Lesson 8
Polymorphism

Absolute Java

Walter Savitch

Polymorphism

- Inheritance allows you to define a base class and to define software for the base class.
- That software can then be used not only for objects of the base class but also for objects of any class derived from the base class.
- *Polymorphism* allows you to make changes in the method definition for the derived classes and to have those changes apply to the software written *in the base class.*

Polymorphism

Dynamic Binding and Inheritance

- Consider a program that uses the **Person, Student,** and **Undergraduate** classes. Let's say that we would like to set up a committee that consists of four people who are either students or employees. If we use an array to store the list of committee members, then it makes sense to make the array of type **Person** so it can accommodate any class derived from it.
- Here is a possible array declaration:

```
Person[ ] people = new Person[4];
```

Polymorphism

Dynamic Binding and Inheritance

- Next we might add objects to the array that represent members of the committee.
- In the example below we have added three objects of type **Undergraduate** and one object of type **Student** :

```
people[0] = new Undergraduate("Cotty, Manny", 4910, 1);
people[1] = new Undergraduate("Kick, Anita", 9931, 2);
people[2] = new Student("DeBanque, Robin", 8812);
people[3] = new Undergraduate("Bugg, June", 9901, 4);
```

Polymorphism

Dynamic Binding and Inheritance

- In this case we are assigning an object of a derived class (either **Student** or **Undergraduate**) to a variable defined as an ancestor of the derived class (**Person**).
- This is valid because **Person** encompasses the derived classes.
- In other words, **Student "is-a" Person and Undergraduate "is-a" Person**, so we can assign either one to a variable of type **Person**.

Polymorphism

Dynamic Binding and Inheritance

- Next, let's output a report containing information about all of the committee members.
- The report should be as detailed as possible. For example, if a **student** is an **undergraduate**, then the report should contain the **student's name, student number, and student level**.
- If the student is of type **Student**, then the report should contain the **name and student number**.

Polymorphism

Dynamic Binding and Inheritance

- There are three of them, one defined for **Undergraduate, Student, and Person.**
- Java recognizes that an object of type **Undergraduate** is stored in **people[0]**.
- As a result, even though **people[0]** is declared to be of type **Person**, the method associated with the class used to create the object is invoked.
- This is called **dynamic binding** or **late binding.**

Polymorphism

Dynamic Binding and Inheritance

- **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.
- A variable of any ancestor class can reference an object of a descendant class, but the object always remembers which method actions to use for every method name.
- The type of the variable does not matter. What matters is the class name when the object was created.

Polymorphism

- **Binding** refers to the process of associating a method definition with a method invocation.
- If the method definition is associated with the method invocation when the code is compiled, that is called **early binding**.
- If the method definition is associated with the method invocation when the method is invoked (at run time), that is called **late binding or dynamic binding.**

Polymorphism

- With dynamic, or late, binding the definition of a method is not bound to an invocation of the method until run time when the method is called.
- Polymorphism refers to the ability to associate many meanings to one method name through the dynamic binding mechanism.
- Thus, polymorphism and dynamic binding are really the same topic.

Polymorphism

- Java does not use late binding with private methods, methods marked final, or static methods.
- With private methods and final methods, this is not an issue because dynamic binding would serve no purpose anyway.
- When Java (or any language) does not use late binding, it uses static binding.
- With static binding, the decision of which definition of a method to use with a calling object is made at compile time based on the type of the variable naming the object.

Upcasting

- Assigning an object of a derived class to a variable of a base class (or any ancestor class) is called **upcasting** because it is like a type cast to the type of the base class.
- **Example (Sale – Base Class, DiscountSale – Derived Class):**

```
Sale saleVariable;
DiscountSale discountVariable = new DiscountSale("paint", 15, 10);
saleVariable = discountVariable; //upcasting
System.out.println(saleVariable.toString());
```

Downcasting

- When you do a type cast from a base case to a derived class (or from any ancestor class to any descendent class), it is called a **downcast**.
- Downcasting is troublesome and does not always make sense.
- For example, the downcast

```
Sale saleVariable = new Sale("paint", 15);
DiscountSale discountVariable;
discountVariable = (DiscountSale)saleVariable; //Error
```

Downcasting

- This does not make sense because the object named by saleVariable has no instance variable named discount and so cannot be an object of type DiscountSale.
- Every DiscountSale is a Sale, but not every Sale is a DiscountSale, as indicated by this example.
- It is your responsibility to use downcasting only in situations where it makes sense.

