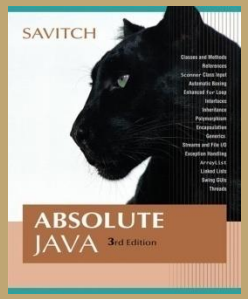
Comp 249
Programming Methodology
Chapter 8 - *Polymorphism*

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of Absolute Java 3rd Edition by Savitch; which has originally been prepared by Rose Williams of Binghamton University. Absolute Java is published by Pearson Education / Addison-Wesley.



Introduction to Polymorphism

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism
- Polymorphism is the ability to associate many meanings to one method name
  - It does this through a special mechanism known as *late binding* or *dynamic binding*

Introduction to Polymorphism

- Inheritance allows a base class to be defined, and other classes derived from it
  - Code for the base class can then be used for its own objects, as well as objects of any derived classes
- Polymorphism allows changes to be made to method definitions in the derived classes, *and have those changes apply to the software written for the base class*

Late Binding

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

Late Binding

- Java uses late binding for all methods (except private, `final`, and static methods)

- Because of late binding, a method can be written in a base class to perform a task, even if portions of that task aren't yet defined

See Polymorphism1.java

Pitfall: No Late Binding for Static Methods

- When the decision of which definition of a method to use is made at compile time, that is called *static binding*
  - This decision is made based on the *type of the variable naming the object*
- Java uses static, not late, binding with private, **`final`**, and static methods
  - In the case of **`private`** and **`final`** methods, late binding would serve no purpose (these methods cannot be overridden, so only one version exists)
  - (Warning:) However, in the case of a *static* method invoked using a calling object, it does make a difference

See Polymorphism2.java

Pitfall: No Late Binding for Static Methods

Example (*See Polymorphism2.java*):

- The **`Vehicle`** class **`DisplayNumberOfCreatedObjects ()`** method:

```
public static void DisplayNumberOfCreatedObjects()
{
System.out.println("The number of created Vehicle objects so
   far is " + numOfCreatedObjects + ".");
}
```

- The **`Bus`** class **`DisplayNumberOfCreatedObjects ()`** method:

```
public static void DisplayNumberOfCreatedObjects()
{
System.out.println("The number of created Bus objects so far
   is " + numOfCreatedObjects + ".");
}
```

Pitfall: No Late Binding for Static Methods

Example (*See Polymorphism2.java*) – Continues:

- In the previous example, the object **`v1`** was created from the **`Vehicle`** class, and the object **`b1`** was created from the **`Bus`** class,

- Given the following assignment:

```
v1 = b1;
```

  - Now the two variables point to the same object

Pitfall: No Late Binding for Static Methods

Example (*See Polymorphism2.java*) – Continues:

- Given the invocations:

```
v1.DisplayNumberOfCreatedObjects();
b1.DisplayNumberOfCreatedObjects();
```

The output is:

```
The number of created Vehicle objects so far is 11.
The number of created Bus objects so far is 3.
```

- Note that here, **`DisplayNumberOfCreatedObjects`** is a static method invoked a calling object (i.e. v1, b1) (instead of its class name)
  - Therefore the exact executed method is determined by its variable name, not the object that it references

Pitfall: No Late Binding for Static Methods

- There are other cases where a static method has a calling object in a more inconspicuous way
- For example, a static method can be invoked within the definition of a nonstatic method, but without any explicit class name or calling object
- In this case, the calling object is the implicit **`this`**

The `final` Modifier

- A *method* marked **`final`** indicates that it cannot be overridden with a new definition in a derived class
  - If **`final`**, the compiler can use early binding with the method

```
public final void someMethod() { . . . }
```

- A *class* marked **`final`** indicates that it cannot be used as a base class from which to derive any other classes

Upcasting and Downcasting

- *Upcasting* is when an object of a derived class is assigned ***to*** a variable of a base class (or any ancestor class)

```
Vehicle v1 = new Vehicle(); //Base class object
Bus b1 = new Bus(2, 55000, 37); //Derived class object
v1 = b1;
```

- *Downcasting* is when a ***type cast*** is performed from a base class to a derived class (or from any ancestor class to any descendent class)
  - Downcasting has to be done very carefully
  - In many cases it doesn't make sense, or is illegal:

```
B1 = v1;          //will produce compiler error
B1 = (Bus)v1;     //will produce run-time error
```

