Java Object-Oriented Concepts Unit

Lesson 8: Specialization and Inheritance

The Learning House

427 S 4th Street #300

Louisville KY 40202

Lesson 8: Specialization and Inheritance

Overview

In this lesson, we will take a look at specialization and inheritance. In object-oriented languages, a class can be **derived** from another class. By doing this, the derived class **inherits** all of the fields and methods of the class from which it was derived. The **derived class** is also known as the **subclass, extended class,** or **child class** and the class from which it was derived is known as the **superclass, base class,** or **parent class**. Inheritance is a great tool for promoting code reuse and can help us to organize our code.

Specialization Through Inheritance

A well-designed inheritance hierarchy is built so the parent class is the most general (i.e. it contains all of the fields and methods common to its descendants) and the child classes **specialize** the parent.

For example, an Employee class may have the following methods:

- doWork
- createObjectives

All employees have these capabilities whether they are accountants, software developers, managers, or even the CEO. However, many employees have additional capabilities and responsibilities. For example, a Manager class has the following additional methods:

1. hire
2. fire
3. givePerformanceReview

It may also be the case that a manager may have to do something special when he/she creates objectives for the year, so we might also need a new implementation of the createObjectives method. As you can see, a Manager is a special kind of Employee — it inherits some common properties and behaviors but it also extends the functionality of Employee with new and/or different properties and behaviors.

This type of specialization hierarchy should look familiar to you — we see it in the world all the time. For example:

1. A dog is a special kind of mammal (which, in turn, is a special kind of animal)
2. A rose is a special kind of plant

It is quite common for us to say that a dog **is a** mammal and that a rose **is a** plant. In the same manner, we say that a Manager **is an** Employee. In Java, this specialization relationship is achieved via i**nheritance** and is implemented via the **extends** keyword.

Terminology

Inheritance is one area of object-oriented languages where terminology can be a bit confusing for newcomers. There are several ways of expressing the relationship between a base class and its descendants:

- Base Class:
    - Sometimes referred to as Superclass or Parent Class
    - When classes extend a base class, they inherit the properties and behaviors of the base class
- Derived Class:
    - Sometimes referred to as the Subclass, Extended Class, or Child Class
    - This class inherits the properties and behaviors of the base class when it extends the base class.
    - This class can add properties and behaviors to those of the base class
    - This class can override (i.e. provide its own implementation of) properties and behaviors of the base class
    - We say this this class specializes the base class

Code Reuse Through Inheritance

As mentioned in the previous section, a derived class inherits the properties and behaviors of the base class. That means that, if we have a group of objects (employees, for example) that all have common properties and behaviors, we can put the common properties and behaviors into a base class and then extend it into particular subclasses (Manager or SummerIntern would be subclasses of an Employee base class, for example).

This means we only have to write the common code one time. These subclasses each get all of the common code from the Employee class, in this example, and are free to add properties and behaviors for their particular purposes. The subclasses are also free to override properties or behaviors of the base class. In this way, inheritance promotes code reuse and is a great tool to help organize our code.

Method Overriding

We have already talked about method overloading — the ability to create methods with the same name but with different signatures. With inheritance comes the ability to override methods, which simply means that the child class will replace the implementation of a base class method with an implementation of its own.

For example, all employees in our system have the ability to set objectives via the setObjectives method. This method is implemented in the Employee base class and can be reused by all subclasses. It is possible that a Manager would require a different implementation for setObjectives — perhaps the Manager must set his/her own objectives *and* must set goals for each of his/her direct reports, for example. If this is the case, we would simply implement setObjectives (*same name and same signature*) in the Manager class, which would override (i.e. replace) the implementation contained in the Employee class. Please note a key difference between overriding and overloading:

- When you **override** a method in a child class, ***it must have the same signature*** as the corresponding method in the parent class.
- When you **overload** a method, ***it must have a unique signature***.

Access Control

So far we have seen public and private access to properties and methods. We now look at a new keyword: protected. If we mark a property or method of a base class protected, it means that property or method can be seen by the base class and all derived classes but it cannot be seen by any other class (in other words, it is as if that property or method were private with respect to other classes). If you think there is a good chance that a class will be extended in the future, it is good practice to mark the properties of the class **protected** instead of **private**.

Constructors

We know from past lessons that constructors are simply special methods that, when used in conjunction with the **new** keyword, get called when instantiating an object of a particular class. Thus far, we have only had to deal with one constructor because our classes have not extended a base class. Things get a bit more complicated with derived classes — now we have the constructor of the derived class and the constructor of the base class. Remember also that a derived class can extend a class that is itself a derived class, which means that we could be dealing with several constructors. The following are rules for dealing with constructors in derived classes (***you should memorize these rules***):

- The base class constructor can be invoked in a derived class by calling **super.**
- You can only call **super** from within the constructor of the derived class, *nowhere else.*
- The call to **super** must be the *first statement* in the constructor.
- The call to **super** must match the signature of a valid constructor in the base class.
- If you do not explicitly call **super** in the constructor of a derived class, the compiler will automatically call the base class default constructor. If one doesn't exists, a compilation error will occur.
- If your derived class does not define a constructor, the compiler provides the derived class with a default constructor that does nothing but call **super**, invoking the default constructor of the base class.

Polymorphism

Polymorphism is a pillar of object-oriented design. Polymorphism means "many-formed" and the key idea is that an object can take more than one form. The reason that an object can take more than one form stems from the idea discussed above — when an object extends another object, it creates an **"is-a"** relationship with the base class.

For example, a Manager **is-an** Employee, which means that we can use an Employee object reference to point to a Manager object. In other words, we can treat the Manager object like an Employee. Please keep in mind that the reverse is not true — a Manager has all of the capabilities and characteristics of an Employee (plus some) but not all Employees have the capabilities and characteristics of Managers. A real-world example of this would be the relationship between Mammals and Dogs — all Dogs are Mammals (and can be treated as such) but not all Mammals are Dogs.

Always keep in mind:

Derived types *are* base types

Calling Methods Polymorphically

The fact that we can override superclass methods in a subclass, along with the fact that we can use a superclass reference to point to a subclass object, leads to some interesting questions as to which version of the method will be invoked — the superclass version or the subclass version?

Here are the rules, using Employee and Manager (and the createObjectives method) as an example:

- If you have an instance of Employee pointed to by an Employee reference, the Employee version of createObjectives is called (obviously, because no other version exists).
- If you have an instance of Manager that ***has not*** overridden the Employee version of createObjectives and is pointed to by a Manager reference, the Employee version of createObjectives is called (again, this is straightforward because there is no other version).
- If you have an instance of Manager (as in the previous bullet) but it is pointed to by an Employee reference, the Employee version of createObjectives is called (it is still the only version of the method).
- If you have an instance of Manager that ***has*** overridden the Employee version of createObjectives and is pointed to by a Manager reference, the Manager version of createObjectives is called (this is also intuitive).
- If you have an instance of Manager that ***has*** overridden the Employee version of createObjectes and is pointed to by an Employee reference, ***the Manager version of createObjectives*** is called (perhaps not what you would expect).

Abstract Base Classes

The final object-oriented code organization tool that we'll talk about in this lesson is the abstract base class. An abstract base class is similar to a regular base class except for two things:

1. An abstract class cannot be instantiated — only subclasses of an abstract class can be instantiated.
2. An abstract base class can define methods (definition only — no implementation) and then force subclasses to provide an implementation.

These two features allow you to create a class that implements code common to potential subclasses (so the code can be reused) and forces subclasses to have certain behaviors (i.e. methods) and forces them to provide their own implementations of those behaviors.