CS112

Polymorphism
Lecture 06

الفصل الدراسي الثاني 1442 -Spring 2021

College of Computer Science and Engineering

Introduction

- 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*.

- Therefore, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

Polymorphism, Dynamic Binding and 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 {
  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 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 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.

C_n ⇐ C_{n-1} ⇐ ..... ⇐ C_2 ⇐ C_1

Object
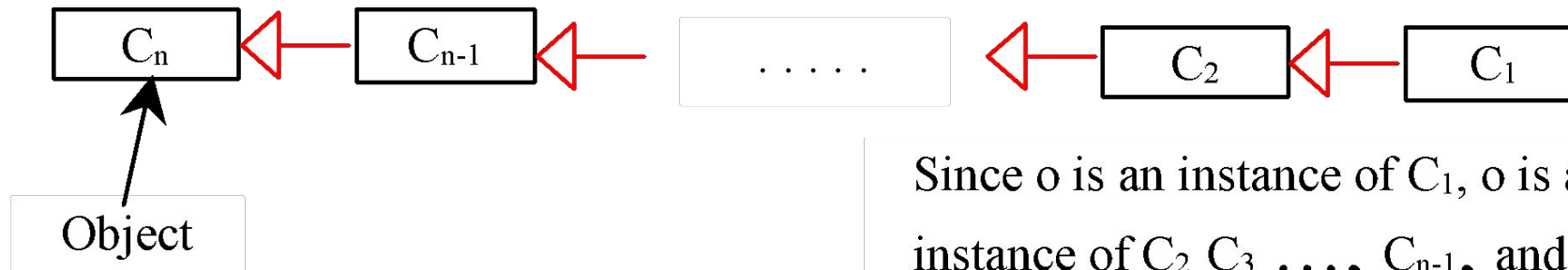
Since o is an instance of C_1, o is also an instance of C_2, C_3, ..., C_{n-1}, and C_n

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

```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";
  }
}
```

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.
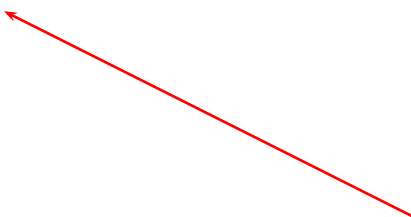
Casting Objects

- 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 compile 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;
```

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
/** 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.

Example: CastingDemo.java

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;
}
```

For example, the equals method is overridden in the Circle 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 <u>ArrayList</u> 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.

java.util.ArrayList\<E\>	
+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.

Generic Type

• 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 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<String>();

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

Differences and Similarities between Arrays and ArrayList

Operation	Array	ArrayList
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

Array Lists from/to Arrays

- Creating an ArrayList from an array of objects:

String[] array = {**"red"**, **"green", "blue"**};

ArrayList<String> list = **new** ArrayList<>(Arrays.asList(array));

- Creating an array of objects from an ArrayList:

String[] array1 = **new** String[list.size()];

list.toArray(array1);

max and min in an Array List

String[] array = {**"red"**, **"green", "blue"**};

System.out.pritnln(java.util.Collections.max(
   new ArrayList<String>(Arrays.asList(array)));

String[] array = {**"red"**, **"green", "blue"**};

System.out.pritnln(java.util.Collections.min(
   new ArrayList<String>(Arrays.asList(array)));