  - There are times, however, when downcasting is necessary, e.g., inside the `equals` method for a class

See Polymorphism3.java

Revisit Object3.java

Pitfall: Downcasting

- It is the responsibility of the programmer to use downcasting only in situations where it makes sense
  - The compiler does not check to see if downcasting is a reasonable thing to do
- Using downcasting in a situation that does not make sense usually results in a run-time error

Tip: Checking to See if Downcasting is Legitimate

- Downcasting to a specific type is only sensible if the object being cast is an instance of that type
  - This is exactly what the **`instanceof`** operator tests for:

    `object` **`instanceof`** ***`ClassName`***

  - It will return true if ***`object`*** is of type ***`ClassName`***; in particular, it will return true if ***`object`*** is an instance of any descendent class of ***`ClassName`***

Pitfall: Limitations of Copy Constructors

- A copy constructor is supposed to create a good copy of a new object from an existing one
- However, when polymorphism is used, a copy constructor may have a strong limitation

See Polymorphism4.java

A First Look at the `clone` Method

- Every object inherits a method named **`clone`** from the **`Object`** class
  - The method **`clone`** has no parameters
  - It is supposed to return a *deep copy* of the calling object
- However, the inherited version of the method is not designed to be used as is
  - Instead, each class is expected to override it with a more appropriate version

A First Look at the `clone` Method

- The heading for the **`clone`** method defined in the **`Object`** class is as follows:

```
protected Object clone()
```

- The heading for a **`clone`** method that overrides the **`clone`** method in the **`Object`** class can differ somewhat from the heading above
  - A change to a more permissive access, such as from protected to public, is always allowed when overriding a method definition
  - Changing the return type from **`Object`** to the type of the class being cloned is allowed because every class is a descendent class of the class **`Object`**
  - This is an example of a *covariant* return type

A First Look at the `clone` Method

- If a class has a copy constructor, the **`clone`** method for that class can use the *copy constructor* to create the copy returned by the **`clone`** method

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

and another example:

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

See Polymorphism5.java

Pitfall: Limitations of Copy Constructors

- Although the clone() methods may in fact use the copy constructors to perform the copying, this works because the method **`clone`** has the same name in all classes, and polymorphism works with method names
  - The copy constructors (i.e **`Vehicle`**, **`Bus`**, **`RaceCar`**) have different names, and polymorphism doesn't work with methods of different names

Pitfall: Sometime the `clone` Method Return Type is `Object`

- Prior to version 5.0, Java did not allow covariant return types, so no changes whatsoever were allowed in the return type of an overridden method

- Therefore, the **`clone`** method for all classes had **`Object`** as its return type
- Consequently, the **`clone`** method for any class, i.e. the **`Vehicle`** class would have looked like this:

```
public Object clone()
{
  return new Vehicle(this);
}
```

- Therefore, the result needed to always be type casted when using a **`clone`** method written for an older version of Java

```
Vehicle newVec = (Vehicle)originalVec.clone();
```

Pitfall: Sometime the `clone` Method Return Type is `Object`

- It is still perfectly legal to use **`Object`** as the return type for a clone method, even with classes defined after Java version 5.0
  - When in doubt, it causes no harm to include the type cast
  - For example, the following is legal for the clone method of the Vehicle class:

```
Vehicle newVec = originalVec.clone();
```

  - However, adding the following type cast produces no problems:

```
Vehicle newVec = (Vehicle)originalVec.clone();
```

Abstract Class

- Sometimes, it is does NOT make sense to create objects from specific classes
- In such case, these classes should be created as *abstract*
- An abstract class can only be used to derive other classes; you cannot create objects from an abstract class
- An abstract class must have at least one *abstract method*

Abstract Method

- An abstract method has a complete method heading, to which has been added the modifier **`abstract`**
- It has no method body, and ends with a semicolon in place of its body

```
public abstract long getSerNumber();
```

- An abstract method cannot be private

See Abstract1.java

Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods
- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add **`abstract`** to its modifier

- A class that has no abstract methods is called a *concrete class*

Pitfall: You Cannot Create Instances of an Abstract Class

- An abstract class constructor cannot be used to create an object of the abstract class
  - However, a derived class constructor will include an invocation of the abstract class constructor in the form of **`super`**
- Although an object of an abstract class cannot be created, it is perfectly fine to have a parameter of an abstract class type
  - This makes it possible to plug in an object of any of its descendent classes