CEN 419
Introduction to Java Programming

Dr. H. Esin ÜNAL

FALL 2021

Slides are modified from original slides of Y. Daniel Liang

Polymorphism

- The **occurrence of different forms** among the members of a population or colony, or in the life cycle of an individual organism.
- In programming languages and type theory, **polymorphism** (from Greek, "many, much" "form, shape") is the provision of a single interface to entities of different types.
- A **polymorphic type** is one whose operations can also be applied to values of some other type, or types.

Subtype-Supertype

- 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

- A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa.
- Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.

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

Polymorphism Demo

```
public class PolymorphismDemo {
  /** Main method */
  public static void main(String[] args) {
    // Display circle and rectangle properties
    displayObject(new Circle(1, "red", false));
    displayObject(new Rectangle(1, 1, "black", true));
  }

  /** Display geometric object properties */
  public static void displayObject(GeometricObject object) {
    System.out.println("Created on " + object.getDateCreated()
         + ". Color is " + object.getColor());
  }
}
```

Dynamic Binding

- A method can be defined in a superclass and overridden in its subclass. For example, the **toString()** method is defined in the **Object** class and overridden in the **GeometricObject** class.
- Consider the following code:

```
Object o = new GeometricObject();
System.out.println(o.toString());
```

Which **toString()** method is invoked by **o**?

To answer this question, we first introduce two terms: ***declared type*** and ***actual type***.

Dynamic Binding

- A variable must be declared by a type. The type that declares a variable is called the variable's ***declared type***. Here **o**'s declared type is **Object** (`Object o = new GeometricObject();`).
- A variable of a reference type can hold a **null** value or a reference to an instance of the declared type. The instance may be created using the constructor of the declared type or its subtype.
- The **actual type** of the variable is the actual class for the object referenced by the variable. Here o's actual type is **GeometricObject**, because **o** references an object created using **new GeometricObject()**.

*Which toString() method is invoked by o is determined by **o's actual type**. This is known as **dynamic binding**.*

Dynamic Binding

- *A method can be implemented in several classes along the inheritance chain. The JVM decides which method is invoked at runtime.*
  - ✓ 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.

```
public class DynamicBindingDemo {
  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***.

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 example, 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. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

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

`instanceof` Operator

- For the casting to be successful, you must make sure that the object to be cast is an instance of the subclass.
- If the superclass object is not an instance of the subclass, a runtime ***ClassCastException*** occurs.
- This can be ensured by using the **`instanceof`** operator:

```
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());
...
}
```

Example: Demonstrating Polymorphism and Casting

This example creates two geometric objects: a circle, and a rectangle, invokes the displayGeometricObject method 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.

```java
public class CastingDemo {
  /** Main method */
  public static void main(String[] args) {
    // Create and initialize two objects
    Object object1 = new Circle (1);
    Object object2 = new Rectangle (1, 1);

    // Display circle and rectangle
    displayObject(object1);
    displayObject(object2);
  }

  /** A method for displaying an object */
  public static void displayObject(Object object) {
    if (object instanceof Circle) {
       System.out.println("The circle area is " + ((Circle)object).getArea());
       System.out.println("The circle diameter is " + ((Circle)object).getDiameter());
    }
    else if (object instanceof Rectangle) {
       System.out.println("The rectangle area is " +((Rectangle)object).getArea());
    }
  }
}
```

CAUTION

- The object member access operator (.) precedes the casting operator. Use parentheses to ensure that casting is done before the . operator, as in

```
((Circle)object).getArea();
```

Use Parantheses!!!

NOTE

- Casting a primitive type value is different from casting an object reference. Casting a primitive type value returns a new value. For example:

```
int age = 45;
byte newAge = (byte)age; // A new value is assigned to newAge
```

- However, casting an object reference does not create a new object. For example:

