Polymorphism

Introduction to Programming and Computational Problem Solving - 2
CSE 8B
Lecture 11
Announcements

• Assignment 5 is due today, 11:59 PM
• Quiz 5 is Nov 4
• Assignment 6 will be released today
  – Due Nov 9, 11:59 PM
• Educational research study
  – Nov 4, weekly survey
• Reading
  – Liang
    • Chapter 11
Inheritance

• Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses)

• A subclass inherits from a superclass
  – For example, both a circle and a rectangle are geometric objects
    • GeometricObject is a superclass
    • Circle is a subclass of GeometricObject
    • Rectangle is a subclass of GeometricObject

• Models is-a relationships
  – For example
    • Circle is-a GeometricObject
    • Rectangle is-a GeometricObject
Polymorphism

• Remember, a class defines a type

• A type defined by a subclass is called a subtype, and a type defined by its superclass is called a supertype
  – For example
    • Circle is a subtype of GeometricObject, and GeometricObject is a supertype for Circle

• **Polymorphism** means that a variable of a supertype can refer to a subtype object
  – Greek word meaning “many forms”
Polymorphism

• An object of a *subtype* can be used wherever its *supertype* value is required
  – For example
  • Method `m` takes a parameter of the `Object` type, so you can invoke it with any object

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}
class Student extends Person {
}
class Person {
}
```

![Object](Object) ← [Person](Person) ← [Student](Student) ← [GraduateStudent](GraduateStudent)
Declared type and actual type

- The type that declares a variable is called the variable’s *declared type*.
- The actual class for the object referenced by the variable is called the *actual type* of the variable.
- Remember, a variable of a reference type can hold a null value or a reference to an instance of the declared type.

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}
class Student extends Person {
}
class Person {
}
```
Declared type and actual type

- In all executions of `m`, the variable `x`'s declared type is `Object`.
- In the first execution of `m`, the variable `x`'s actual type is `GraduateStudent`.
- In the second execution of `m`, the variable `x`'s actual type is `Student`.
- In the third execution of `m`, the variable `x`'s actual type is `Person`.
- In the fourth execution of `m`, the variable `x`'s actual type is `Object`.

```java
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
}

class Person {
}
```
Dynamic binding

• When the method \( m \) is executed, the argument \( x \)'s toString method is invoked
• \( x \) may be a reference to an instance of GraduateStudent, Student, Person, or Object
• Classes Student, Person, and Object have their own implementation of the toString method
• Which implementation is used will be determined dynamically by the JVM at runtime
• This capability is known as dynamic binding

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person {
    public String toString() {
        return "Person";
    }
}
```

Method overridden in subclasses
**Dynamic binding**

- Suppose an object \( o \) is an instance of classes \( C_1, C_2, ... , C_{n-1}, \) and \( C_n \), where \( C_1 \) is a subclass of \( C_2 \), \( C_2 \) is a subclass of \( C_3, ... , \) and \( C_{n-1} \) is a subclass of \( C_n \)
  - That is, \( C_n \) is the most general class, and \( C_1 \) is the most specific class
- In Java, \( C_n \) is the Object class
- If object \( o \) invokes a method \( p \), the JVM searches the implementation for the method \( p \) in \( C_1, C_2, ... , C_{n-1} \) and \( C_n \), in this order, until it is found
- Once an implementation is found, the search stops and the first-found implementation is invoked

Since \( o \) is an instance of \( C_1, o \) is also an instance of \( C_2, C_3, ... , C_{n-1}, \) and \( C_n \)
Matching and binding

• **Matching** a method *signature*
  – The *declared type* of the reference variable decides which method to match at *compile time*

• **Binding** a method *implementation*
  – A method may be implemented in several classes along the inheritance chain
  – The *actual type* of the reference variable decides which implementation of the method the JVM dynamically binds at *runtime*
Matching and binding

• In all executions of `m`, the variable `x`’s declared type is `Object`
• In the first execution of `m`, the variable `x`’s actual type is `GraduateStudent`
• In the second execution of `m`, the variable `x`’s actual type is `Student`
• In the third execution of `m`, the variable `x`’s actual type is `Person`
• In the fourth execution of `m`, the variable `x`’s actual type is `Object`

```java
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}
```

```java
class GraduateStudent extends Student {
}
class Student extends Person {
    public String toString() {
        return "Student";
    }
}
class Person {
    public String toString() {
        return "Person";
    }
}
```

Matching at compile time
Method overridden in subclasses
Binding at runtime
Casting objects

• You have been using the casting operator to convert variables of one primitive type to another

• Casting can also be used to convert an object of one class type to another within an inheritance hierarchy
  – This is called *casting object*
Upcasting is implicit

• The statement
  
m(new Student());

  is equivalent to
  
  Object o = new Student();
  m(o);

• It is always possible to cast an instance of a subclass to a variable of a superclass
  – This is called upcasting
Downcasting

• Warning: if you find yourself wanting to perform (explicit) downcasting from a superclass to a subclass, it is a sign you are likely approaching things the wrong way!

• Override methods in subclasses instead
Downcasting

- **Downcasting is such a bad practice** that explicit casting must be used to confirm your intention to the compiler.

- For example:
  ```java
  Object o = new Student();
  m(o);
  Student b = o; // Compile error
  Student c = (Student)o; // No error
  ```
Downcasting

• If you are downcasting a superclass object to an object that is not an instance of a subclass, then a runtime exception occurs.

• Use the `instanceof` operator to avoid this
  – For example
    ```java
    void someMethod(Object myObject) {
        ... // Some lines of code
        // Perform casting if myObject is an instance of Circle
        if (myObject instanceof Circle) {
            System.out.println("The circle diameter is "+
                        ((Circle)myObject).getDiameter());
        ... // Some lines of code
    }
    ```

“Safe” downcasting

Explicit casting
Override equals method in Object

• Remember, usually a class should override the `toString` method so it returns a digestible string representation of the object
• You may also want to override the `equals` method
  – One of the few reasonable times to use downcasting
Override equals method in Object

- For example
  
  ```java
  public class Circle extends GeometricObject {
      private double radius;
      ...
      public boolean equals(Circle circle) {
          return this.radius == circle.radius;
      }

