Advanced Inheritance

Archie Powell

November 9th, 2023

Advanced Inheritance

Table of Contents

Section 1

Creating and Using Abstract Classes

Concrete classes

- We are now familiar with the concepts of a **superclass** and a **subclass**. Typically, we use a superclass as a general class and extend the class to create more specific versions of the class:
  - e.g Poodle extends from a Dog class
  - e.g EmployeeWithTerritory extends from an Employee class
- Our superclasses so far have all been examples of **concrete classes**, which is a class from which you can instantiate objects.
- What if we want to go *even more* general?

Abstract classes

- An **abstract** class is a class that you can only extend from, and cannot instantiate.
- Abstract classes can include two method types:
  - **Non-abstract methods**, like those have studied so far, which are implemented sin the class and are inherited by its children.
  - **Abstract methods** have no body and must be implemented in child classes.
- Typically, **abstract classes** include at least one **abstract method**, and only **abstract classes** can have **abstract methods**.

Abstract classes - Example

```
public abstract class Animal
{
    private String name;
    public abstract void speak();
    public String getName()
    {
        return name;
    }
    public void setName(String animalName)
    {
        name = animalName;
    }
}
```

Figure 11-1 The abstract Animal class

```
public class Dog extends Animal
{
    @Override
    public void speak()
    {
        System.out.println("Woof!");
    }
}
```

Figure 11-2 The Dog class

```
public class Cow extends Animal
{
    @Override
    public void speak()
    {
        System.out.println("Moo!");
    }
}
```

Figure 11-3 The Cow class

```
public class Snake extends Animal
{
    @Override
    public void speak()
    {
        System.out.println("Ssss!");
    }
}
```

Figure 11-4 The Snake class

Section 2

Using Dynamic Method Binding

Using Dynamic Method Binding

- Recall: when we create a subclass object, the object *is a* superclass object too:
  - Car *is a* Vehicle
  - Evergreen Tree *is a* Tree
- Thus, when creating our subclass object, we can set the reference to the object that of the parent class:
  - Vehicle myCar = new Car();
- This allows us to pass a Car object to a function which requires a Vehicle parameter type.
- A program's ability to select the correct subclass method depending on the argument type is known as **dynamic method binding**.

Using Dynamic Method Binding

```
public class AnimalReference
{
   public static void main(String[] args)
   {
      Animal animalRef;
      animalRef = new Cow();
      animalRef.speak();
      animalRef = new Dog();
      animalRef.speak();
   }
}
```

Figure 11-8 The `AnimalReference` application

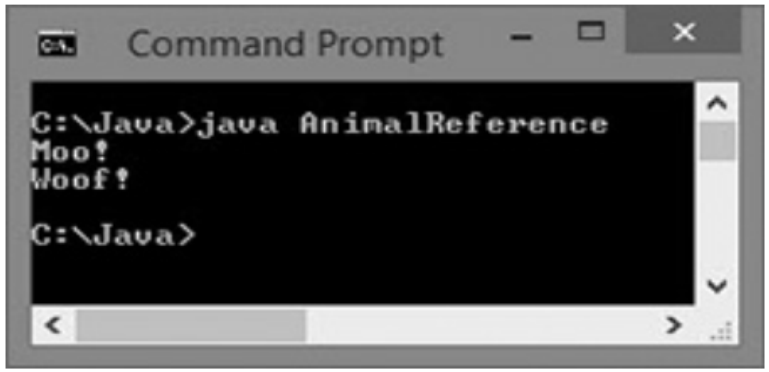

Figure 11-9 Output of the `AnimalReference` application

Using a Superclass as a Method Parameter Type

```
public class TalkingAnimalDemo
{
   public static void main(String[] args)
   {
      Dog dog = new Dog();
      Cow cow = new Cow();
      dog.setName("Ginger");
      cow.setName("Molly");
      talkingAnimal(dog);
      talkingAnimal(cow);
   }
   public static void talkingAnimal(Animal animal)
   {
      System.out.println("Come one. Come all.");
      System.out.println
         ("See the amazing talking animal!");
      System.out.println(animal.getName() +
         " says");
      animal.speak();
      System.out.println("***************");
   }
}
```

