Inheritance and Polymorphism

Review: Classes

- User-defined data types
  - Defined using the “**class**” key word
  - Each class has associated
    - Data members
    - Methods that operate on the data
- New instances of the class are declared using the “**new**” keyword

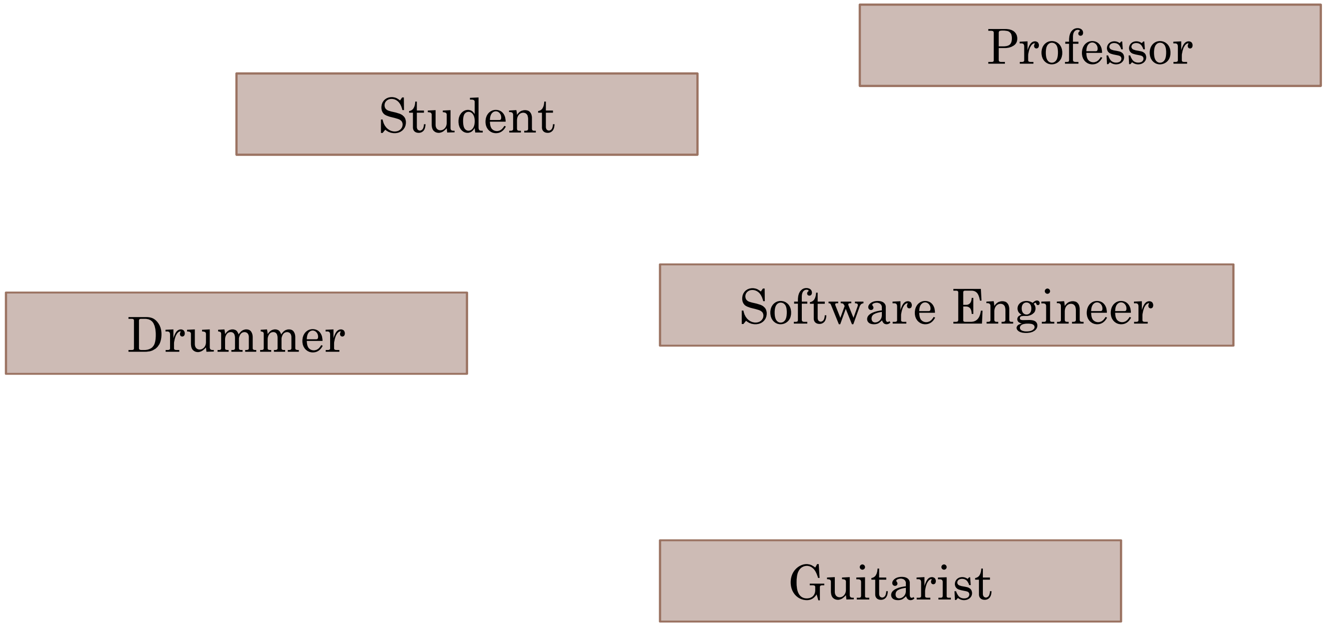
Review: Classes

- Static members/methods have only one copy regardless of how many instances are created so a static variable
- A **static** method can be invoked without the need for creating an instance of a class.

Inheritance

Introduction

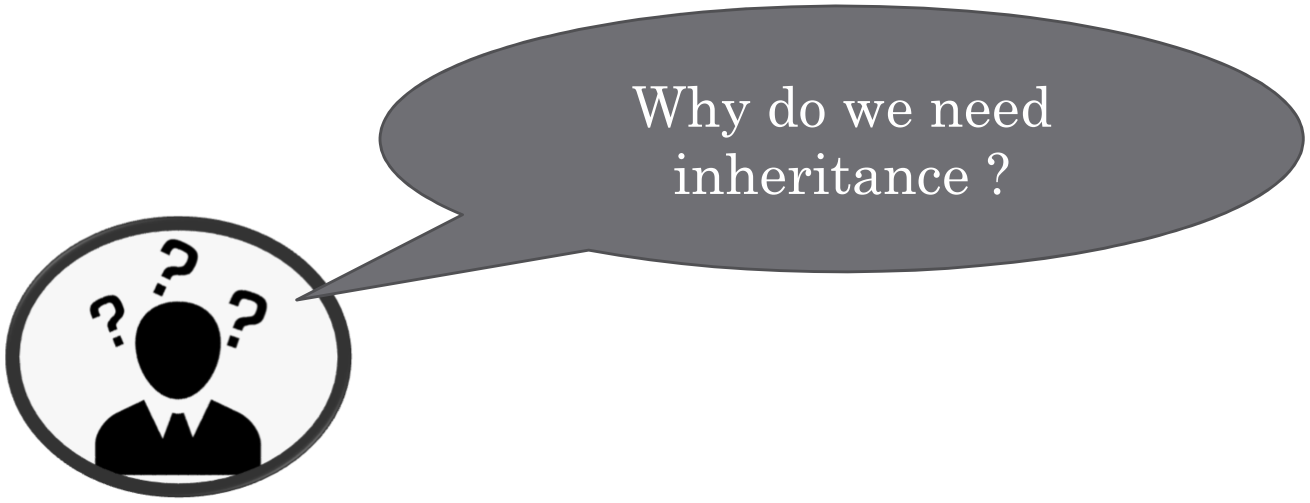
- Objects are unique but they often share similar behavior.



Introduction

- Object-oriented programming allows you to define new classes from existing classes. This is called **inheritance**.
- Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).

Why not just copy-and-paste?