      @Override
      public boolean equals(Object o) {
          if (o instanceof Circle) {
              return radius == ((Circle)o).radius;
          } else {
              return false;
          }
  }
  ```

  "Safe" downcasting
## Methods and data fields visibility

<table>
<thead>
<tr>
<th>Modifiers on Members in a Class</th>
<th>Accessed from the Same Class</th>
<th>Accessed from the Same Package</th>
<th>Accessed from a Subclass in a Different Package</th>
<th>Accessed from a Different Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Default (no modifier)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Private</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Covered in 2 weeks
Subclass and visibility/accessibility

• If desired, a subclass can increase accessibility of a method defined in the superclass, but a subclass cannot decrease accessibility of a method defined in the superclass
  – For example, a subclass may override a protected method in its superclass and change its visibility to public
  – For example, if a method is defined as public in the superclass, it must be defined as public in the subclass
Preventing extending and overriding

• You may occasionally want to prevent classes from being extended
• In such cases, use the `final` modifier to indicate a class is final and cannot be a parent class
The final modifier

• A final class cannot be extended
  – For example
    ```java
    final class Math {
        ...
    }
    ```

• A final method cannot be overridden by its subclasses

• And remember, a final variable is a constant
  – For example
    ```java
    final static double PI = 3.14159;
    ```
The final modifier

• Modifiers are used on classes and class members (data and methods), except the final modifier can also be used on local variables in a method
• A final local variable is a constant inside a method
• A best practice is to use final variables liberally
Modifiers

• Access modifiers
  – For classes
    • public and default (no modifier)
  – For methods (including constructors) and data fields
    • public, protected, default (no modifier), and private

• Non-access modifiers
  – For classes
    • final and abstract (covered in two weeks)
  – For methods (excluding constructors)
    • final, static, and abstract (covered in two weeks)
  – For data fields
    • final and static

• All modifiers
  – Liang, appendix D
The ArrayList class

• You can create an array to store objects, but the array’s size is fixed once the array is created
• Java provides the ArrayList class that can be used to store an unlimited number of objects
The `ArrayList` class

```java
java.util.ArrayList<E>

+ArrayList()  
+add(o: E) : void  
+add(index: int, o: E) : void 
+clear(): void  
+contains(o: Object): boolean  
+get(index: int) : E  
+indexOf(o: Object) : int  
+isEmpty(): boolean  
+lastIndexOf(o: Object) : int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int) : boolean  
+set(index: int, o: E) : E
```

Creates an empty list.
Appends a new element `o` at the end of this list.
Adds a new element `o` at the specified index in this list.
Removes all the elements from this list.
Returns true if this list contains the element `o`.
Returns the element from this list at the specified index.
Returns the index of the first matching element in this list.
Returns true if this list contains no elements.
Returns the index of the last matching element in this list.
Removes the element `o` from this list.
Returns the number of elements in this list.
Removes the element at the specified index.
Sets the element at the specified index.
The ArrayList class

- ArrayList is known as a generic class with a generic type E
- You can specify a concrete type to replace E when creating an ArrayList
- For example
  - The below statements creates an ArrayList used to store strings and assigns its reference to variable cities
    ```java
    ArrayList<String> cities = new ArrayList<String>();
    ArrayList<String> cities = new ArrayList<>();
    ```
## Comparing arrays and ArrayList

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>ArrayList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating an array/ArrayList</td>
<td><code>String[] a = new String[10]</code></td>
<td><code>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;();</code></td>
</tr>
<tr>
<td>Accessing an element</td>
<td><code>a[index]</code></td>
<td><code>list.get(index)</code></td>
</tr>
<tr>
<td>Updating an element</td>
<td><code>a[index] = &quot;London&quot;;</code></td>
<td><code>list.set(index, &quot;London&quot;);</code></td>
</tr>
<tr>
<td>Returning size</td>
<td><code>a.length</code></td>
<td><code>list.size()</code></td>
</tr>
<tr>
<td>Adding a new element</td>
<td></td>
<td><code>list.add(&quot;London&quot;);</code></td>
</tr>
<tr>
<td>Inserting a new element</td>
<td></td>
<td><code>list.add(index, &quot;London&quot;);</code></td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td><code>list.remove(index);</code></td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td><code>list.remove(Object);</code></td>
</tr>
<tr>
<td>Removing all elements</td>
<td></td>
<td><code>list.clear();</code></td>
</tr>
</tbody>
</table>
Array to/from ArrayList

- Creating an `ArrayList` from an array of objects
  
  ```java
  String[] array = {"red", "green", "blue"};
  ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
  ```

- Creating an array of objects from an `ArrayList`
  
  ```java
  String[] array1 = new String[list.size()];
  list.toArray(array1);
  ```
Useful methods in `java.util.Collections`

- Maximum element in `ArrayList`
  ```java
  java.util.Collections.max
  ```
- Minimum element in `ArrayList`
  ```java
  java.util.Collections.min
  ```
- Sort an `ArrayList`
  ```java
  java.util.Collections.sort
  ```
- Shuffle an `ArrayList`
  ```java
  java.util.Collections.shuffle
  ```
Next Lecture

• Exception handling
• Reading
  – Liang
    • Chapter 12