Figure 11-10 The `TalkingAnimalDemo` class

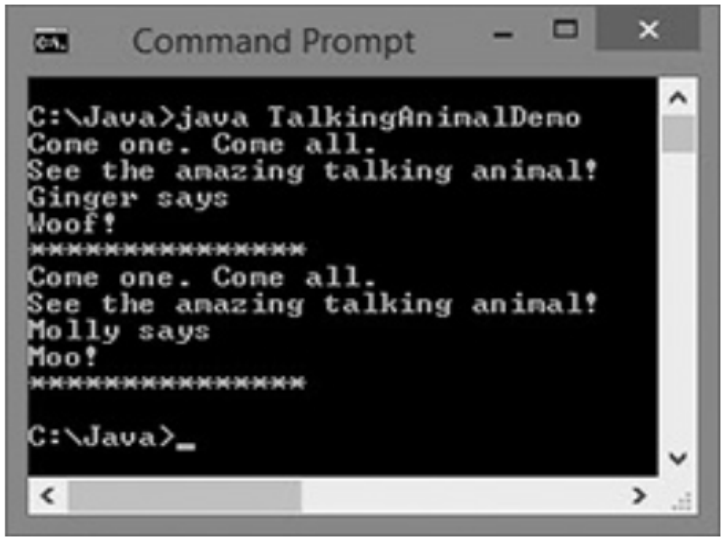

Figure 11-11 Output of the `TalkingAnimalDemo` application

Section 3

Creating Arrays of Subclass Objects

Creating Arrays of Subclass Objects - Demo

- Creating an array of objects uses similar syntax to normal arrays:
  - Vehicle[] vehicleRef = new Vehicle[2];
- Although we are saying "array of objects", what we are actually creating is an array of object *references*, not objects themselves.
- This means that we can create arrays of abstract class types.

Which statement is false?

- A. You can assign a superclass reference to an array of its subclass type.
- B. The following statement creates an array of 10 Table references: Table[] table = new Table[10];
- C. You can assign subclass objects to an array that is their superclass type.

Which statement is false?

- A. You can assign a superclass reference to an array of its subclass type. -**False**
- B. The following statement creates an array of 10 Table references: Table[] table = new Table[10]; -True
- C. You can assign subclass objects to an array that is their superclass type. -True

Section 4

Using the Object Class and Its Methods

The Object Class

- Every single class in Java is actually a subclass, except one: the **Object** class.
- When you create a class, it automatically extends the **Object** class.
- As a result, all objects get access to a number of methods.

Object methods

Method	Description
`Object clone()`	Creates and returns a copy of this object
`boolean equals (Object obj)`	Indicates whether some object is equal to the parameter object (this method is described in detail below)
`void finalize()`	Called by the garbage collector on an object when there are no more references to the object
`Class<?> getClass()`	Returns the class to which this object belongs at run time
`int hashCode()`	Returns a hash code value for the object (this method is described briefly below)
`void notify()`	Wakes up a single thread that is waiting on this object's monitor
`void notifyAll()`	Wakes up all threads that are waiting on this object's monitor
`String toString()`	Returns a string representation of the object (this method is described in detail below)
`void wait()`	Causes the current thread to wait until another thread invokes either the `notify()` method or the `notifyAll()` method for this object
`void wait (long timeout)`	Causes the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed
`void wait (long timeout, int nanos)`	Causes the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed

