Inheritance

Archie Powell

November 7th, 2023

Inheritance

Table of Contents

Section 1

Introduction to Inheritance

Inheritance in a nutshell

- **Inheritance** is a mechanism that enables one class to acquire all the behaviours and attributes of another class.
- We are familiar with the concept of inheritance outside of programming.
- E.g you have common features that you have inherited from your parents, such as eye colour (attribute) or way of approaching tasks (methods).

Unified Modelling Language (UML)

- Understanding the relationship between classes can become a challenge in large OOP systems.
- To help us, we can use a graphical language, **UML**, to describe classes and their relationships with one another.
- Classes are represented using a **class diagram**, which details the **class name**, the **attributes** and **methods**, as well as their **parameters** and **return types**.
- The instructions that made up the method body are omitted.

The Employee class

```
public class Employee
{
   private int id;
   private double salary;
   public int getId()
   {
      return id;
   }
   public double getSalary()
   {
      return salary;
   }
   public void setId(int idNum)
   {
      id = idNum;
   }
   public void setSalary(double sal)
   {
      salary = sal;
   }
}
```

Figure 10-1 The Employee class

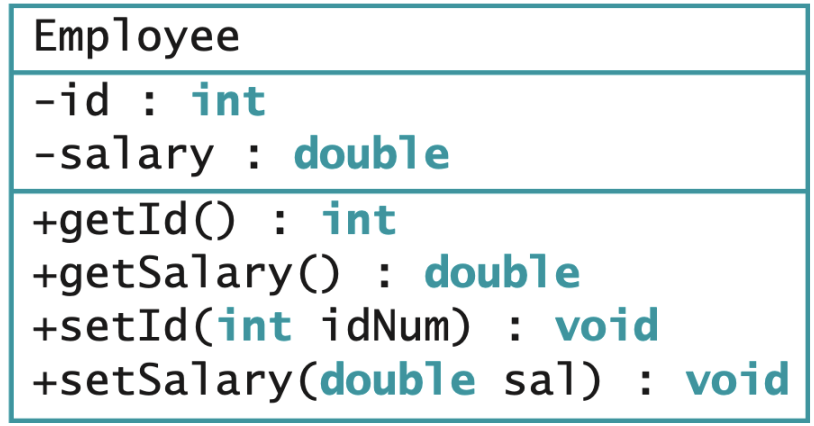

Figure 10-2 The Employee class diagram

Employee with territory

- Let's say we have hired a new employee, for example a salesperson, who not only requires an ID and salary, but also the territory they are serving.
- We *could* create a completely new class e.g EmployeeWithTerritory with three class fields (idNum, salary and territory) and six methods (accessors and mutators).
- Why is this not the best approach?

Employee with territory

- Let's say we have hired a new employee, for example a salesperson, who not only requires an ID and salary, but also the territory they are serving.
- We *could* create a completely new class e.g EmployeeWithTerritory with three class fields (idNum, salary and territory) and six methods (accessors and mutators).
- Why is this not the best approach?
  - We are **duplicating** much of the work we have already done in our original Employee class!
- Instead, we can inherit the attributes and methods from the Employee class and just add the new territory attribute and related methods.

