Chapter 11 Inheritance and Polymorphism

Motivation

Suppose you want to define classes to model circles, rectangles, and triangles

These classes have many common features

QUESTION: What is the best way to design these classes so to avoid redundancy?

ANSWER: Use inheritance
Objectives

- To define a subclass from a superclass through inheritance (§11.2).
- To invoke the superclass’s constructors and methods using the super keyword (§11.3).
- To override instance methods in the subclass (§11.4).
- To distinguish differences between overriding and overloading (§11.5).
- To explore the toString() method in the Object class (§11.6).
- To discover polymorphism and dynamic binding (§§11.7–11.8).
- To describe casting and explain why explicit downcasting is necessary (§11.9).
- To explore the equals method in the Object class (§11.10).
- To store, retrieve, and manipulate objects in an ArrayList (§11.11).
- To implement a Stack class using ArrayList (§11.12).
- To enable data and methods in a superclass accessible from subclasses using the protected visibility modifier (§11.13).
- To prevent class extending and method overriding using the final modifier (§11.14).

Inheritance

- We use a class to model objects of the same type
- Some classes have some common properties & behaviors, which can be generalized in a class that can be shared by other classes
- We can define a new specialized class (subclass) that extends the existing generalized class (superclass). This is called class inheritance
- The subclasses inherit the properties & methods (except constructors) from the superclass

```java
class Fruit {
    // code
}

class Apple extends Fruit {
    // code
}
```
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.

Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.

The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.

Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.

Using the Keyword super

- The keyword super refers to the superclass of the class in which super appears

- This keyword can be used in two ways:
  1. To call a superclass constructor
  2. To call a superclass method

- You must use the keyword super to call the superclass constructor. Invoking a superclass constructor’s name in a subclass causes a syntax error

- Java requires that the statement that uses the keyword super appear first in the constructor
Are Superclass’s Constructor Inherited?

No. They are invoked explicitly or implicitly.

- They can be invoked from the subclasses’ constructors, using the keyword super
- If the keyword super is not explicitly used, the superclass’s accessible no-arg constructor is automatically invoked

Superclass’s Constructor is Always Invoked

A constructor may invoke an overloaded constructor or its superclass’s constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```
Constructor Chaining

Constructing an instance of a class invokes all the superclasses’ constructors along the inheritance chain. This is known as constructor chaining.

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

1. Start from `main()`
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }
    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }
    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
Trace Execution

```java
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee’s no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person’s no-arg constructor is invoked");
    }
}
```

4. Invoke Employee(String) constructor

5. Invoke Person() constructor
Trace Execution

```java
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

6. Execute `println`

7. Execute `println`
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee’s overloaded constructor");
        System.out.println("(3) Employee’s no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person’s no-arg constructor is invoked");
    }
}
Example of the Impact of a Superclass without a *no-arg constructor*

Find out the error in the following program:

```java
class Fruit {
    public Fruit(String name) {
        System.out.println("Fruit's constructor is invoked");
    }
}

public class Apple extends Fruit {
}
```

A compile-time error will be generated indicating the need for a *no-arg constructor* in the superclass.

Properties & Methods of Subclasses

- A subclass inherits properties and methods (except constructors) from a superclass

- You can also:
  - Add new properties to the subclass
  - Add new methods to the subclass
  - Override (i.e. redefine) the implementation of the methods defined in the superclass. To *override* a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass
Calling Superclass Methods

```java
class Circle extends GeometricObject {
    // Other code

    /** Use toString() defined in GeometricObject */
    public void printCircle() {
        return super.toString();
    }
}
```

The `super` keyword in this case is optional because the subclass has inherited `toString()` from the superclass.