Example: Shared Functionality

```java
public class Student {
   String name;
   char gender;
   Date birthday;
   Vector<Grade> grades;

   double getGPA() {
      …
   }

   int getAge(Date today) {
      …
   }
}
```

```java
public class Professor {
   String name;
   char gender;
   Date birthday;
   Vector<Paper> papers;

   int getCiteCount() {
      …
   }

   int getAge(Date today) {
      …
   }
}
```

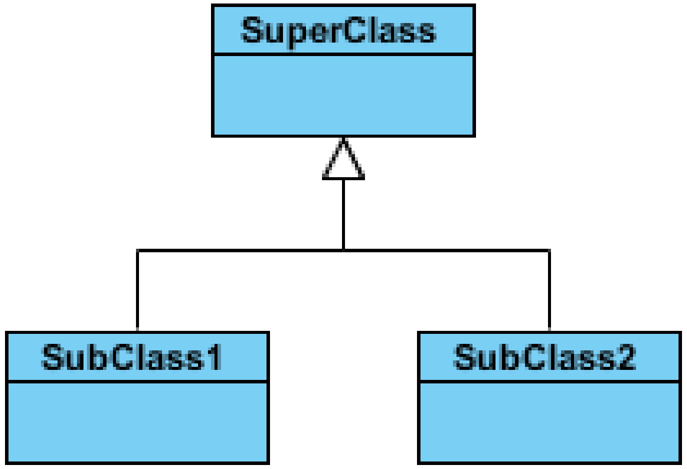
```
public class Person {
  String name;
  char gender;
  Date birthday;

  String getName() {
    return this.name;
  }

  int getAge(Date today) {
    …
  }
}
```

```
public class Student
    extends Person {

  Vector<Grade> grades;

  double getGPA() {
    …
  }
}
```

```
public class Professor
    extends Person {

  Vector<Paper> papers;

  int getCiteCount() {
    …
  }
}
```

Inheritance

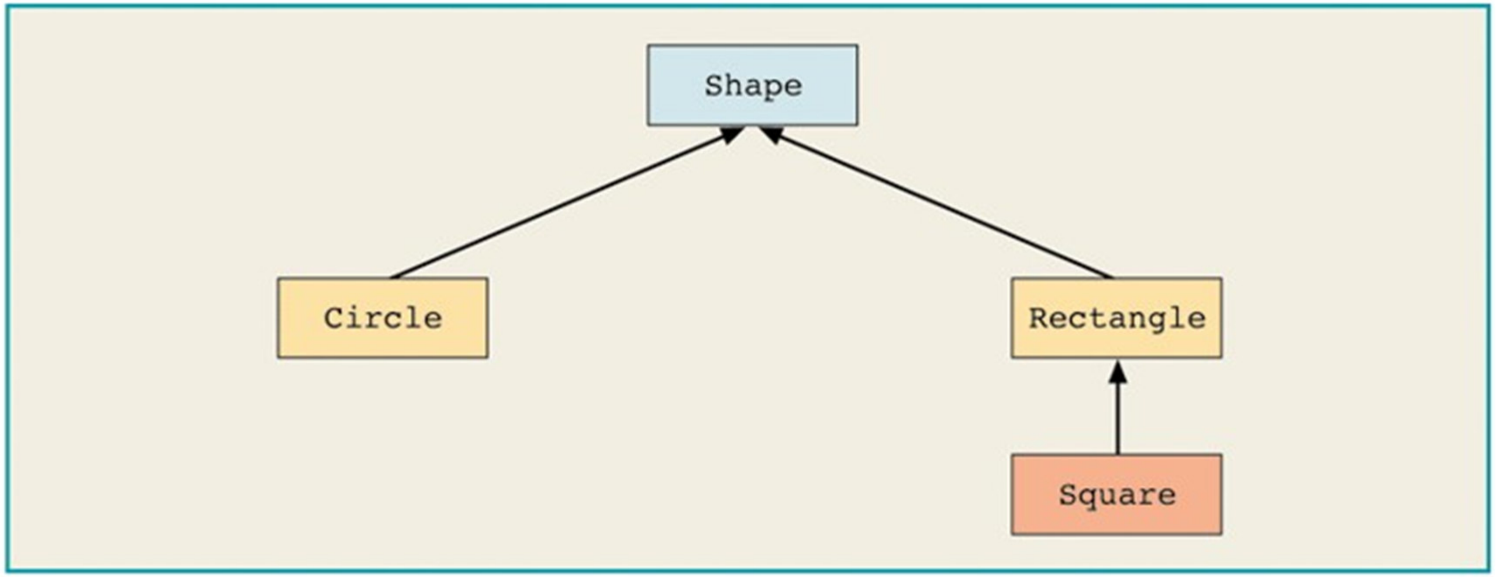
- When you design with inheritance, you put common code in a class and then tell other more specific classes that the common (more abstract) class is their superclass (parent class). When one class inherits from another, the subclass (Child class) inherits from the superclass.



Advantage of Inheritance

- The biggest advantage of Inheritance is that the code that is already present in base class need not be rewritten in the child class. (reuse code)
- Inheritance can also make application code more intuitive/expressive.