Employee with territory class diagram

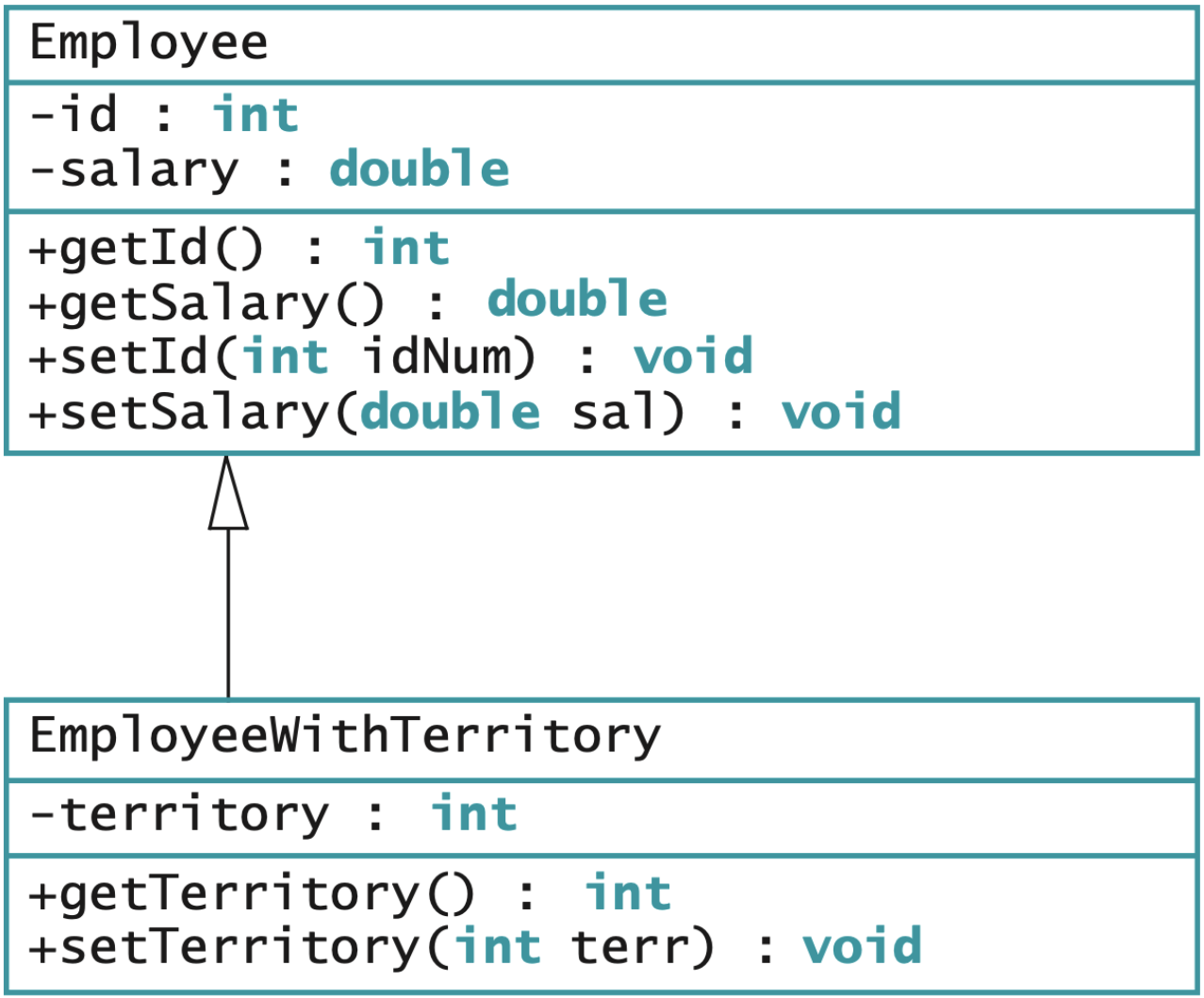

Figure 10-3 Class diagram showing the relationship between `Employee` and `EmployeeWithTerritory`

Benefits of Inheritance

By using inheritance to create the EmployeeWithTerritory class, there are several benefits:

- Saves time because the Employee fields and methods already exist.
- Reduces the likelihood of errors as the fields and methods have already been tested.
- Reduces the amount of new learning required for programmers as they are already familiar with the original class.

Inheritance Terminology

- A class that is used as a basis for inheritance (e.g Employee) is called a **base** class, **parent** class, or **superclass**.
- A class that inherits from a base class (e.g EmployeeWithTerritory), is called a **derived** class, a **child** class, or a **subclass**.
- **Inheritance** is used to implement an "is-a" relationship:
  - EmployeeWithTerritory *is an* Employee.
  - Evergreen Tree *is a* Tree
  - Car *is a* Vehicle

Inheritance Terminology

- If a class has instance variables containing one or more members of another class, this relationship is called **Composition**.
- **Composition** is used to express a "has-a" relationship:
  - A business *has* departments
  - A supermarket *has* isles
- If members would continue to exist if an object containing them did not, is called **Aggregation**.
  - An employee would exist even if a business did not, so this is **Aggregation**.
  - A department has no meaning without a business, so this is simply **Composition**.

Which statement is false?

- A. When you use inheritance in Java, you can create a new class that contains all the data and methods of an existing class.
- B. When you use inheritance, you save time and reduce errors.
- C. A class that is used as a basis for inheritance is called a subclass.

Which statement is false?

- A. When you use inheritance in Java, you can create a new class that contains all the data and methods of an existing class. -True
- B. When you use inheritance, you save time and reduce errors. -True
- C. A class that is used as a basis for inheritance is called a subclass. **-False, it is a superclass or base class**

Section 2

Extending Classes

Extending a class in Java (demo)

- To use inheritance in Java, we use the *extend* keyword.
- To check if an object is an instance of a class, you can use the *instanceof* keyword.

```
public class Employee
{
   private int id;
   private double salary;
   public int getId()
   {
      return id;
   }
   public double getSalary()
   {
      return salary;
   }
   public void setId(int idNum)
   {
      id = idNum;
   }
   public void setSalary(double sal)
   {
      salary = sal;
   }
}
```

Figure 10-1 The Employee class

```
public class EmployeeWithTerritory extends Employee
{
   private int territory;
   public int getTerritory()
   {
      return territory;
   }
   public void setTerritory(int terr)
   {
      territory = terr;
   }
}
```

Figure 10-4 The EmployeeWithTerritory class

Section 3

Overriding Superclass Methods

Polymorphism and Overriding

- Sometimes, we may want to use the **override** the parent class method and use the child's version instead.
- For example, an ElectricCar class may inherit a *start()* method from a parent Car class, but it would have different functionality.
- Using the same method name to indicate different implementations is called **polymorphism**.
- When **overriding** a parent's method, it must have the name and parameters.
- If the method has different parameters, this is **overloading**, which was covered in earlier lectures.