Table 11-1 `Object` class methods

Object methods

Method	Description
`Object clone()`	Creates and returns a copy of this object
`boolean equals (Object obj)`	Indicates whether some object is equal to the parameter object (this method is described in detail below)
`void finalize()`	Called by the garbage collector on an object when there are no more references to the object
`Class<?> getClass()`	Returns the class to which this object belongs at run time
`int hashCode()`	Returns a hash code value for the object (this method is described briefly below)
`void notify()`	Wakes up a single thread that is waiting on this object's monitor
`void notifyAll()`	Wakes up all threads that are waiting on this object's monitor
`String toString()`	Returns a string representation of the object (this method is described in detail below)
`void wait()`	Causes the current thread to wait until another thread invokes either the `notify()` method or the `notifyAll()` method for this object
`void wait (long timeout)`	Causes the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed
`void wait (long timeout, int nanos)`	Causes the current thread to wait until another thread invokes the `notify()` or `notifyAll()` method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed

Table 11-1 `Object` class methods

toString() Method

- The Object class toString() method converts an Object into a String that contains information about the Object.
- By default, this will be the class name, followed by the @ symbol, and then a hash code (a unique identifier)
  - e.g Dog@6d06d69c

```
public abstract class Animal
{
   private String name;
   public abstract void speak();
   public String getName()
   {
      return name;
   }
   public void setName(String animalName)
   {
      name = animalName;
   }
}
public class Dog extends Animal
{
   public void speak()
   {
      System.out.println("Woof!");
   }
}
public class DisplayDog
{
   public static void main(String[] args)
   {
      Dog myDog = new Dog();
      String dogString = myDog.toString();
      System.out.println(dogString);
   }
}
```

Figure 11-15 The Animal and Dog classes and the DisplayDog application

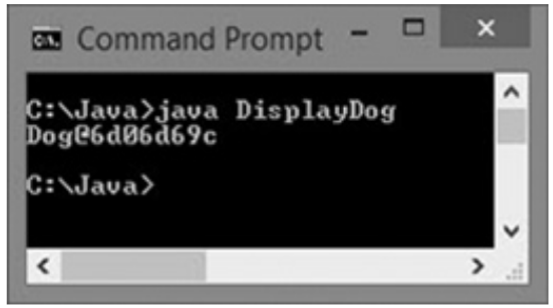

Figure 11-16 Output of the DisplayDog application

toString() Method

- This information isn't particularly helpful, so it can be useful to override the method and write your own.
- By doing so, you can print out object attributes which can be handy when debugging code.
- For example, here we have incorrectly assigned the account number to the customer balance.
- By printing out the attributes in our toString() method, we can see this mistake.

```
public class BankAccount
{
   private int acctNum;
   private double balance;
   public BankAccount(int num, double bal)
   {
      acctNum = num;
      balance = num;
   }
   @Override
   public String toString()
   {
      String info = "BankAccount acctNum = " + acctNum +
         " Balance = $" + balance;
      return info;
   }
}
```

Don't Do It
The bal parameter should be assigned to balance.

Figure 11-17 The BankAccount class

```
public class TestBankAccount
{
   public static void main(String[] args)
   {
      BankAccount myAccount = new BankAccount(123, 4567.89);
      System.out.println(myAccount.toString());
   }
}
```

Figure 11-18 The TestBankAccount application

Figure 11-19 Output of the TestBankAccount application

equals() Method

- The equals method takes in an object as a parameter and compares it with the object calling the method.
- If the objects are equal, the method will return a boolean set to true. Otherwise it will return a boolean set to false.
- By default, the two are equal only if the objects have the same hashcode/memory reference.
- Why is this not particularly useful?

equals() Method

- The equals method takes in an object as a parameter and compares it with the object calling the method.
- If the objects are equal, the method will return a boolean set to true. Otherwise it will return a boolean set to false.
- By default, the two are equal only if the objects have the same hashcode/memory reference.
- Why is this not particularly useful?
  - We may consider objects with all the same attribute values equal, but the equals() method will return false in this case! (unless the objects are the exact same).

