Northeastern University

Lecture 13: Inheritance and Polymorphism - 3

Prof. Chen-Hsiang (Jones) Yu, Ph.D.

College of Engineering

Materials are edited by Prof. Jones Yu from

Liang, Y. Daniel. Introduction to Java Programming, Comprehensive Version, 12th edition, Pearson, 2019.

Outline

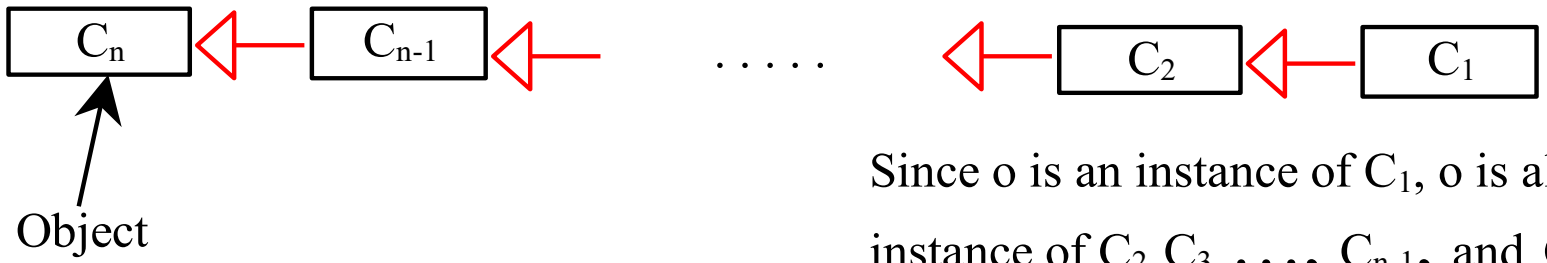
- Inheritance
- Polymorphism

Outline

- Inheritance
- Polymorphism

Polymorphism (cont.)

Dynamic Binding



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

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.

Casting Objects

- *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy.
- 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);
```

Why Casting Is Necessary?

- Suppose you want to assign the object reference `o` (`Object 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?

Why Casting Is Necessary? (cont.)

- This is because a `Student` object is always an instance of `Object`, but an `Object` 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.

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
```

- Explicit casting must be used when casting an object from a superclass to a subclass.

Exercise

- Please create a new Java project and type following codes.
- What is the output of following program? Why?

```java
public class CSYE6200 {
    public static void main(String[] args){
        new A();
        new B();
    }
}

class A{
    int i = 7;

    public A(){
        setI(20);
        System.out.println("i from A is " + i);
    }

    public void setI(int i){
        System.out.println("A's setI is called.");
        this.i = 2 * i;
    }
}

class B extends A{
    public B(){
        System.out.println("i from B is "+ i);
    }

    public void setI(int i){
        System.out.println("B's setI is called.");
        this.i = 3 * i;
    }
}
```

Answer

```
A's setI is called.
i from A is 40
B's setI is called.
i from A is 60
i from B is 60
```

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());
  ...
}
```

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:

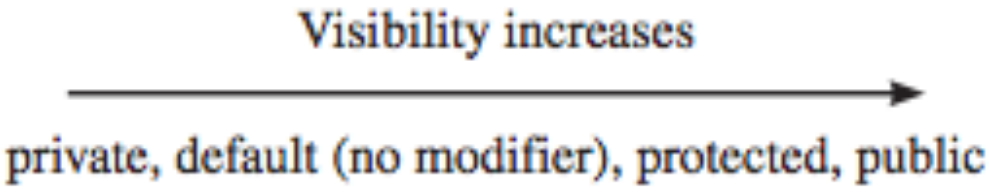
```
public boolean equals(Object obj) {
   return this == obj;
}
```

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;
}
```

