Chapter 9 – Inheritance

Learning About the Concept of Inheritance

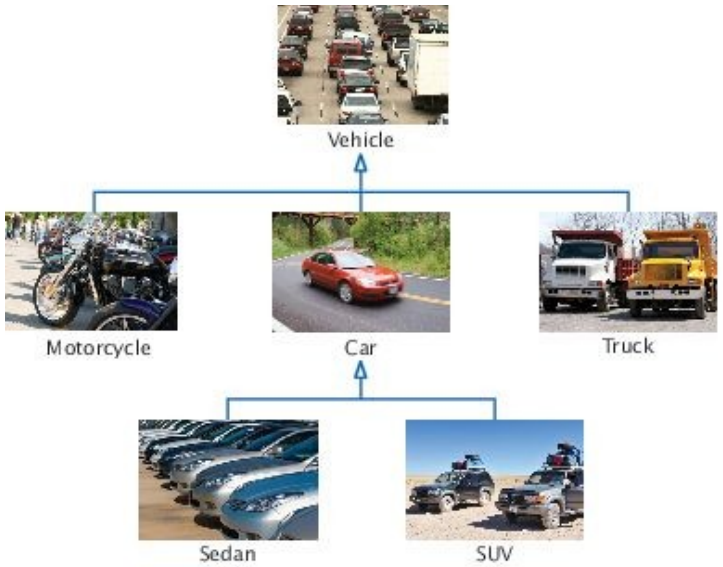
- **Inheritance**
  - A mechanism that enables one class to inherit both the behavior and the attributes of another class
  - Apply your knowledge of a general category to more specific objects

Inheritance Hierarchies

Inheritance: the relationship between a more general class (superclass) and a more specialized class (subclass).

The subclass inherits data and behavior from the superclass. Cars share the common traits of all vehicles

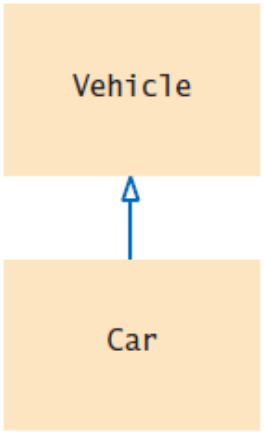
Example: the ability to transport people from one place to another



Diagramming Inheritance Using the UML

- **Unified Modeling Language (UML)**
  - Consists of many types of diagrams
- **Class diagram**
  - A visual tool
  - Provides an overview of a class

Inheritance Hierarchies



The class `Car` inherits from the class `Vehicle`

The `Vehicle` class is the superclass

The `Car` class is the subclass

Diagramming Inheritance Using the UML (cont'd.)

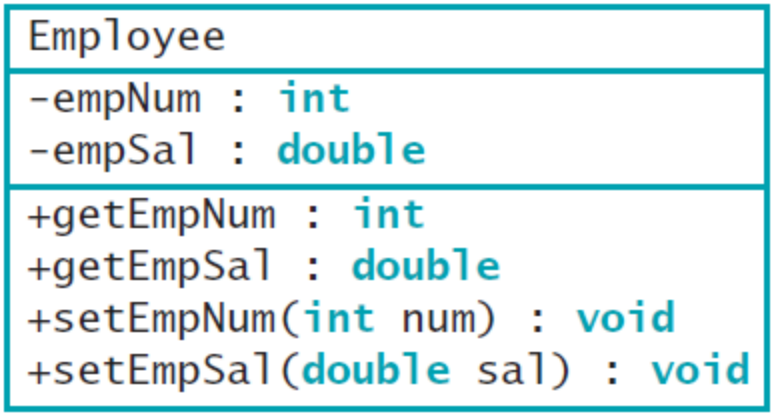


Figure 10-2 The Employee class diagram

Diagramming Inheritance Using the UML (cont'd.)