Default equals() Method - Example

```
public class CompareAccounts
{
   public static void main(String[] args)
   {
      BankAccount acct1 = new BankAccount(1234, 500.00);
      BankAccount acct2 = new BankAccount(1234, 500.00);
      if(acct1.equals(acct2))
         System.out.println("Accounts are equal");
      else
         System.out.println("Accounts are not equal");
   }
}
```

Figure 11-20 The CompareAccounts application

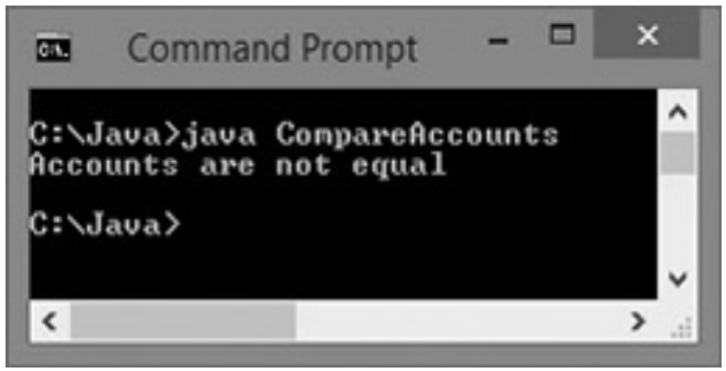

Figure 11-21 Output of the CompareAccounts application

Writing our own equals() Method

- If we want to compare two objects based on the values they hold, there are three approaches:
  - Create an all new method which compares objects, e.g areTheyEqual().
  - Overload the Object class equals() method.
  - Override the Object class equals() method.
- The first approach is just like creating any other Java method and comparing the attributes of the classes.
- The advantage of using the equals() identifier is that programmers expect it to be used to compare objects.

Overloading equals() - Example

```
public class BankAccount
{
   private int acctNum;
   private double balance;
   public BankAccount(int num, double bal)
   {
      acctNum = num;
      balance = bal;
   }
   @Override
   public String toString()
   {
      String info = "BankAccount acctNum = " + acctNum +
         " Balance = $" + balance;
      return info;
   }
   public boolean equals(BankAccount secondAcct)
   {
      boolean result;
      if(acctNum == secondAcct.acctNum && balance == secondAcct.balance)
         result = true;
      else
         result = false;
      return result;
   }
}
```

Figure 11-22 The `BankAccount` class containing its own `equals()` method

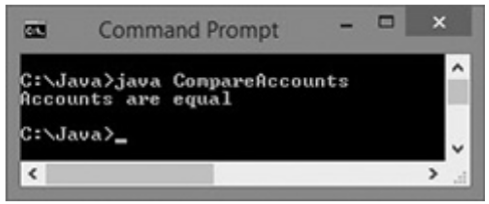

Figure 11-23 Output of the `CompareAccounts` application after adding an overloaded `equals()` method to the `BankAccount` class

Overriding equals()

- To override a method, it must have the same signature as the parent's method, therefore to override equals() the parameter must be of Object type.
- Before doing any attribute comparisons, you must first cast the Object parameter to the specific class.

```
@Override
public boolean equals(Object obj)
{
    BankAccount secondAcct = (BankAccount)obj;
    boolean result;
    if(acctNum == secondAcct.acctNum && balance == secondAcct.balance)
        result = true;
    else
        result = false;
    return result;
}
```

Figure 11-24 BankAccount equals() method that overrides Object class version

Overriding equals()

- The previous approach will work, but is not very robust. When overriding the equals() method, there are four recommendations:
  - Determine if the equals() argument is the same object as the calling object, and return true if it is.
  - Return false if the Object argument is null.
  - Return false if the calling object and argument object are not the same class.
  - Cast the Object argument to the same type as the calling object only if they are the same class.

Overriding equals() - Improved Example?

```
public boolean equals(Object obj)
{
   boolean result;
   if(obj == this)
      result = true;
   else
      if(obj == null)
         result = false;
      else
         if(obj.getClass() != this.getClass())
            result = false;
   BankAccount secondAcct = (BankAccount)obj;
   if(acctNum == secondAcct.acctNum && balance == secondAcct.balance)
      result = true;
   else
      result = false;
   return result;
}
```

Figure 11-25 Improved BankAccount equals() method that overrides Object class version

Which statement is false?