The `protected` Modifier

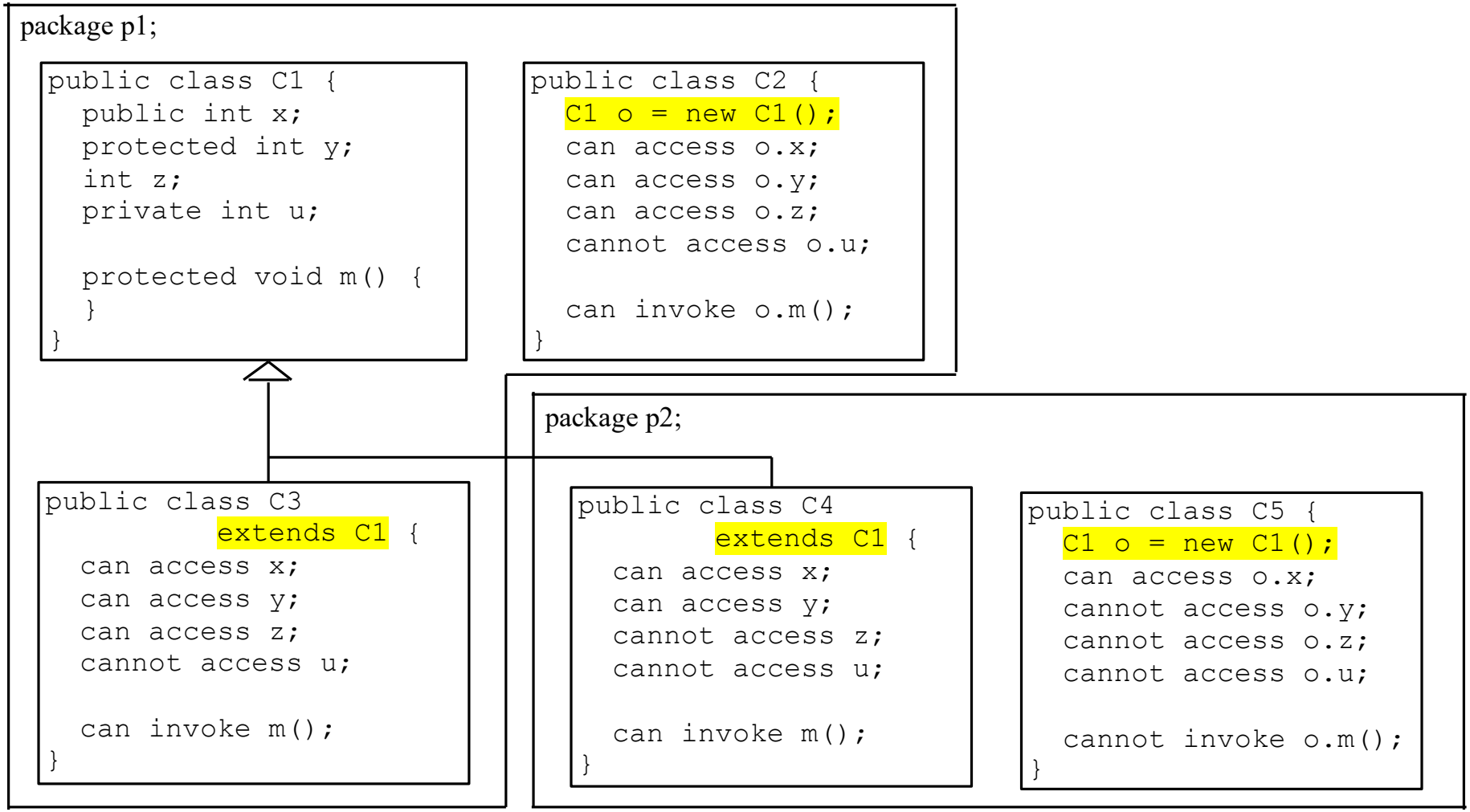
- The `protected` modifier can be applied on data and methods in a class.
- A `protected` data or a `protected` method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- `private`, `default`, `protected`, `public`



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



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 (i.e., more restrictive) 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.

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.

ArrayList

The `ArrayList` Class

- You can create an array to store objects. But the array's size is fixed once the array is created.
  - `Student[ ] myStudents = new Student[20];`
- 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 first element o from this list. Returns true if an element is removed.
+size(): int	Returns the number of elements in this list.
+remove(index: int): boolean	Removes the element at the specified index. Returns true if an element is removed.
+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>();
```

Arrays vs. 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);
```

Exercise

- Please ask the user to enter a sequence of integer numbers and display the distinct numbers in the sequence.
- The user can end the input by adding a non-integer character, such as “e”.

```
Please enter numbers: 5 4 3 3 3 3 3 3 4 2 e
Distinct numbers: 5 4 3 2
```

Answer

```
public class ClassExample{

     public static void main(String[] args) {

          Scanner input = new Scanner(System.in);
          ArrayList<Integer> data = new ArrayList<>();


          System.out.print("Please enter numbers: ");

          while(input.hasNextInt()) {
               Integer number = input.nextInt();

               if(!data.contains(number)) {
                    data.add(number);
               }
          }

          System.out.print("Distinct numbers: ");
          for(Integer i: data) {
               System.out.print(i + " ");
          }

          input.close();

     }

}
```

Exercise

- Continued from the previous exercise.
- Please sort the distinct numbers in an ascending order.

```
Please enter numbers: 5 4 3 3 3 3 3 3 4 2 e
Distinct numbers: 2 3 4 5
```

Answer

```
public class ClassExample{

     public static void main(String[] args) {

          Scanner input = new Scanner(System.in);
          ArrayList<Integer> data = new ArrayList<>();


          System.out.print("Please enter numbers: ");

          while(input.hasNextInt()) {
               Integer number = input.nextInt();

               if(!data.contains(number)) {
                    data.add(number);
               }
          }

          Collections.sort(data);

          System.out.print("Distinct numbers: ");
          for(Integer i: data) {
               System.out.print(i + " ");
          }

          input.close();

     }

}
```