Syntax: Inheritance in Java



```
modifier(s) class ClassName extends ExistingClassName
                                              {
   \\ memberList
   \\ Methods
}
```

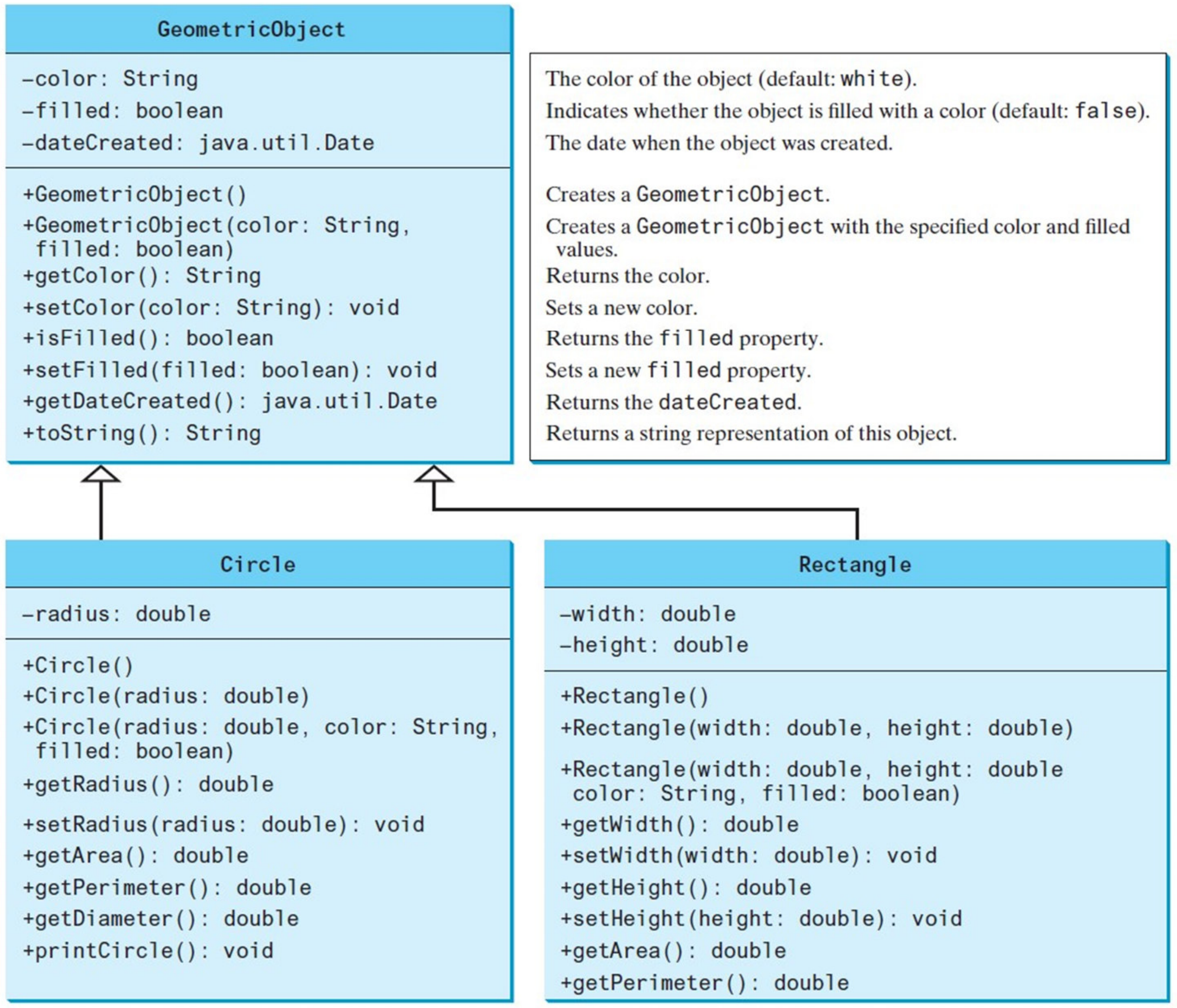
Superclass and Subclass

- We can think of many examples in real life of how a (super) class can be extended to a set of (sub) classes
- For example a Polygon class can be extended to be a Quadrilateral
- We can think of these classes as following an IS-A relationship
  - A Quadrilateral IS-A Polygon
  - Orange IS-A fruit
  - A dog IS-An animal

Superclass and Subclass

Superclass	**Subclass**
Shape	Triangle, Circle, Rectangle
Student	Undergraduate, Postgraduate
Vehicle	Car, Truck, Bus
Filter	Low-pass, Band-pass, High-pass

Superclass and Subclass


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
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.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Superclass and Subclass

