

Lecture 11 Polymorphism

Polymorphism, Dynamic Binding and Generic Programming

```
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 `m` takes a parameter of the `Object` type. You can invoke it with any object.

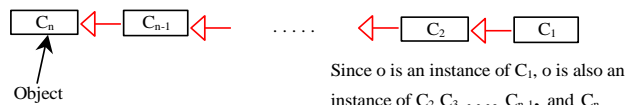
An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the method `m(Object x)` is executed, the argument `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 the `toString` method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Dynamic Binding

Dynamic binding works as follows:

- 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 a method `p`, the JVM searches the implementation for the method `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.



Method Matching vs. Binding

- Matching a method signature and binding a method implementation are two 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.
- The Java Virtual Machine dynamically binds the implementation of the method at runtime.

Generic Programming

```
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";
    }
}
```

- 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

You have already 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

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

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

```
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?

- Suppose you want to assign the object reference `o` to a variable of the `Student` type using the following statement:

```
Student b = o;
```
- A compilation error would occur.
- 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 you 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.
- The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```

Casting 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.

```
Apple x = (Apple)fruit;
```

```
Orange x = (Orange)fruit;
```

Why?

The instanceof Operator

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

```
Object myObject = new Circle();
... // 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());
    ...
}
```

TIP

- To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange.
- An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit.
- However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

The equals Method

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

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

- This method often should be overridden in the user defined classes.

```
For example, the equals method is overridden in the Circle class.
class.
public boolean equals(Object o) {
    if (o instanceof Circle) {
        return radius == ((Circle)o).radius;
    }
    else
        return false;
}
```

NOTE

- The `==` comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references.
- The `equals()` method 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 the `equals()` method, in that the `==` operator checks whether the two reference variables refer to the same object.

The ArrayList and Vector Classes

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.

java.util.ArrayList	
+ArrayList()	Creates an empty list.
+add(o: Object) : void	Appends a new element o at the end of this list.
+add(index: int, o: Object) : 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) : Object	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) : Object	Removes the element at the specified index.
+set(index: int, o: Object) : Object	Sets the element at the specified index.

Liang, Introduction to Java Programming, Eighth Edition, (c) 2011 Pearson Education, Inc. All rights reserved. 0132130807

13

The MyStack Classes

A stack to hold objects.


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

Liang, Introduction to Java Programming, Eighth Edition, (c) 2011 Pearson Education, Inc. All rights reserved. 0132130807

14

The protected Modifier

- The `protected` modifier can be applied on data and methods in a class.
- A protected data 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.
- private, default, protected, public

Visibility increases

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

Liang, Introduction to Java Programming, Eighth Edition, (c) 2011 Pearson Education, Inc. All rights reserved. 0132130807

15

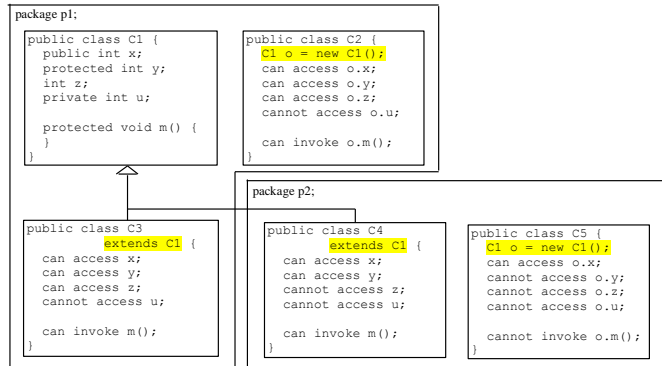
Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-

Liang, Introduction to Java Programming, Eighth Edition, (c) 2011 Pearson Education, Inc. All rights reserved. 0132130807

16

Visibility Modifiers



Liang, Introduction to Java Programming, Eighth Edition, (c) 2011 Pearson Education, Inc. All rights reserved. 0132130807

17

A Subclass Cannot 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.
- For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

Liang, Introduction to Java Programming, Eighth Edition, (c) 2011 Pearson Education, Inc. All rights reserved. 0132130807

18

NOTE

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.

The final Modifier

- The final class cannot be extended:

```
final class Math {
    ...
}
```
- The final variable is a constant:

```
final static double PI = 3.14159;
```
- The final method cannot be overridden by its subclasses.

Liang, Introduction to Java Programming, Eighth Edition, (c) 2011 Pearson Education, Inc. All rights reserved. 0132130807

19

Liang, Introduction to Java Programming, Eighth Edition, (c) 2011 Pearson Education, Inc. All rights reserved. 0132130807

20