Inheritance, Polymorphism, and Interfaces

Inheritance

- **Inheritance** enables the use of existing class to define a new class.
- A derived **class** is defined by adding instance variables and methods to an existing class. It is also known as **subclass**, **child class**, and **descendant class**.
- The existing class that the derived class is built upon is called the **base class**. It is also called **superclass**, **parent class**, and **ancestor class**.
- A derived class is defined by starting with another defined class and by adding methods and instance variables. The syntax for inheritance is:
  ```
  public class DerivedClass extends BaseClass { }
  ```
  Sample usage:
  ```
  public class Student extends Person { }
  ```
  The derived class is *Student* while the base class is *Person*. The *Student* class inherits the declared instance variables and public methods in the *Person* class except its constructor.
- If a derived class defines a method with the same name, the same number and types of parameters, and the same return type as a method in the base class, the definition in the derived class is said to **override** the definition in the base class.
- A derived class has its own constructors. It does not inherit any constructors from the base class.
- In the definition of a constructor for the derived class, the typical first action is to call a constructor of the base class.
- One way of defining a constructor is by calling another constructor in the same class. The default constructor calls another in that class by using the ***this*** keyword.
  ```
  public Person() {
     this(“No name yet”);
  }
  ```
  The default constructor *Person()* calls the other constructor.
  ```
  public Person(String initialName) {
     name = initialName;
  }
  ```
- Any use of the *super* keyword must be the first action in a constructor definition. Thus, a constructor definition cannot contain both a call using *super* and a call using *this*.
- If you want to include both calls, use *this* to call a constructor that has *super* as its first action.

Polymorphism

- **Polymorphism** is the ability of an object to take on many forms.
- Polymorphism allows you to make changes in the method definition for the derived classes and have those changes apply to the methods written in the base class.
- The most common use of polymorphism occurs when a base class reference is used to refer to a derived class object.
- When an overridden method is invoked, its action is the one defined in the class used to create the object using the new operator. It is not determined by the type of the variable naming the object.
- **Dynamic binding** allows many meanings to be associated to one method name.
- In dynamic binding, the definition of a method is not bound to an invocation of the method until run time when the method is called. This is the mechanism used by polymorphism.
- Example of polymorphism: *(Assume that Undergraduate inherits from the class Student.)*
  ```
  public class PolymorphismSample {
     public static void main (String[] args) {
        Person[] people = new Person[4];
        people[0] = new Undergraduate(“Velasquez, Veronica”, 4910, 1);
        people[1] = new Undergraduate(“Santos, Stephanie”, 9931, 2);
        people[2] = new Student(“Mendoza, Marie”, 8812);
  ```

```
            people[3] = new Undergraduate("Dela Cruz, Diane", 9901, 4);

              for (Person p : people) {
                 p.writeOutput();
                 System.out.println();
              }
          }
      }
```

Even though p is of type Person, the *writeOuput()* method associated with *Undergraduate* or *Student* is invoked depending upon which class was used to create the object.

Interface and Abstract Methods

- An **interface** is used to specify methods that a class must implement. It contains headings for a number of public methods.
- An interface should include comments that describe the methods.
- Different classes can implement the same interface. Here is the syntax:
  ```
  public class ClassName implements InterfaceName1, InterfaceName2 { }
  ```
  Sample usage:
  ```
  public class Rectangle implements Countable, Measurable { }
  ```
- An **abstract class** cannot be instantiated but can be subclassed. The syntax is:
  ```
  public abstract class ClassName { }
  ```
- An abstract class serves as a base for subclasses. It may or may not include abstract methods.
- An **abstract method** is a method that is declared as abstract and does not have an implementation. This is written without braces, followed by a semicolon.
- The syntax for the abstract method is:
  ```
  public abstract void methodName();
  ```
- Abstract classes are similar to interfaces. They both cannot be instantiated and may contain a mix of methods declared with or without an implementation.
- Consider using interfaces if any of these statements apply to your situation:
  - You expect that unrelated classes would implement your interface.
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  - You want to take advantage of multiple inheritance of type.
- Consider using abstract classes if any of these statements apply to your situation:
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract have many common methods or fields, or require access modifiers other than public (such as protected and private).
  - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

References:

Baesens, B., Backiel, A. & Broucke, S. (2015). *Beginning java programming: The object-oriented approach*. Indiana: John Wiley & Sons, Inc.

Farrell, J. (2014). *Java programming, 7th Edition*. Boston: Course Technology, Cengage Learning

Savitch, W. (2014). *Java: An introduction to problem solving and programming, 7th Edition*. California: Pearson Education, Inc.