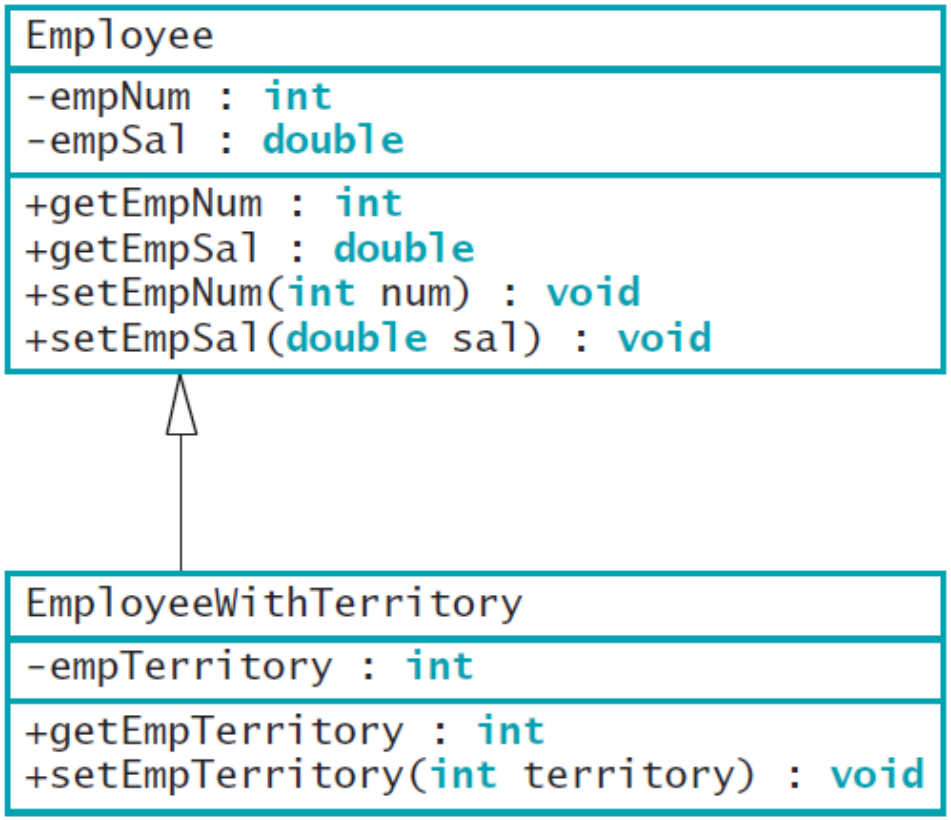


Figure 10-3 Class diagram showing the relationship between Employee and EmployeeWithTerritory

Diagramming Inheritance Using the UML (cont'd.)

- Use inheritance to create a derived class
  - Save time
  - Reduce errors
  - Reduce the amount of new learning required to use a new class

Inheritance Terminology

- *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 Terminology

- **Base class**
  - Used as a basis for inheritance
  - Also called:
    - **Superclass**
    - **Parent class**

Inheritance Terminology (cont'd.)

- **Derived class**
  - Inherits from a base class
  - Always "is a" case or an example of a more general base class
  - Also called:
    - **Subclass**
    - **Child class**

Extending Classes

- Keyword **`extends`**
  - Used to achieve inheritance in Java
  - Example:

```
public class EmployeeWithTerritory extends
Employee
```

- Inheritance is a one-way proposition
  - A child inherits from a parent, not the other way around
- Subclasses are more specific
- **`instanceof` operator**
  - **True** for parent class and child class

Extending Classes (cont'd.)

```
public class EmployeeWithTerritory extends Employee
{
   private int empTerritory;
   public int getEmpTerritory()
   {
      return empTerritory;
   }
   public void setEmpTerritory(int num)
   {
      empTerritory = num;
   }
}
```

Figure 10-4 The EmployeeWithTerritory class

Overriding Superclass Methods

- Create a subclass by extending an existing class
  - A subclass contains data and methods defined in the original superclass
  - Sometimes superclass data fields and methods are not entirely appropriate for subclass objects
- **Polymorphism**
  - Using the same method name to indicate different implementations

Overriding Superclass Methods (cont'd.)

- **Override the method** in the parent class
  - Create a method in a child class that has the same name and parameter list as a method in its parent class
  - Use @Override annotation at the top of the method
- **Subtype polymorphism**
  - The ability of one method name to work appropriately for different subclass objects of the same parent class

@Override Annotation

- **Why should we use @Override**
  - Use @Override annotation at the top of the method
- Do it so that you can take advantage of the compiler checking to make sure you actually are overriding a method when you think you are.
  - This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does.
- Secondly, it makes your code easier to understand because it is more obvious when methods are overwritten.
- Aside from that, there is no other benefits: it does not impact polymorphism subtyping in anyway

Calling Constructors During Inheritance

- When you instantiate an object that is a member of a subclass, you call at least two constructors:
  - The constructor for the base class
  - The constructor for the extended class
- The superclass constructor must execute first
- When the superclass contains a default constructor, the execution of the superclass constructor is not apparent, however, we should still call the superclass constructor in the subclass constructor

Calling Constructors During Inheritance (cont'd.)

```
public class ASuperClass
{
   public ASuperClass()
   {
      System.out.println("In superclass constructor");
   }
}
public class ASubClass extends ASuperClass
{
   public ASubClass()
   {
      System.out.println("In subclass constructor");
   }
}
public class DemoConstructors
{
   public static void main(String[] args)
   {
      ASubClass child = new ASubClass();
   }
}
```

Figure 10-8 Three classes that demonstrate constructor calling when a subclass object is instantiated

Calling Constructors During Inheritance (cont'd.)