```
Object o = new Circle();
Circle c = (Circle)o; // No new object is created
```

Now reference variables **o** and **c** point to the same object.

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 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**.

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) : E`	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.

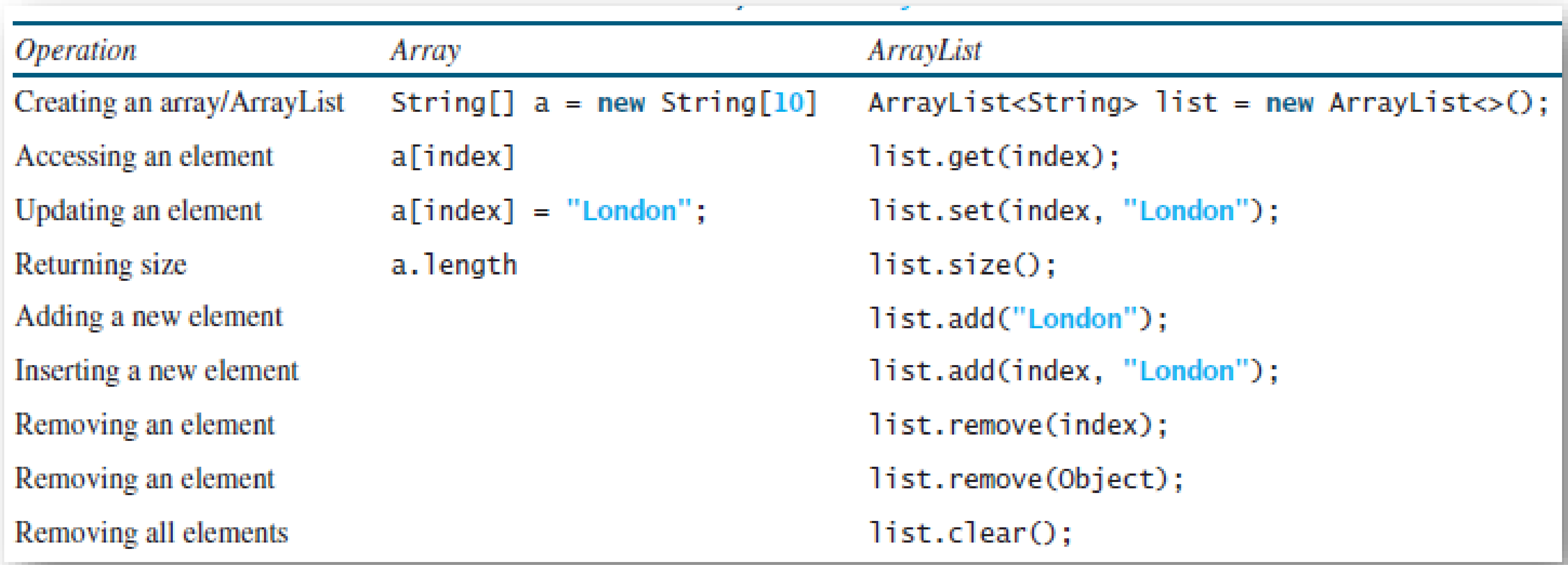
```
ArrayList<String> cities = new ArrayList<String>();

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

Test ArrayList

https://liveexample.pearsoncmg.com/liang/intro10e/html/TestArrayList.html

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();`

- ✓ You can sort an array using the **java.util.Arrays.sort(array)** method.
- ✓ To sort an array list, use the **java.util.Collections.sort(arraylist)** method.

Example: ArrayList

- Suppose you want to create an **ArrayList** for storing integers. Can you use the following code to create a list?

```
ArrayList<int> list = new ArrayList<>();
```

- ***This will not work*** because the elements stored in an ArrayList must be of an object type (put `Integer` instead of `int`).

```
ArrayList<Integer> list = new ArrayList<>();
```

- So, lets write a program that prompts the user to enter a sequence of numbers and displays the distinct numbers in the sequence.
- Assume that the input ends with **0** and **0** is not counted as a number in the sequence.

