
    }

    public String getMake() {
        return make;
    }
}

Car Dealership Program

/***************************************************************************/
/* Manager.java */
/* Dean & Dean */
/* This class implements a car dealership sales manager. */
/***************************************************************************/

public class Manager {
    private String name;

    public Manager(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

} // end Manager class
Car Dealership Program

//**************************************************************************************************
/* SalesPerson.java */
/* Dean & Dean */
/* */
/* This class implements a car sales person */
//**************************************************************************************************

public class SalesPerson {
    private String name;
    private double sales = 0.0; // sales to date

    //**********************************************************
    public SalesPerson(String name) {
        this.name = name;
    }
    //**********************************************************

    public String getName() {
        return name;
    }
} // end SalesPerson class

Car Dealership Program

//**************************************************************************************************
/* DealershipDriver.java */
/* Dean & Dean */
/* */
/* This class demonstrates car dealership composition. */
//**************************************************************************************************

public class DealershipDriver {
    public static void main(String[] args) {
        Manager ahmed = new Manager("Ahmed Abdi");
        SalesPerson ash = new SalesPerson("Ash Lawrence");
        SalesPerson jeffrey = new SalesPerson("Jeffrey Leung");
        Dealership dealership = new Dealership("OK Used Cars", ahmed);

        dealership.addPerson(ash);
        dealership.addPerson(jeffrey);
        dealership.addCar(new Car("GMC");
        dealership.addCar(new Car("Yugo");
        dealership.addCar(new Car("Dodge");
        dealership.printStatus();
    } // end main
} // end DealershipDriver class
Inheritance Overview

- There are different ways that classes can be related. We've covered aggregation and composition, where one class is the whole and other classes are parts of the whole. Inheritance is another type of class relationship.

- Suppose you're in charge of designing cars for a car manufacturer:
  - You could create independent design blueprints for the manufacturer's five car models, but to save time, it's better to first make a master blueprint that describes features that are common to all the models.
  - Then make additional blueprints, one for each model. The additional blueprints describe features that are specific to the individual models.

- Creating a more specific blueprint that's based on an existing master blueprint is analogous to the concept of inheritance, in which a new class is derived from an existing class.

- It's called inheritance because the new class inherits/borrows all the features (data and methods) of the existing class.

- Inheritance is a very important feature of Java (and all OOP languages) because it allows programmers to reuse existing software. More specifically, the existing class is used by all of the classes that are derived from it.
Here's a UML class diagram for an inheritance hierarchy that keeps track of people in a department store:

- **Person**
  - name : String

- **Customer**
  - address : String

- **Employee**
  - id : Integer

- **FullTime**
  - salary : Double

- **PartTime**
  - hourlyWage : Double

The **Person** class is generic - it contains data and methods that are common to all classes in the hierarchy.

As you go down the hierarchy, the classes get more specific. For example, the **Customer** and **Employee** classes describe specific types of people in the store.

---

**Inheritance Terminology**

- Within an inheritance hierarchy, pairs of classes are linked together. For each pair of linked classes, the more generic class is called the **superclass** and the more specific class is called the **subclass**.

- We say that subclasses are **derived** from superclasses. That makes sense when you realize that subclasses **inherit** all of the superclass's data and methods.

- Unfortunately, the terms superclass and subclass can be misleading. The "super" in superclass could imply that superclasses have more capability and the "sub" in subclass could imply that subclasses have less capability. Actually, it's the other way around - subclasses have more capability. Subclasses can do everything that superclasses can do, plus more.

- We'll stick with the terms superclass and subclass since those are the formal terms used by Oracle, but be aware of this alternative terminology:
  - Programmers often use the terms **parent class** or **base class** when referring to a superclass.
  - Programmers often use the terms **child class** or **derived class** when referring to a subclass.
UML Class Diagrams for Inheritance Hierarchies

- Usually, UML class diagrams show superclasses above subclasses. That's a common practice, but not a requirement. The following is a requirement....
- UML class diagrams use an arrow for inheritance relationships, with a hollow arrowhead pointing to the superclass.
- Warning:
  - The direction of arrows in UML class diagrams is opposite to the direction in which inheritance flows. In the previous slide, the Customer class inherits the name variable from the Person class. And yet the arrow does not go from Person to Customer; it goes from Customer to Person. That's because the arrow points to the superclass, and Person is the superclass.
- UML class diagram review:
  - What are the class boxes' minus signs for?
  - What are the class boxes' third compartments for?