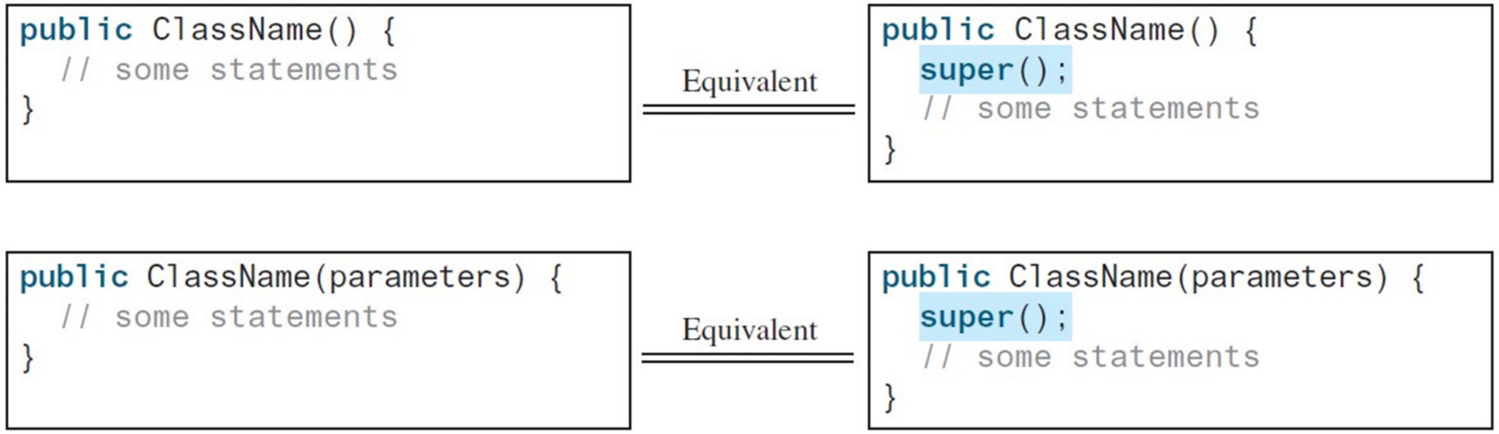
- Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass.
- Not all is-a relationships should be modeled using inheritance.
- Inheritance is used to model the is-a relationship. Do not blindly extend a class just for the sake of reusing methods.
- Java, however, does not allow multiple inheritance.

Are superclass's constructor inherited?

- No. They are not inherited. A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited by a subclass.
- They can only be invoked from the constructors of the subclasses using the keyword `super`.

Superclass' constructor always invoke

- A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts `super()` as the first statement in the constructor. For example:



```
public ClassName() {
  // some statements
}
```

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(parameters) {
  // some statements
}
```

```
public ClassName(parameters) {
  super();
  // some statements
}
```

The `super` Keyword

- The keyword `super` refers to the superclass and can be used to invoke the superclass's methods and constructors.
- It can be used in two ways:
  - To call a superclass' method, use

    `super.methodName(...)`
  - To call a superclass' constructor, use

    `super()` or `super(parameters)`

    from the child class' constructor

CAUTION

You must use the keyword `super` to call the superclass constructor, and the call must be the first statement in the constructor. Invoking a superclass constructor's name in a subclass causes a syntax error.

Constructor chaining

```
public class Faculty extends Employee {
    public static void main(String[] args) {
    new Faculty();
    }
    // Faculty’s constructor
    public Faculty() {
    System.out.println("(4) Performs Faculty's tasks");
    }
}
```

```
class Employee extends Person {

    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Performs Employee's tasks ");
    }

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

```
public class Person {
    public Person() {
        System.out.println("(1) Performs Person's tasks");
    }
}
```

CAUTION

If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors. Consider the following code:

```
public class Apple extends Fruit {
   //some code here
}

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

Constructor Chain

- If possible, you should provide a no-arg constructor for every class to make the class easy to extend and to avoid errors.

Defining a subclass

- A subclass inherits from a superclass. You can also:
  - Add new properties
  - Add new methods
  - Override the methods of the superclass

Calling Superclass Methods

- You could rewrite the `printCircle()` method in the `Circle` class as follows:

```
public void printCircle() {
    System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

Overriding Methods

- A subclass inherits methods from a superclass. Sometimes, it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*

```
public class Circle extends GeometricObject {
    // other methods are omitted
/** Override the toString method defined in GeometricObject */



    public String toString(){
        return super.toString() + "\nradius is " + radius;
    }
}
```

NOTE

- The overriding method must have the same signature as the overridden method and same or compatible return type.
- 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.

NOTE

- 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.

Summary

	inheritanced	**override**
Static	Yes	No (hidden)
private	No	No

Overriding vs. Overloading

- Overloading means to define multiple methods with the same name but different signatures.
- Overriding means to provide a new implementation for a method in the subclass.

Overriding vs. Overloading

```java
public class TestOverriding {
    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
    public void p(double i) {
        System.out.println(i);
    }
}
```

```java
public class TestOverloading {
    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 overloads the method in B
    public void p(int i) {
        System.out.println(i);
}
}
```

Overriding vs. Overloading

- *Overridden* methods are in different classes related by inheritance; *overloaded* methods can be either in the same class, or in different classes related by inheritance.
- *Overridden* methods have the same signature; *overloaded* methods have the same name but different parameter lists.

Overriding (cont'd)

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

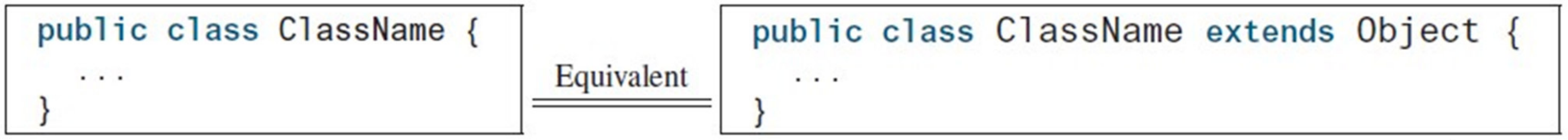
```
public class B {
    public void p(double i) {    //private?
        System.out.println(i*2);
    }

    public void d(double i) {
        p(i);
    }

}
```

What happens if method p (of class B) is changed to *Private* ?

```
public class A extends B {
    public void p(double i) {
        System.out.println(i);
    }
}
```