@Override annotation (Demo)

- When you override a parent class method in a child class, you can insert the tag **@Override** before the method definition.
- This tells the compiler that you want to override the method in the parent class and not create a new method.
- Using **@Override** is not mandatory, but useful as the compiler will not compile the code if the method header is not correct.
    - For example, if you confuse overloading and overriding and create the same method as the parent class but with different parameters, the code will not compile if the @Override tag is used.

Which statement is false?

- A. Any child class object has all the attributes of its parent, but all of those attributes might not be directly accessible.
- B. You override a parent class method by creating a child class method with the same identifier but a different parameter list or return type.
- C. When a child class method overrides a parent class method, and you use the method name with a child class object, the child class method version executes.

Which statement is false?

- A. Any child class object has all the attributes of its parent, but all of those attributes might not be directly accessible. - True
- B. You override a parent class method by creating a child class method with the same identifier but a different parameter list or return type. -**False, this is overloading, not overriding**
- C. When a child class method overrides a parent class method, and you use the method name with a child class object, the child class method version executes. -True

Section 4

Calling Constructors During Inheritance

Constructors During Inheritance

- When you create any object, you are calling a constructor:
  - SomeClass anObject = new SomeClass();
- When you instantiate an object that is a subclass, at least two constructors are called:
  - The superclass constructor
  - The subclass constructor
- When the object is created, the superclass constructor runs *first*, followed by the subclass constructor.

Constructors During Inheritance - Example

```
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

Superclass Constructors that Require Arguments (Demo)

- If a superclass has a constuctor which takes no arguments, there is no requirement to create a constructor in the subclass.
- However, when a superclass contains only constructors that require arguments, you must include at least one constructor for each subclass you create.
- If this is the case, the first statement within each subclass constructor **must** call one of the superclass constructors.
- To do so, you can use the *super* keyword in the subclass to access the superclass constructor.
  - super(*list of arguments*);
- This **must** be the first line in the subclass constructor.

Section 5

Accessing Superclass Methods

super.methodName()

- We mentioned earlier that we can override a parent class method in a subclass using the same method name and parameters.
- However, we may still want to call the parent's original method from within the subclass.
- For example, displaying details about a customer in a loyalty program may include a line about a special discount, but may also include details that apply for all customers.
- It doesn't make sense to rewrite code if the normal customer class will already display these details.
- To do so, we can use *super.methodName()*.

super.methodName() - Example

```
public class Customer
{
   private int idNumber;
   private double balanceOwed;
   public Customer(int id, double bal)
   {
      idNumber = id;
      balanceOwed = bal;
   }
   public void display()
   {
      System.out.println("Customer #" + idNumber +
         " Balance $" + balanceOwed);
   }
}
```

Figure 10-12 The Customer class

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

Figure 10-13 The PreferredCustomer class

Comparing *this* and *super*

- You should familiar with the *this* keyword, which can be used to access the attributes or methods of the class.
  - e.g *this*.id = id;
- If a **subclass** has overridden a **superclass** method named someMethod(), then *super*.someMethod() refers to the **superclass** version of the method, and both someMethod() and *this*.someMethod() refer to the **subclass** version.
- If a **subclass** has **not** overridden the **superclass**, then *super*.someMethod(), someMethod() and *this*.someMethod() all refer to the **superclass** version.

Which statement is false?

- A. You can use the keyword *this* within a method in a subclass to access an overridden method in a superclass.
- B. You can use the keyword *super* within a method in a subclass to access an overridden method in a superclass.
- C. You can use the keyword *super* within a method in a subclass to access a method in a superclass that has not been overridden.

Which statement is false?

- A. You can use the keyword *this* within a method in a subclass to access an overridden method in a superclass. -**False, as doing this would call the method in thus subclass**
- B. You can use the keyword *super* within a method in a subclass to access an overridden method in a superclass. -True
- C. You can use the keyword *super* within a method in a subclass to access a method in a superclass that has not been overridden. -True

Section 6

Employing Information Hiding

Information/Data Hiding

- In Week 2, we covered **information/data hiding**, the idea that information in classes should be kept private and accessible only via accessors (getters) and mutators (setters).
- We are familiar with both **public** and **private** access specifiers:
  - **public** allows access from both inside and outside the class.
  - **private** only allows access from within the class.
- When you create a subclass, you can access all the methods and attributes of parent class, but **not** if their access specifier is **private**.
- As most attributes will be private, how can we access them in a subclass?

Protected access

- We can instead use **protected** access:
  - **public** allows access from both inside and outside the class.
  - **private** only allows access from within the class.
  - **protected** allows access from within the class and subclasses but not from outside.
- This is useful if you want subclasses to access parent class data but you don't want this data to be accessible elsewhere.
- However, **protected** should be used sparingly, as the public accessors (getters) and mutators (setters) should be used wherever possible.
- Using *protected* throughout your code is usually a sign that you are not writing good object-orientated code.

Section 7

Methods You Cannot Override

The motivation for non-overridable methods

- You may want to disable the ability to override a method in your parent class.
- For example, an Employee class may have a method to generate an ID, and you don't want any derived classes to be able to generate their own version.
- There are three types of methods that you cannot override in a subclass:
  - static methods
  - final methods
  - methods within final classes

Cannot override: static methods

- A subclass cannot override methods that are declared static in the superclass.
- You *can* create a static method with the same method signature but this will only *hide* the superclass method, not override it.
  - If you use the @Override tag here, the code will not compile.
- If you do this, you cannot use the *super* keyword as the method can be ran without any parent object existing.
- However, as static methods can be accessed from anywhere, you *can* use any methods that are static in the subclass (like you can in any class).

Cannot override: static methods - Example