```java
class Circle extends GeometricObject {
    // Other code

    /** Override toString() defined in GeometricObject */
    @Override
    public String toString() {
        return super.toString() + "\radius is " + radius;
    }
}
```

The `super` keyword in this case is necessary because out of the two versions of `toString()`, we are referring to superclass version here.

Overriding Methods in the Superclass

- An instance method can be overridden only if it is accessible. Thus a `private` method cannot be overridden, because it is not accessible outside its own class.

- If a method defined in a subclass is `private` in its superclass, the two methods are completely unrelated.

- Like an instance method, a `static` method can be inherited. However, a `static` method cannot be overridden.

- If a `static` method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.
Overriding vs. Overloading

```java
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}

class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}

class A extends B {
    // This method overrides the method in B
    @Override
    public void p(double i) {
        System.out.println(i);
    }
}
```

| Same name/arguments, different body | Same name, different arguments/body |

The **Object** Class and its Methods

- Every class in Java is descended from the `java.lang.Object` class
- If no inheritance is specified when a class is defined, the superclass of the class is **Object**

```java
public class Circle {
    ...
}
```

Equivalent

```java
public class Circle extends Object {
    ...
}
```
**toString() in Object**

- `toString()` returns a `String` representation of the object

- The default implementation of `toString()` returns a `String` consisting of a **class name** of which the object is an instance, the at sign `@`, and a **number** representing this object

  ```java
  Loan loan = new Loan();
  System.out.println(loan.toString());
  ```

  displays `Loan@15037e5`

- This message is not very helpful or informative. Usually you should override `toString()` so that it returns an informative `String` representation of the object

---

**Polymorphism**

Polymorphism means that a variable of a supertype can refer to a subtype object

- 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**
- Polymorphism means that we can pass an instance of a subclass to a parameter of its superclass type (but not vice versa!)
- In the following demo, `Circle` is a **subtype** of `GeometricObject` and `GeometricObject` is a **supertype** for `Circle`
Polymorphism, Dynamic Binding & Generic Programming

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

m() takes a parameter of the Object type. You can invoke it with any object type. An object of a subtype can be used wherever its supertype required. This feature is known as polymorphism.

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

Dynamic Binding

- Suppose an object o is an instance of classes C₁, C₂, ..., Cₙ₋₁, and Cₙ, where C₁ is a subclass of C₂, C₂ is a subclass of C₃, ..., and Cₙ₋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 o invokes p(), the JVM searches the implementation for the p() in C₁, C₂, ..., Cₙ₋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, ..., Cₙ₋₁, and Cₙ.
Method Matching vs. Binding

- Matching a method signature and binding a method implementation are two separate issues.

- The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses.

- JVM dynamically binds the implementation of the method at runtime.

---

Generic Programming

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

- Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as **generic programming**.

- If a method’s parameter type is a superclass (e.g., `Object`), you may pass an object to this method of any of the parameter’s subclasses (e.g., `Student` or `String`).

- When an object (e.g., a `Student` object or a `String` object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., `toString()`) is determined dynamically.
Casting Objects

We have used 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. In the preceding section, the statement

```java
m(new Student());
```

assigns the object `new Student()` to a parameter of the `Object` type. This statement is equivalent to:

```java
Object o = new Student(); // Implicit casting
m(o);
```

The statement `Object o = new Student()`, known as implicit casting, is legal because an instance of `Student` is automatically an instance of `Object`.

Why Casting is Necessary?

- A compile error would occur if you assign the object reference `o` of `Object` type to a variable of the `Student` type using the statement: `Student b = o;`

- Why does the statement `Object o = new Student()` work and the statement `Student b = o` doesn’t?

- This is because a `Student` object is always an instance of `Object`, but an `Object` is not necessarily an instance of `Student`. Even though we can see that `o` is really a `Student` object, the compiler is not so clever to know it.

- To tell the compiler that `o` is a `Student` object, use an explicit casting. Enclose the target object type in parentheses and place it before the object to be cast: `Student b = (Student)o;`
**Downcasting from Superclass to Subclass**

- Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

  ```java
  Apple x = (Apple)fruit; // successful if fruit is of type Apple
  Fig x = (Fig)fruit; // causes error if fruit is of type Apple
  ```

- **Upcasting** always succeeds.