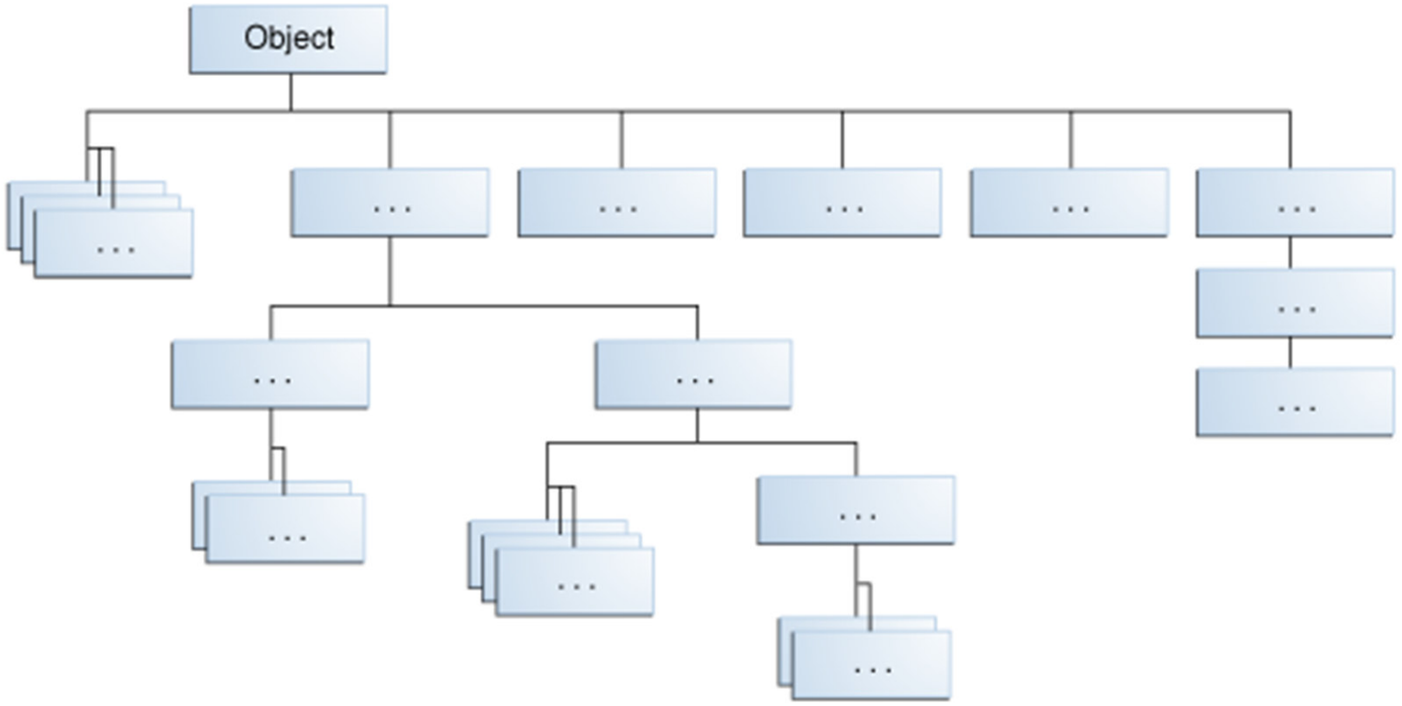
The Object Class and Its Method

- 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` by default.



```
public class ClassName {
  ...
}
```

```
public class ClassName extends Object {
  ...
}
```


Object
...

The `toString()` Method in `Object`

- Invoking `toString()` on an object returns a string that describes the object. By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal.

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

The output for this code displays something like Loan@15037e5.

Polymorphism

Polymorphism

- ***Polymorphism*** is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism literally means *many forms*.
- It occurs when we have many classes that are related to each other by inheritance.
- Polymorphism allows a reference to a superclass to be used instead of a reference to its subclass

Polymorphism

- Essentially we are able to get many different types of object behavior from a single reference type
  - This enables us to write easily extensible applications

```java
public class PolymorphiMain {

    public static void main(String[] args) {
         Vehicle v1 = new Vehicle();
         Vehicle v2 = new Car();  // polymorphism
         Car c1 = new Car();
       //Car c2 = new Vehicle(); superclass can assigns to subclass

        v1.printName();
        v2.printName();

        v1.description();
        v2.description();
}
```

```java
public class Vehicle {
    private String name = "vehicle";
    public void printName() {
        System.out.println(this.name);
    }
    public void description() {
        System.out.println("I am a vehicle");
    }
}
```

```java
public class Car extends Vehicle {
    private String name = "Car";
    public void printCar() {
        System.out.println(this.name);
    }
    @Override
    public void description() {
        System.out.println("I am a Car");
    }
}
```