```
public class BaseballPlayer
{
   private int jerseyNumber;
   private double battingAvg;
   public static void showOrigins()
   {
      System.out.println("Abner Doubleday is often " +
         "credited with inventing baseball");
   }
}
```

Figure 10-17 The `BaseballPlayer` class

```
public class ProfessionalBaseballPlayer extends BaseballPlayer
{
   double salary;
   @Override
   public static void showOrigins()
   {
      super.showOrigins();
      System.out.println("The first professional " +
         "major league baseball game was played in 1871");
   }
}
```

Don't Do It
You cannot refer to `super` in a static method.

Figure 10-20 The `ProfessionalBaseballPlayer` class with a static method that attempts to reference `super`

Cannot override: static methods - Example

```
public class BaseballPlayer
{
   private int jerseyNumber;
   private double battingAvg;
   public static void showOrigins()
   {
      System.out.println("Abner Doubleday is often " +
         "credited with inventing baseball");
   }
}
```

Figure 10-17 The `BaseballPlayer` class

```
public class ProfessionalBaseballPlayer extends BaseballPlayer
{
   double salary;
   public static void showOrigins()
   {
      BaseballPlayer.showOrigins();
      System.out.println("The first professional " +
         "major league baseball game was played in 1871");
   }
}
```

Figure 10-22 The `ProfessionalBaseballPlayer` class

Cannot override: final methods

- It may not be desirable to use static methods in your classes to prevent overriding.
- Instead, the *final* keyword can be used in the parent class method which prevents any subclasses overriding it.
- The advantage of using final is that the compiler can optimise the code.
  - Java uses **virtual method calls**, which means the exact version of a method (e.g superclass or subclass) is not known until the code runs.
  - However, by using *final*, the compiler knows the parent class version is the one that will run, which makes the program more efficient.
  - If the method is small (one or two lines), it can replace the method call with the method itself in the code. This is known as **inlining**.

Cannot override: final methods - Example

```
public class BasketballPlayer
{
   private int jerseyNumber;
   public final void displayMessage()
   {
      System.out.println("Michael Jordan is the " +
         "greatest basketball player - and that is final");
   }
}
```

Figure 10-25 The `BasketballPlayer` class

```
public class ProfessionalBasketballPlayer extends BasketballPlayer
{
   double salary;
   @Override
   public void displayMessage()
   {
      System.out.println("I have nothing to say");
   }
}
```

Don't Do It
A child class method cannot override a `final` parent class method.

Figure 10-26 The `ProfessionalBasketballPlayer` class that attempts to override a `final` method

Cannot override: final superclass - Example

- For maximum strictness, you can declare a class itself *final*.
- By doing so, this prevents the class being extended (the class cannot be a parent).

```
public final class HideAndGoSeekPlayer
{
   private int count;
   public void displayRules()
   {
      System.out.println("You have to count to " + count +
         " before you start looking for hiders");
   }
}
public final class ProfessionalHideAndGoSeekPlayer
   extends HideAndGoSeekPlayer
{
   private double salary;
}
```

Notice the keyword `final` in the method header.

Don't Do It
You cannot extend a `final` class.

Figure 10-28 The `HideAndGoSeekPlayer` and `ProfessionalHideAndGoSeekPlayer` classes

Section 8

Lecture summary

Lecture summary

Questions

Thank you! Questions?