  ```java
  Fruit x = apple; // apple is of type Apple
  Fruit x = fig; // fig is of type Fig
  ```

---

**The `instanceof` Operator**

Use the `instanceof` operator to test whether an object is an instance of a class.

```java
Object myObject = new Circle();
// Some lines of code

/** Cast if myObject is an instance of Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
          ((Circle)myObject).getDiameter());
  // Code
}
```
Example: Polymorphism & Casting

This example creates two geometric objects: a circle, and a rectangle, invokes displayGeometricObject() to display the objects.

The displayGeometricObject() displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

equals()

equals() compares the contents of two objects. The default implementation of the equals() in the Object class is as follows:

```java
public boolean equals(Object obj) {
    return this == obj;
}
```

equals() is overridden in the Circle class:

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

- The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references.

- equals() is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects.

- The == operator is stronger than equals(), in that the == operator checks whether the two reference variables refer to the same object.

The ArrayList Class

We 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 supports dynamic arrays that can grow if needed.

```
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.
ArrayList is a Generic 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

Example: The following statement creates an ArrayList and assigns its reference to variable cities. This ArrayList object can be used to store strings

```
ArrayList<String> cities = new ArrayList<>();
```

Arrays vs. 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>Returning size</td>
<td>a.length</td>
<td>list.size();</td>
</tr>
<tr>
<td>Adding a new element</td>
<td></td>
<td>list.add(“London”);</td>
</tr>
<tr>
<td>Inserting a new element</td>
<td></td>
<td>list.add(index, “London”);</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>
Arrays from/to Arrays

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);
```

max and min in an ArrayList

```java
String[] array = {"red", "green", "blue"};
System.out.println(java.util.Collections.max(
    new ArrayList<String>(Arrays.asList(array))));
```

```java
String[] array = {"red", "green", "blue"};
System.out.println(java.util.Collections.min(
    new ArrayList<String>(Arrays.asList(array))));
```
shuffle an ArrayList

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
java.util.Collections.shuffle(list);
System.out.println(list);
```

### The MyStack Class

<table>
<thead>
<tr>
<th>MyStack</th>
<th>A list to store elements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-list: ArrayList</td>
<td>Returns true if this stack is empty.</td>
</tr>
<tr>
<td>+isEmpty(): boolean</td>
<td>Returns the number of elements in this stack.</td>
</tr>
<tr>
<td>+getSize(): int</td>
<td>Returns the top element in this stack.</td>
</tr>
<tr>
<td>+peek(): Object</td>
<td>Returns and removes the top element in this stack.</td>
</tr>
<tr>
<td>+pop(): Object</td>
<td>Adds a new element to the top of this stack.</td>
</tr>
<tr>
<td>+push(o: Object): void</td>
<td>Returns the position of the first element in the stack from the top that matches the specified element.</td>
</tr>
<tr>
<td>+search(o: Object): int</td>
<td></td>
</tr>
</tbody>
</table>

**Example:** A stack to hold objects

**Animation:** A stack of integers
The protected Visibility Modifier

- The protected modifier can be applied on data and methods in a class, but not to a class.
- A protected data field or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.

Visibility increases:
private, none (if no modifier is used), protected, public

Accessibility Summary

<table>
<thead>
<tr>
<th>Modifier 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</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</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>
Visibility Modifiers

### A Subclass Can’t Weaken the Accessibility

- A subclass may override a `protected` method in its superclass and change its visibility to `public`
- However, a subclass cannot weaken the accessibility of a method defined in the superclass

**Example:** If a method is defined as `public` in the superclass, it must be defined as `public` in the subclass
The final Modifier

- A final class cannot be extended
- A final method cannot be overridden by its subclasses
- A final variable is a constant
  - The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method
  - A final local variable is a constant inside a method

First Midterm Exam

- 60 minutes duration
- Will cover all lectures delivered before the exam date
- Will consist of MCQ’s and coding/programming tasks
- If you miss this exam for any valid reason, you will have to appear for an exam after the approval of your absence by the vice-dean of CCSE.