Dynamic Binding

```
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());
    }
}

public class GraduateStudent extends Student {

}

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

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

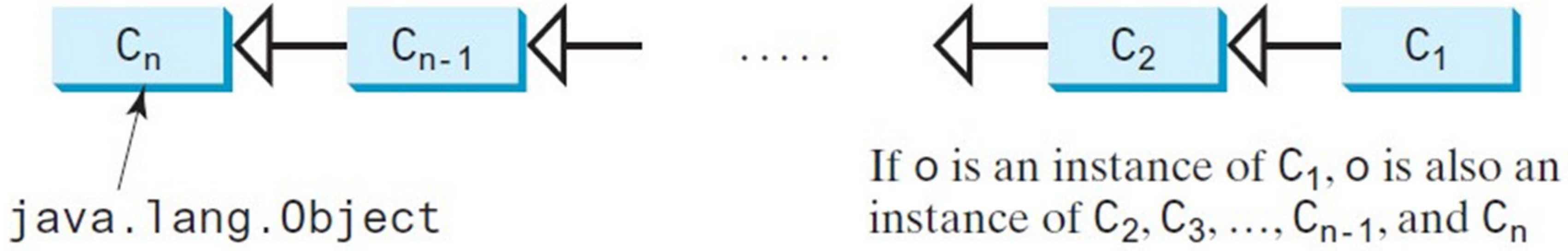
When the method `m(Object x)` is executed, the argument x's `toString` method is invoked. x may be an instance of `GraduateStudent`, `Student`, `Person`, or `Object`. Classes `GraduateStudent`, `Student`, `Person`, or `Object` have their own implementation of the `toString` method. Which implementation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

Method Matching vs. Binding

- Matching a method signature and binding a method implementation are two separate issues.
- The *declared type* of the reference variable decides which method to match at compile time. The compiler finds a matching method according to the parameter type, number of parameters, and order of the parameters at compile time.
- A method may be implemented in several classes along the inheritance chain. The JVM dynamically binds the implementation of the method at runtime, decided by the *actual type* of the variable.

Dynamic Binding

Dynamic binding works as follows: Suppose that 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**, as shown in Figure 11.2. 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 for the implementation of 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.



C_n
C_{n-1}
.....
C_2
C_1
java.lang.Object
If 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

- Whether a method can be invoked literally on an instance is a *syntax issue*—matching the method invocation to some method signature.
- When a method can be invoked, which implementation to select is a *sematic issue*– binding a particular implementation according to the instance that reference actually points to at run time.

Dynamic Binding

```
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());
    }
}

public class GraduateStudent extends Student {

}

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

public 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.

Casting Objects

- You have already used the casting operator to convert variables of one primitive type to another.

```
int intVal = (int)5.3;
```

- 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);
```

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, but **Student b = o** doesn't? The reason is that 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 clever enough to know it. To tell the compiler **o** is a **Student** object, use *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;

Ograngе x = (Orange)fruit;
```

Casting

- Casting a reference does not change anything, especially not the object pointed to by the reference. Roughly speaking, it changes the class *label* of the object – as if compiler invokes the methods of the object according to the class label.

```
Shape s = new Circle(); // s is created as a circle
                           but now is labeled as shape

s.setRadius(1.0);       // this is syntactically wrong,
                          s is a now reference of shape

Circle c = (Circle)s;   //downcasting must be explicit.
                          But it can fail if s does not
                          point to a Circle at runtime

c.setRadius(1.0);       // correct
```

```java
public class TestDownCast {
   public static void main(String[] args) {
      Dog d = new Dog();
      ((Animal)d).callme(); //Compiles,this invokes callme from Dog class.
      ((Animal)d).callme2(); // Compilation error

      Animal d2 = new Dog();
      d2.callme(); //compiles, this invokes callme from Dog class
      d2.callme2(); // compilation error
      ((Dog)d2).callme2(); //Complies, invokes callme2 from Dog
     }
}

public class Animal {
    public void callme() {
        System.out.println("In callme of Animal");
    }
}

public class Dog extends Animal {
    @Override
    public void callme() {
        System.out.println("In callme Dog");
    }
    public void callme2() {
        System.out.println("In callme2 Dog");
    }
}
```

Note how the overridden Methods are invoked after casting

The `instanceof` Operator

- The java `instanceof` operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

```
class Simple1{
    public static void main(String args[]){
    Simple1 s=new Simple1();
    System.out.println(s instanceof Simple1);//true
    }
}
```

The `equals` Method

- Like the `toString()` method, the `equals(Object)` method is another useful method defined in the Object class. The default implementation of the `equals` method in the `Object` class is

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

- You can override the **equals** method in the **Circle** class to compare whether two circles are equal based on their radius as follows:

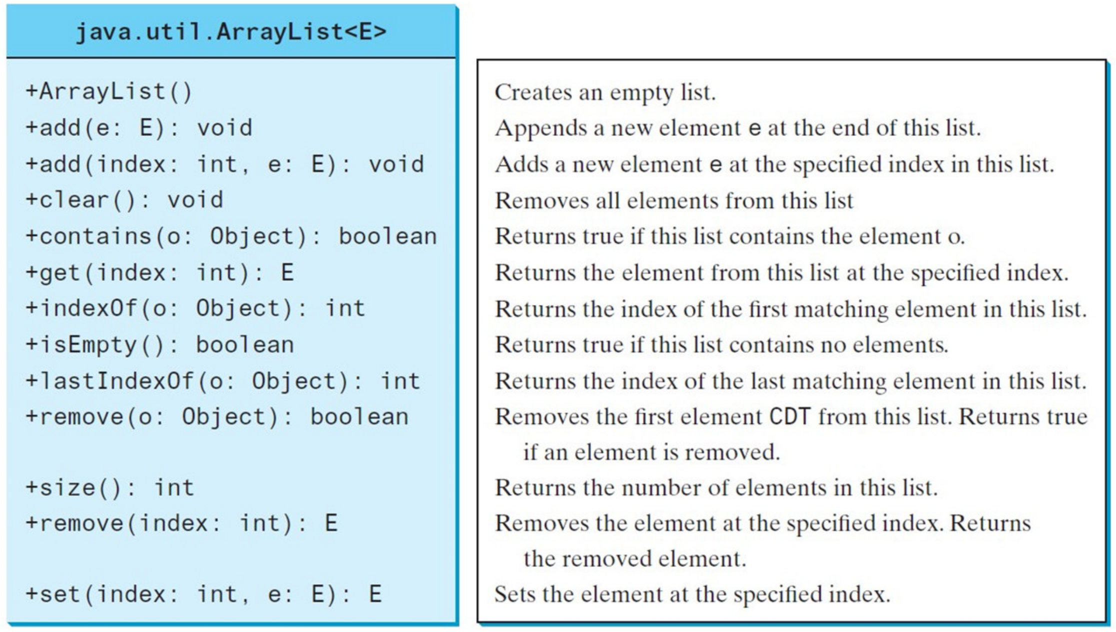
```
@Override
public boolean equals(Object o) {
    if (o instanceof Circle)
        return radius == ((Circle)o).radius;
    else
        return false;
}
```