Benefits of Inheritance

- Benefits of inheritance:
  - It helps with code reusability -
    - A superclass's code can be used for multiple subclasses. That eliminates code redundancy and makes debugging and upgrading easier.
    - A programmer can use an existing class to easily create a new subclass (no need to "reinvent the wheel.")
  - Smaller modules (because classes are split into superclasses and subclasses) -
    - That makes debugging and upgrading easier.
Person-Employee Example

- Implement a Person superclass with an Employee subclass.
- The Person class should:
  - Declare a name instance variable.
  - Define appropriate constructors.
  - Define a getName accessor method.
- The Employee class should:
  - Inherit Person's members.
  - Declare an id instance variable.
  - Define appropriate constructors.
  - Define a display method.

```java
public class Person {
    private String name = "";

    //**********************************************************
    public Person() {
    }

    public Person(String name) {
        this.name = name;
    }

    //**********************************************************
    public String getName() {
        return this.name;
    }
}
```
public class Employee extends Person
{
    private int id = 0;

    //**********************************************************
    public Employee()
    { }

    public Employee(String name, int id)
    { }

    //**********************************************************
    public void display()
    { }

    } // end Employee class

Inheritance for a superclass's private Instance Variables

- Since name is a private instance variable in the Person superclass, the Employee class's methods and constructors cannot access name directly (that's the same interpretation of private that we've always had).
- So how can Employee methods and constructors access name?
- By using the Person class's public methods and constructors!
- Aside - even though a superclass's private instance variables are not directly accessible from a subclass, private instance variables are still considered to be "inherited" by the subclass. So for this example, each Employee object does indeed contain a name instance variable.
Using `super` to Call Superclass Constructor

- A constructor may call a constructor in its superclass by using this syntax:
  
  ```java
  super(<arguments>);
  ```

- A constructor call is allowed only if it's the very first line in the constructor's body. This rule is applicable for constructor calls to constructors in the same class (using `this`) and also for constructor calls to constructors in the superclass (using `super`).

- What happens if you have a `super` constructor call and then a `this` constructor call as the first two lines in a constructor?

---

Calling a Superclass's Method from Within a Subclass

- As you may recall, in an instance method, if you call a method that's in the same class as the class you're currently in, the reference variable dot prefix is unnecessary.

- Likewise, in an instance method, if you call a method that's in the superclass of the class you're currently in, the reference variable dot prefix is unnecessary.

- Thus, in the `Employee` class's `display` method, call the `Person` class's `getName` method like this:

  ```java
  System.out.println("name: " + getName());
  ```
Default Call to Superclass Constructor

- The Java designers at Oracle are fond of calling superclass constructors since that promotes software reuse.
- If you write a subclass constructor and don’t include a call to another constructor (with this or with super), the Java compiler will sneak in and insert a superclass zero-parameter constructor call by default.
- Thus, although we showed nothing in Employee's zero-parameter constructor, the Java compiler automatically inserts super(); in it for us. So these two constructors are functionally equivalent:

```java
public Employee() {}
```

```java
public Employee() {
    super();
}
```

Method Overriding

- Method overriding is when a subclass has a method with the same name and the same parameter types as a method in its superclass.
- If a subclass contains an overriding method:
  - By default, an object of the subclass will use the subclass's overriding method (and not the superclass's overridden method).
  - Sometimes, an object of the subclass may need to call the superclass's overridden method. To do that, preface the method call with "super." (don't forget the dot).
- If a subclass and a superclass have methods with the same name, same parameter types, and different return types, that generates a compilation error.
FullTime Class

- Complete the implementation of the FullTime class below. In particular, provide a 3-parameter constructor and a display method.

```java
public class FullTime extends Employee {
    private double salary = 0.0;

    public FullTime() {
    }

    public static void main(String[] args) {
        FullTime fullTimer = new FullTime("Alan O'Brien", 5733, 80000);
        fullTimer.display();
        System.out.println(fullTimer.getName());
    }
}
```

FullTime Class

- From the previous slide's main method, note this getName call:
  ```java
  System.out.println(fullTimer.getName());
  ```

- fullTimer is a FullTime reference variable, so you would expect it to call methods defined in the FullTime class.