- A. When you define a class, if you do not explicitly extend another class, your class is an extension of the Object class.
- B. The Object class is defined in the java.lang package that is imported automatically every time you write a program.
- C. The Object class toString() and equals() methods are abstract.

Which statement is false?

- A. When you define a class, if you do not explicitly extend another class, your class is an extension of the Object class. -True
- B. The Object class is defined in the java.lang package that is imported automatically every time you write a program. -True
- C. The Object class toString() and equals() methods are abstract. **-False, as they can be called without any implementation in a subclass.**

Section 5

Creating and Using Interfaces

Multiple Inheritance

- We may want to inherit properties from more than one class.
  - For example, creating an InsuredCar class which inherits from both an InsuredItem class and a Car class.
- The capability to inherit from more than one class is called **multiple inheritance**.
- Practically speaking, there are challenges associated with this:
  - What happens if two classes have the attribute identifier (i.e variable name)?
  - Which constructor is called first?
  - Which class does the *super* keyword refer to?

Interfaces

- Instead of offering multiple inheritance, Java offers an alternative - an **interface**.
- An interface is a description of what a class does, but not how it is done.
- An **interface** looks like a Java class but **all** of its methods are *public* and *abstract*, and any attributes are *public*, *static* and *final*.
- When you create a class that uses an interface, you include the *implements* keyword and the interface name in the class header.
- We can think of an interface as a protocol or agreed-on behaviour:
  - An Oven is a device for heating food.
  - A Microwave is a device for heating food.
  - However, the way they heat food is very different.
  - A FoodHeatingDevice interface may define the behaviour of heating food e.g a heat() method but the implementation will be different in Oven and Microwave classes.

Interfaces - Example

```
public abstract class Animal
{
   private String name;
   public abstract void speak();
   public String getName()
   {
      return name;
   }
   public void setName(String animalName)
   {
      name = animalName;
   }
}
public class Dog extends Animal
{
   public void speak()
   {
      System.out.println("Woof!");
   }
}
```

Figure 11-26 The Animal and Dog classes

```
public interface Worker
{
   public void work();
}
```

Figure 11-27 The Worker interface

```
public class WorkingDog extends Dog implements Worker
{
   private int hoursOfTraining;
   public void setHoursOfTraining(int hrs)
   {
      hoursOfTraining = hrs;
   }
   public int getHoursOfTraining()
   {
      return hoursOfTraining;
   }
   public void work()
   {
      speak();
      System.out.println("I am a dog who works");
      System.out.println("I have " + hoursOfTraining +
         " hours of professional training!");
   }
}
```

Figure 11-28 The WorkingDog class

Interfaces vs Abstract classes

Feature	Abstract classes	Interfaces
Can instantiate objects from them?	No	No
Can contain non-abstract methods?	Yes	No
How many can be inherited from?	1	Unlimited
Can contain private attributes?	Yes	No
Can have getters and setters?	Yes	No

Section 6

Creating and Using Packages

Creating and Using Packages

- A **package** is a collection of classes e.g java.util
- Packages allow programmers to use other people's code without needing access to the source code.
- When you create a number of classes that inherit from another, it is convenient to place them in a package.
- If you don't specify a package, then any class is stored in a **default package**. If you specify a package, then any class must be public in order for others to use it.

Creating and Using Packages (Demo)

- To specify the package you want to place your class in, e.g com/course/animals, you must put the following at the top of your .java file:
  - package com.course.animals;
- Once you have compiled the java file, you can move the .class file to the specified directory.
- Alternatively, you can tell the java compiler to place your compiled class into the specified directory for you:
  - javac -d . Animal.java
- This will create the specified directory in the current directory (the full-stop) and then place your compiled class there.
- To use your compiled classes, use the *import* keyword:
  - e.g import com.course.animals.Dog;

Section 7

Lecture summary

Lecture summary

1. Creating and Using Abstract Classes
2. Using Dynamic Method Binding
3. Creating Arrays of Subclass Objects
4. Using the Object Class and Its Methods
5. Creating and Using Interfaces
6. Creating and Using Packages
7. Lecture summary

Questions

Thank you! Questions?