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 the method is overridden 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` Class

- You can create an array to store objects. However, once the array is created, its size is fixed. Java provides the `ArrayList` class, which can be used to store an unlimited number of objects.



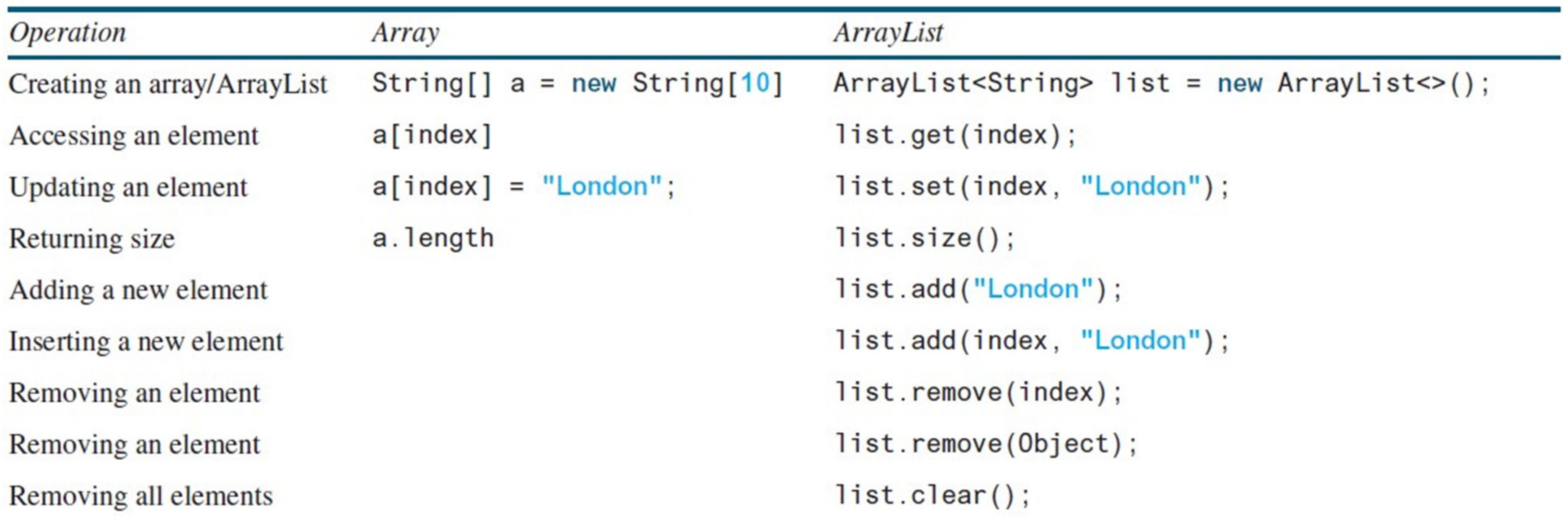
java.util.ArrayList<E>	
`+ArrayList()`	Creates an empty list.
`+add(e: E): void`	Appends a new element `e` at the end of this list.
`+add(index: int, e: E): void`	Adds a new element `e` at the specified index in this list.
`+clear(): void`	Removes all elements from this list
`+contains(o: Object): boolean`	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 CDT from this list. Returns true if an element is removed.
`+size(): int`	Returns the number of elements in this list.
`+remove(index: int): E`	Removes the element at the specified index. Returns the removed element.
`+set(index: int, e: E): E`	Sets the element at the specified index.

