Inheritance and Polymorphism

CSE 114, Computer Science 1

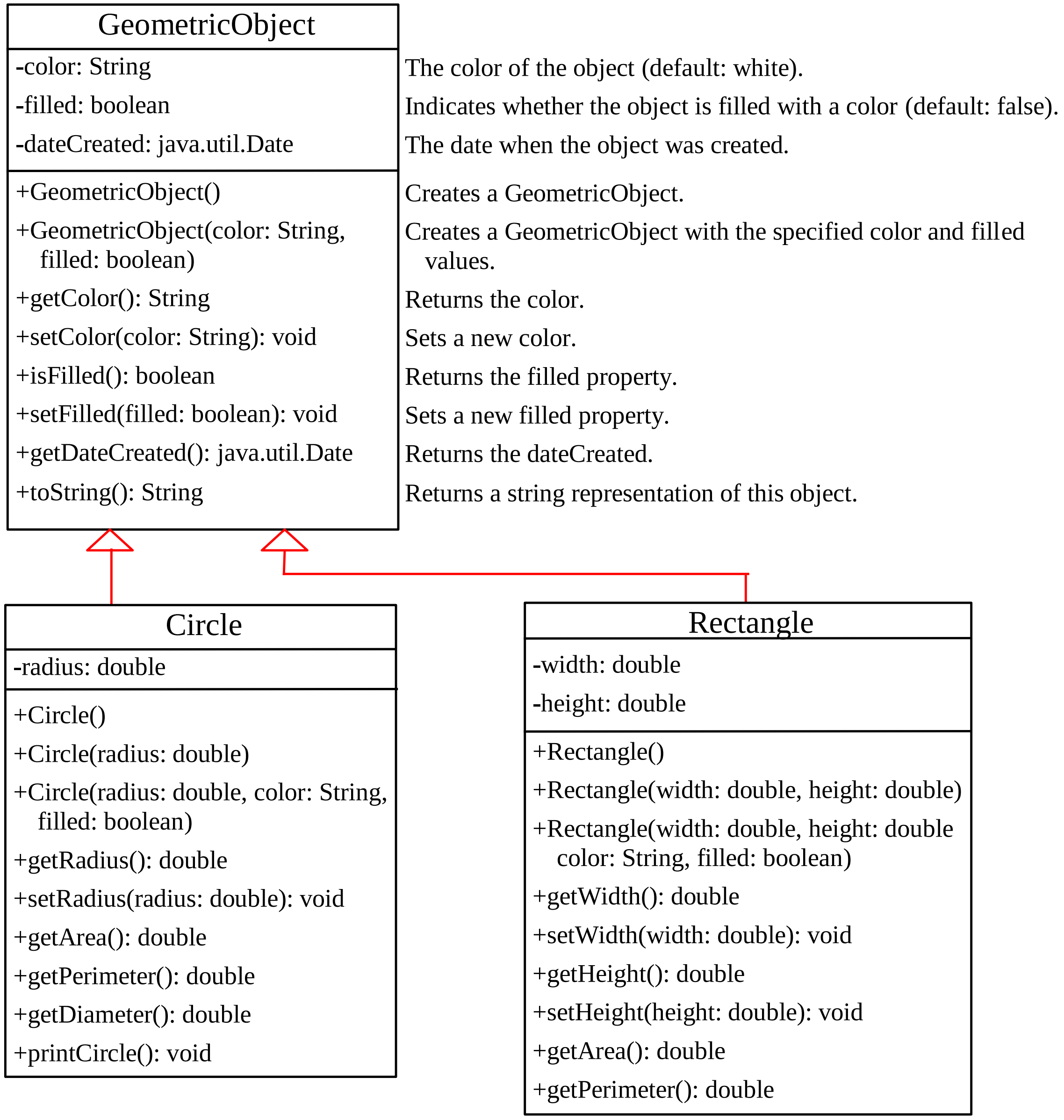
Stony Brook University

http://www.cs.stonybrook.edu/~cse114

Motivation

- Model classes with similar properties and methods:
  - Circles, rectangles and triangles have many common features:
    - **`getArea(): double`**
    - **`getPerimeter(): double`**
- Inheritance: design classes so to avoid redundancy
  - If we have many colored circles, they all implement the same method **`getArea()`**

Superclasses and Subclasses


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

```
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }
  protected GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }
  public String getColor() {   return color;  }
  public void setColor(String color) {  this.color = color;  }
  public boolean isFilled() {   return filled;  }
  public void setFilled(boolean filled) {  this.filled = filled;  }
  public java.util.Date getDateCreated() {   return dateCreated;  }
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }
  /** Abstract method getArea */
  public abstract double getArea();
  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