- But fullTimer calls getName, which is defined in the Person class, not the FullTime class. So how can fullTimer call the getName method?

- The Employee class inherits the getName method from the Person class and the FullTime class inherits the inherited getName method from the Employee class.
The final Access Modifier

- If you want to specify that a method definition cannot be overridden with a new definition in a subclass, use the `final` modifier in the method heading. For example:

```java
public class Person {
    ...
    public final String getName() {
        ...
    }
}

public class Employee extends Person {
    ...
    public String getName() {
        // compilation error
    }
}
```

Why would you ever want to use `final`?
- If you think that your method is perfect, then you might want to use `final` to prevent anyone else from overriding it in the future.
- `final` methods might run faster since the compiler can generate more efficient code for them (because there's no need to prepare for the possibility of inheritance).

- If you want to specify that a class cannot have any subclasses, use the `final` access modifier in the class heading. For example:

```java
public final class FullTime {
    ...
}

public class FullTimeNightShift extends FullTime {
    // compilation error
}
```
Aggregation, Composition, and Inheritance Compared

- We've covered two basic types of class relationships:
  - Aggregation and composition relationships are when one class is a whole and other classes are parts of that whole.
  - An inheritance relationship is when one class is a more detailed version of another class. The more detailed class is a subclass, and the other class is a superclass. The subclass inherits the superclass's members (variables and methods).
- We call aggregation and composition relationships "has a" relationships because one class, the container class, has a component class inside of it.
- We call an inheritance relationship an "is a" relationship because one class, a subclass, is a more detailed version of another class.

Aggregation, Composition, and Inheritance Combined

- In the real world, it's fairly common to have aggregation, composition, and inheritance relationships in the same program.
- For example, what sort of inheritance relationship could/should be added to our earlier Dealership program, shown below?

![Class Diagram](image_url)
Card Game Program

- Provide a class diagram for a card game program:
  - Assume it's a game like war or gin rummy where you have a deck of cards and two players.
  - Decide on appropriate classes. For each class, draw a UML-notation three-partition rectangle and put the class name in the top partition.
  - Look for composition relationships between classes. For each pair of classes related by composition, draw a composition connecting line with a diamond next to the containing class. For example, the left composition association line below is for Game, the containing class, and Deck, the contained class.

Card Game Program (continued):

- For each class, decide on appropriate instance variables and put them in the middle partition of the class's rectangle.
- For each class, decide on appropriate public methods and put their declarations in the bottom partition of the class's rectangle.
- Look for common instance variables and methods. If two or more classes contain a set of common instance variables and/or methods, implement a superclass and move the common entities to the superclass. For each subclass/superclass pair, draw an arrow from the subclass to the superclass to indicate an inheritance relationship.
Card Game Program

**Inheritance Implementation**

```java
public class Deck extends GroupOfCards {
    public static final int TOTAL_CARDS = 52;

    public Deck() {
        for (int i=0; i<TOTAL_CARDS; i++) {
            addCard(new Card((2 + i%13), i/13));
        }
    }
}
```

**Composition Implementation**

```java
public class Deck {
    public static final int TOTAL_CARDS = 52;
    GroupOfCards groupOfCards = new GroupOfCards();

    public Deck() {
        for (int i=0; i<TOTAL_CARDS; i++) {
            groupOfCards.addCard(new Card((2 + i%13), i/13));
        }
    }
}
```

---

Here's a main method for the card game program:

```java
public static void main(String[] args) {
    Scanner stdIn = new Scanner(System.in);
    String again;
    Game game;

    do {
        game = new Game();
        game.playAGame();
        System.out.print("Play another game (y/n)? ");
        again = stdIn.nextLine();
    } while (again.equals("y"));
    // end main
}
```
Here's a `playAGame` method for the `Game` class:

```java
public void playAGame()
{
    Card card;
    deck.shuffle();
    
    while (deck.getCurrentSize() > 0)
    {
        card = deck.dealCard();
        player1.addCard(card);
        card = deck.dealCard();
        player2.addCard(card);
    }
    ...
} // end playAGame
```