Generic Types

- 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<java.util.Date> dates
    = new ArrayList<java.util.Date>();
```

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();`

The `Protected` Modifier

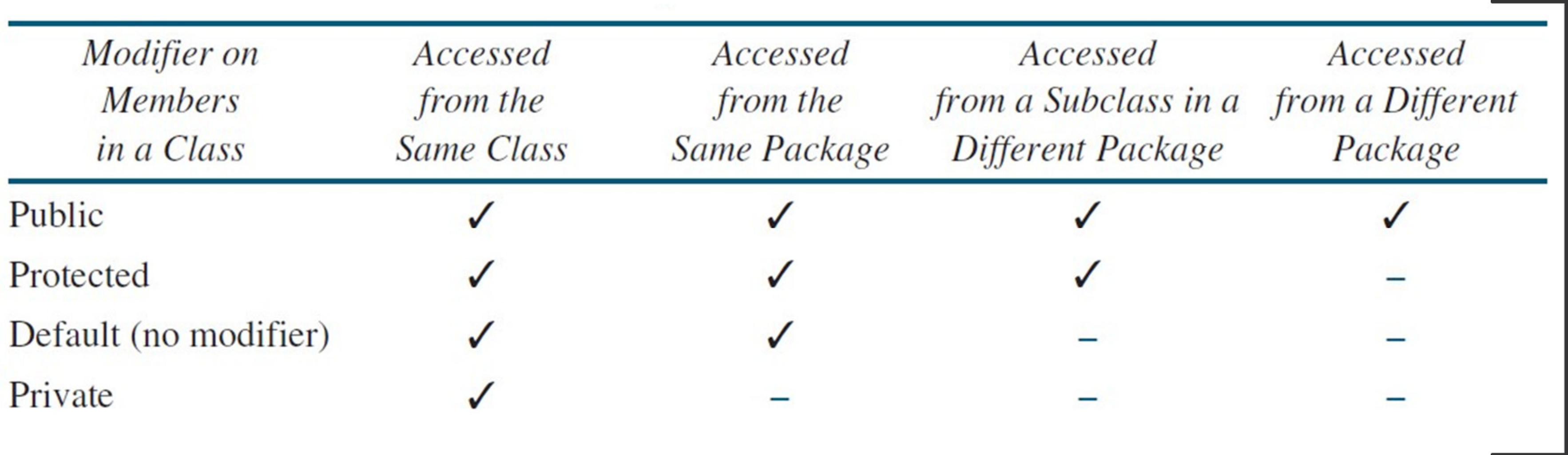
- *A protected modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.*
- *Private, default, protected, public*

Visibility increases

$\longrightarrow$

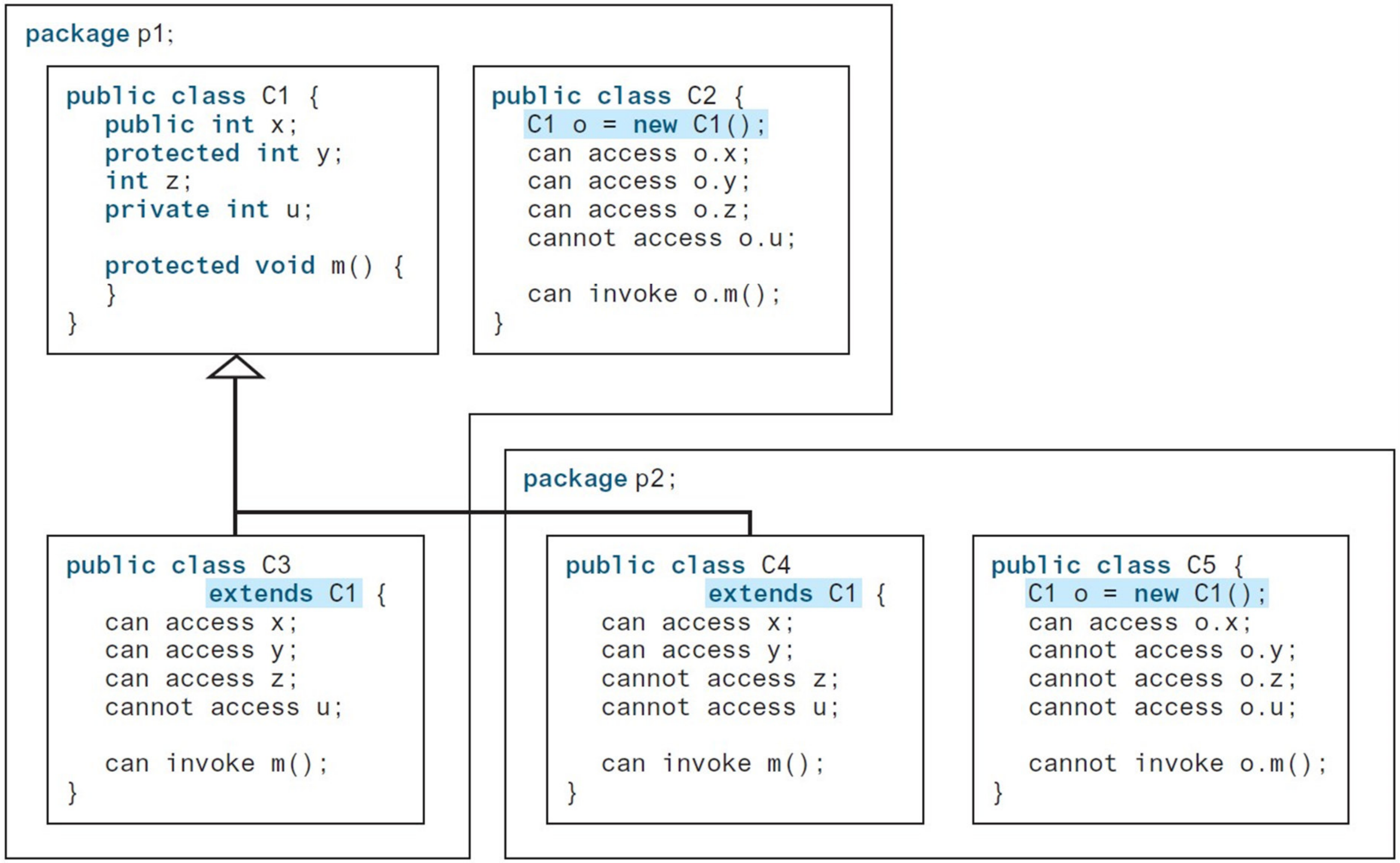
Private, default(no modifier), protected, public

Accessibility



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 defined 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 **public**, **protected**, **private**, **static**, **abstract**, and **final** 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 can't be extended

```
final class Math{
    ...
}
```

- The final variable is a constant

```
final static double PI = 3.14159;
```

- The final method can't be overridden by its subclasses.