Polymorphism

Introduction to Programming and Computational Problem Solving - 2
CSE 8B
Lecture 10
Announcements

• Assignment 4 is due today, 11:59 PM
• Assignment 5 will be released today
  – Due May 6, 11:59 PM
• Midterm is May 2
• 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

```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 extends Object {
}
```
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

```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 extends Object {
}
```
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`
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.

```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 {
    public String toString() {
        return "Student";
    }
}

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

Method overridden in subclasses.
Dynamic binding

- Suppose an object o is an instance of classes C₁, C₂, ..., C_{n-1}, and Cₙ, where C₁ is a subclass of C₂, C₂ is a subclass of C₃, ..., and C_{n-1} is a subclass of Cₙ
  - That is, Cₙ is the most general class, and C₁ is the most specific class
- In Java, Cₙ is the Object class
- If object o invokes a method p, the JVM searches the implementation for the method p in C₁, C₂, ..., C_{n-1} and Cₙ, 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 C₁, o is also an instance of C₂, C₃, ..., C_{n-1}, and Cₙ
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
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 extends Object {
    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

CSE 8B, Spring 2022
Upcasting is implicit

- The statement
  \[ \text{m(new Student());} \]
  is equivalent to
  \[ \text{Object o = new Student(); m(o);} \]
- It is always possible to
  cast an instance of a
  subclass to a variable of
  a superclass
  – This is called \textit{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

  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
    }
    ```
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>
A subclass cannot lessen the accessibility

• A subclass may override a protected method in its superclass and change its visibility to public

• However, a subclass cannot lessen the accessibility of a method defined in the superclass
  – 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
    final class Math {
      ...
    }

• A final method cannot be overridden by its subclasses

• And remember, a final variable is a constant
  – For example
    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
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

CSE 8B, Spring 2022 25
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.util.ArrayList<E>
+ArrayList()  
+add(o: E) : void  
+add(index: int, o: E) : void 
+clear(): void ... number of elements in this list. 
Removes the element at the specified index. 
Sets the element at the specified index.
```

- `ArrayList()` Creates an empty list.
- `add(o: E) : void` Appends a new element `o` at the end of this list.
- `add(index: int, o: E) : void` Adds a new element `o` at the specified index in this list.
- `clear(): void` Removes all the elements from this list.
- `contains(o: Object): boolean` Returns true if this list contains the element `o`.
- `get(index: int) : E` Returns the element from this list at the specified index.
- `indexOf(o: Object) : int` Returns the index of the first matching element in this list.
- `isEmpty(): boolean` Returns true if this list contains no elements.
- `lastIndexOf(o: Object) : int` Returns the index of the last matching element in this list.
- `remove(o: Object): boolean` Removes the element `o` from this list.
- `size(): int` Returns the number of elements in this list.
- `remove(index: int) : boolean` Removes the element at the specified index.
- `set(index: int, o: E) : E` 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>String[] a = new String[10]</td>
<td>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;()</td>
</tr>
<tr>
<td>Accessing an element</td>
<td>a[index]</td>
<td>list.get(index);</td>
</tr>
<tr>
<td>Updating an element</td>
<td>a[index] = &quot;London&quot;;</td>
<td>list.set(index, &quot;London&quot;);</td>
</tr>
<tr>
<td>Returning size</td>
<td>a.length</td>
<td>list.size();</td>
</tr>
<tr>
<td>Adding a new element</td>
<td></td>
<td>list.add(&quot;London&quot;);</td>
</tr>
<tr>
<td>Inserting a new element</td>
<td></td>
<td>list.add(index, &quot;London&quot;);</td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td>list.remove(index);</td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td>list.remove(Object);</td>
</tr>
<tr>
<td>Removing all elements</td>
<td></td>
<td>list.clear();</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.util.Collections.max
• Minimum element in ArrayList
  java.util.Collections.min
• Sort an ArrayList
  java.util.Collections.sort
• Shuffle an ArrayList
  java.util.Collections.shuffle
Next Lecture

• Exception handling
• Reading
  – Liang
    • Chapter 12