Figure 10-9 Output of the `DemoConstructors` application

Using Superclass Constructors That Require Arguments

- When you write your own constructor, you replace the automatically supplied version
- When extending a superclass with constructors that require arguments, the subclass must provide the superclass constructor with the arguments it needs

Using Superclass Constructors That Require Arguments (cont'd.)

- When a superclass has a default constructor, you can create a subclass with or without its own constructor
- When a superclass contains only constructors that require arguments, you must include at least one constructor for each subclass you create
  - The first statement within each constructor must call one of the superclass constructors

Using Superclass Constructors That Require Arguments (cont'd.)

- Call the superclass constructor
  - `super(`*`list of arguments`*`);`
- Keyword **`super`**
  - Always refers to the superclass

Accessing Superclass Methods

- Use the overridden superclass method within a subclass
  - Use the keyword `super` to access the parent class method

Accessing Superclass Methods (cont'd.)

```
public class PreferredCustomer extends Customer
{
   double discountRate;
   public PreferredCustomer(int id, double bal, double rate)
   {
      super(id, bal);
      discountRate = rate;
   }
   public void display()
   {
      super.display();
      System.out.println("   Discount rate is " + discountRate);
   }
}
```

Figure 10-13 The `PreferredCustomer` class

Comparing `this` and `super`

- Think of the keyword `this` as the opposite of `super` within a subclass
- When a parent class contains a method that is not overridden, the child can use the method name with `super` or `this`, or alone
  - Otherwise, the child class method can specify which version it wants by using the super or this objects

Employing Information Hiding

- Within the `Student` class:
  - The keyword `private` precedes each data field
  - The keyword `public` precedes each method
- **Information hiding**
  - The concept of keeping data private
  - Data can be altered only by methods you choose and only in ways that you can control

Employing Information Hiding (cont'd.)

```
public class Student
{
   private int idNum;
   private double gpa;
   public int getIdNum()
   {
      return idNum;
   }
   public double getGpa()
   {
      return gpa;
   }
   public void setIdNum(int num)
   {
      idNum = num;
   }
   public void setGpa(double gradePoint)
   {
      gpa = gradePoint;
   }
}
```

Figure 10-16 The `Student` class

Employing Information Hiding (cont'd.)

- When a class serves as a superclass, subclasses inherit all data and methods of the superclass
  - Except `private` members of the parent class are not accessible within a child class's methods or objects
    - If there are accessor/mutator methods for the private member of the parent class, that member would then be accessible to both the child class member function and child class object
  - Private members are not readily accessible, however, all other members are accessible within the child class and also to the child class object

Employing Information Hiding (cont'd.)

- Keyword **`protected`**
  - Provides an intermediate level of security between `public` and `private` access
  - Can be used within its own class or in any classes extended from that class
  - Protected members are accessible within the child class and also to the child class object (as long as the child class belongs to the same package)

Employing Information Hiding (cont'd.)

- Keyword **`protected`**

Modifier	Classes and interfaces	Methods and variables
default (no modifier)	Visible in its package.	Inherited by any subclass in the same package as its class. Accessible by any class in the same package as its class.
public	Visible anywhere.	Inherited by all subclasses of its class. Accessible anywhere.
protected	N/A	Inherited by all subclasses of its class. Accessible by any class in the same package as its class.
private	Visible to the enclosing class only	Not inherited by any subclass. Not accessible by any other class.

@Override Annotation

- **Why should we use @Override**
  - Use @Override annotation at the top of the method
- Do it so that you can take advantage of the compiler checking to make sure you actually are overriding a method when you think you are.
  - This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does.
- Secondly, it makes your code easier to understand because it is more obvious when methods are overwritten.
- Aside from that, there is no other benefits: it does not impact polymorphism subtyping in anyway

Methods You Cannot Override

- `static` methods
- `final` methods
- Methods within `final` classes

Thank you

- Let me know if you have any questions