Downcasting

- The following statement

```
discountVariable = (DiscountSale)saleVariable;
```

- produces a run-time error but will compile with no error.
- However, the following, which is also illegal, produces a compile-time error:

```
discountVariable = saleVariable;
```

clone Method

- Every object inherits a method named **clone** from the class **Object**.
- The method **clone** has no parameters and is supposed to return a copy of the calling object.
- However, the inherited version of **clone** was not designed to be used as is. Instead, you are expected to override the definition of **clone** with a version appropriate for the class you are defining.

clone Method

- As with other methods inherited from the class **Object**, the method **clone** needs to be redefined (overridden) before it performs properly**.**
- The heading for the method clone in the class Object is as follows:

```
protected Object clone()
```

- The following definition of the clone method can be added to the definition of Sale

```
public Sale clone()
{
   return new Sale(this);
}
```

Abstract Classes

- **An abstract class is a class that has some methods without complete definitions.**
- **You cannot create an object using an abstract class constructor, but you can use an abstract class as a base class to define a derived class.**
- An abstract class is a class that has one or more abstract methods.
- An abstract method is a method that has no body.
- We create abstract methods when we plan to override them in the derived class.

Abstract Classes

- An abstract method has a heading just like an ordinary method, but no method body.
- The syntax rules of Java require the modifier abstract and require a semicolon in place of the missing method body, as illustrated by the following:

```
public abstract void drawHere();
```

- **In Java, an abstract method cannot be private.**
- **Normally an abstract method is public but protected, and package (default) access is allowed.**

Abstract Classes

- **An abstract class is created when we want to use it as a base class.**
- When we want to force programmers to define a method.
- We do this by including the keyword abstract in the heading of the class definition, as in the following:

```
public abstract class ShapeBase
{ . . .
```

- If the subclass of an abstract class does not define the abstract method, the subclass cannot be instantiated and must be declared as abstract.

Abstract Classes

- **Example of Abstract class:**

```
// Abstract base class for Athlete
public abstract class Athlete
{
//Private instance fields
public abstract String getEquipment(); //Abstract Method
public Athlete(String firstname,String lastname, String  sportPlayed) //Constructor
{
      myFirstName=firstname;
      myLastName=lastName;
      mySport=sportPlayed;
      myHoursTraining=0;
}
//more methods
}
```

Abstract Classes

- Although the class Athlete is abstract, not all of its methods are abstract.
- Constructor is defined for it.
- It can have more methods defined.
- When it makes sense to define a body for a method in an abstract class, it should be given.
- That way, as much detail as possible is pushed into the abstract class, so that such detail need not be repeated in each derived class.

Abstract Classes

- **An abstract class makes it easier to define a base class by relieving you of the obligation to write useless method definitions.**
- If a method will always be overridden, make it an abstract method—and so make the class abstract.
- An abstract method serves a purpose, even though it is not given a full definition. It serves as a placeholder.
- for a method that must be defined in all (non abstract) derived classes.

Abstract Classes

- For example, the subclass **Runner** might define **getEquipment()** in the following way:

```
public String getEquipment()
{
    return “Running shoes”;
}
```

Check your understanding

1. Can a method definition include an invocation of an abstract method?
2. Can you have a variable whose type is an abstract class?
3. Can you have a parameter whose type is an abstract class?
4. Is it legal to have an abstract class in which all methods are abstract?
5. Why do we bother to have any constructors in an abstract class? Aren't they useless?