Shuffling an Array List

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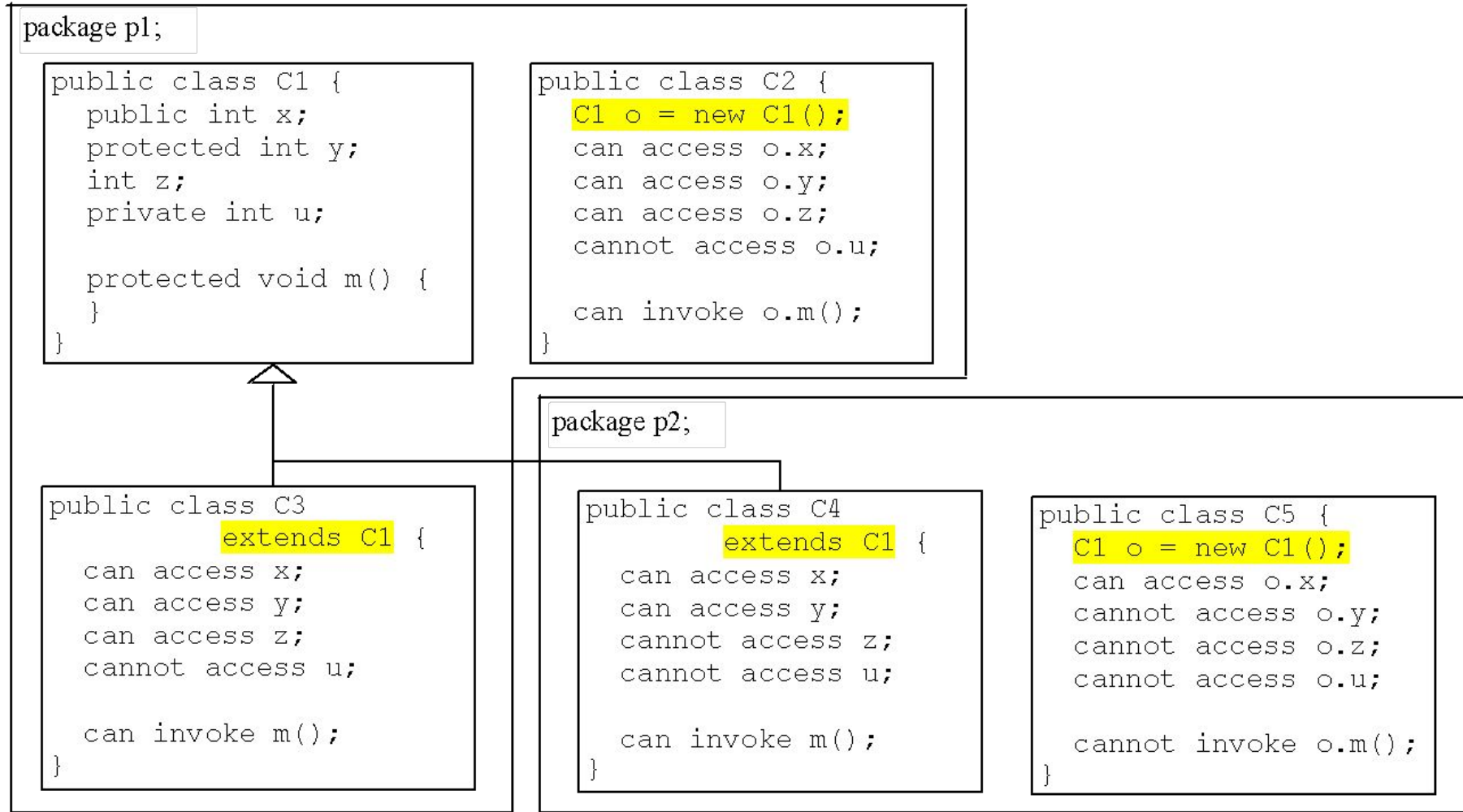
private, none (if no modifier is used), protected, public

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	✓	–	–	–

Visibility Modifiers

```
package p1;

public class C1 {                    public class C2 {
    public int x;                        C1 o = new C1();
    protected int y;                     can access o.x;
    int z;                               can access o.y;
    private int u;                       can access o.z;
                                         cannot access o.u;
    protected void m() {
    }                                    can invoke o.m();
}                                    }
```

```
public class C3            public class C4            public class C5 {
        extends C1 {               extends C1 {           C1 o = new C1();
    can access x;             can access x;               can access o.x;
    can access y;             can access y;               cannot access o.y;
    can access z;             cannot access z;            cannot access o.z;
    cannot access u;          cannot access u;            cannot access o.u;

    can invoke m();           can invoke m();             cannot invoke o.m();
}                         }                          }
```

package p2;

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.

NOTE

The modifiers are used on classes and class members (data and methods), except that the <u>final</u> 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.