Distinct Numbers

https://liveexample.pearsoncmg.com/liang/intro10e/html/DistinctNumbers.html

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);
```

Case Study: A Custom Stack Class

Objective: A stack class to hold objects.

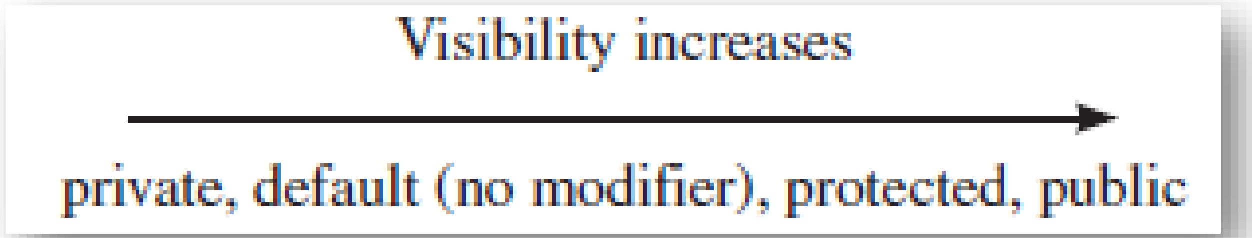
MyStack	
-list: ArrayList	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: 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.

MyStack

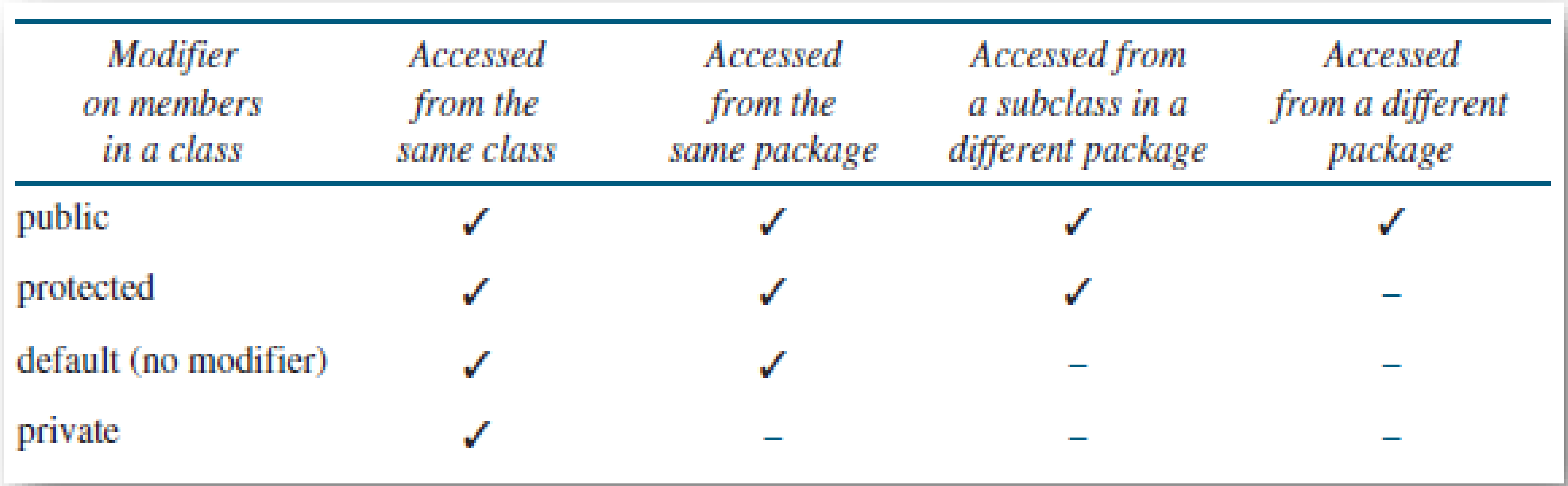
https://liveexample.pearsoncmg.com/liang/intro10e/html/MyStack.html

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

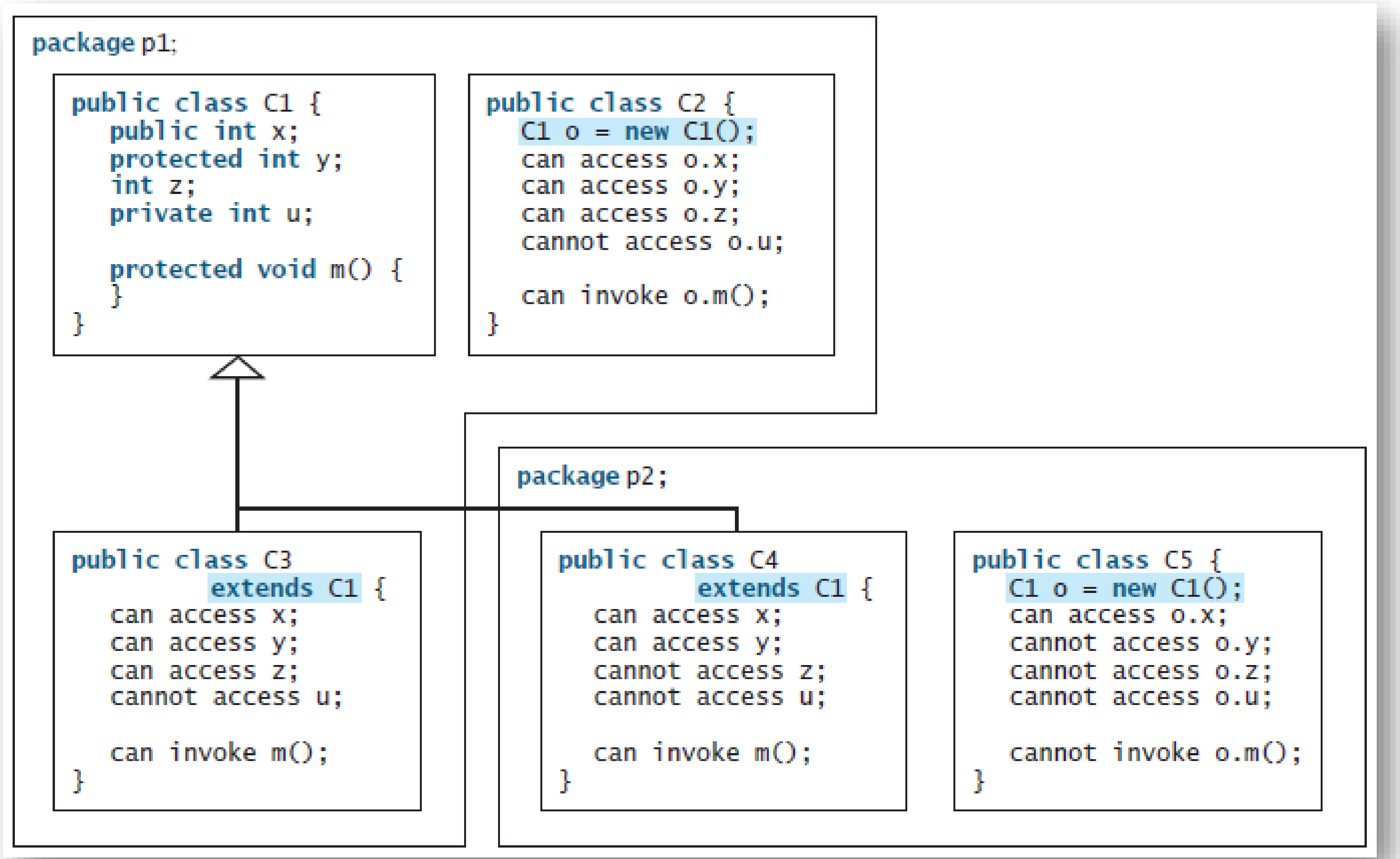


Accessibility Summary



Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers



package p1;

```
public class C1 {
   public int x;
   protected int y;
   int z;
   private int u;

   protected void m() {
   }
}
```

```
public class C2 {
   C1 o = new C1();
   can access o.x;
   can access o.y;
   can access o.z;
   cannot access o.u;

   can invoke o.m();
}
```

```
public class C3
         extends C1 {
   can access x;
   can access y;
   can access z;
   cannot access u;

   can invoke m();
}
```

package p2;

```
public class C4
         extends C1 {
   can access x;
   can access y;
   cannot access z;
   cannot access u;

   can invoke m();
}
```

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

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

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.

The `final` Modifier

- You may occasionally want to prevent classes from being extended.
- In such cases, use the **final** modifier to indicate that a class is final and **cannot be a parent class**.
- The **Math, String**, **StringBuilder**, and **StringBuffer** classes are final classes.

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.