```java
public class Circle extends GeometricObject {
  private double radius;
  public Circle() {    }
  public Circle(double radius) {
    this.radius = radius;
  }
  public double getRadius() {
    return radius;
  }
  public void setRadius(double radius) {
    this.radius = radius;
  }
  public double getArea() {
    return radius * radius * Math.PI;
  }
  public double getDiameter() {
    return 2 * radius;
  }
  public double getPerimeter() {
    return 2 * radius * Math.PI;
  }
  /* Print the circle info */
  public void printCircle() {
    System.out.println("The circle is created " + getDateCreated() +
      " and the radius is " + radius);
  }
}
```

```
public class Rectangle extends GeometricObject {
  private double width;
  private double height;
  public Rectangle() {  }
  public Rectangle(double width, double height, String color,
        boolean filled) {
    super(color,filled);
    this.width = width;
    this.height = height;
  }
  public double getWidth() {    return width;  }
  public void setWidth(double width) {     this.width = width;  }
  public double getHeight() {    return height;  }
  public void setHeight(double height) {    this.height = height;  }

  public double getArea() {
    return width * height;
  }

  public double getPerimeter() {
    return 2 * (width + height);
  }
}
```

Are superclass's Constructor Inherited?

- No. They are not inherited.
- They are invoked explicitly or implicitly:
  - Explicitly using the **`super`** keyword
  - If the keyword **`super`** is not explicitly used, the superclass's no-arg constructor is automatically invoked as the first statement in the constructor, unless another constructor is invoked (when the last constructor in the chain will invoke the superclass constructor)

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

Using the Keyword `super`

- The keyword **`super`** refers to the superclass of the class in which **`super`** appears. This keyword can be used in two ways:
  - To call a superclass constructor: Java requires that the statement that uses the keyword **`super`** appear first in the constructor (unless another constructor is called or the superclass constructor is called implicitly)
  - To call a superclass method

Constructor Chaining

- ***Constructor chaining*** **: constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain.**

```
public class Faculty extends Employee {
  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
  public static void main(String[] args) {
    new Faculty();
  }
}
class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}
class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee’s no-arg constructor

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

Calling Superclass Methods

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

Declaring a Subclass

- A subclass extends properties and methods from the superclass.
- You can also:

☞ Add new properties

☞ Add new methods

☞ Override the methods of the superclass

Overriding Methods in the Superclass

- *Method overriding*: modify in the subclass the implementation of a method defined in the superclass:

```
public class Circle extends GeometricObject {
  /** Override the toString method defined in
       GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
  // Other methods are omitted
  ...
}
```

Overriding Methods in the Superclass

- An instance method can be overridden only if it is accessible
  - 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
- A static method can be inherited
  - 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

Overriding vs. Overloading

```
public class Test {
  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);
  }
}
```

```
10.0
10.0
```

```
public class Test {
  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);
  }
}
```

```
10
20.0
```

The `Object` Class and Its Methods

- 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 **`java.lang.Object`**

```
public class Circle {
  ...
}
```

Equivalent

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

The <u>`toString()`</u> method in <u>`Object`</u>

- The <u>**`toString()`**</u> method returns a string representation of the object
- The default <u>**`Object`**</u> implementation returns a string consisting of a class name of which the object is an instance, the at sign (@), and a number representing this object

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

- The code displays something like <u>**`Loan@12345e6`**</u>
  - you should override the <u>**`toString()`**</u> method so that it returns an informative string representation of the object

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 – can be invoked with any object

Polymorphism: an object of a subtype can be used wherever its supertype value is required

Dynamic binding**:** the Java Virtual Machine determines dynamically at runtime which implementation is used by the method

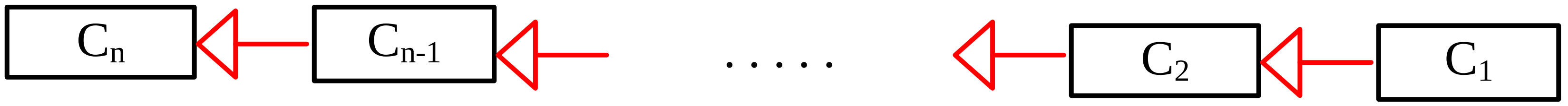
When the method m(Object x) is executed, the argument x's toString method is invoked.

Output:

```
Student
Student
Person
java.lang.Object@12345678
```

Dynamic Binding

- Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n
  - 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
  - C_n is the most general class, and C_1 is the most specific 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, the search stops and the first-found implementation is invoked



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

```
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";
  }
}
```

Output:

```
Student
Student
Person
java.lang.Object@12345678
```

Method Matching vs. Binding

- The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time
- 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";
  }
}
```

Generic programming: polymorphism allows methods to be used generically for a wide range of object arguments: 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) and the particular implementation of the method of the object that is invoked is determined dynamically

Casting Objects

- Casting can be used to convert an object of one class type to another within an inheritance hierarchy

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

is equivalent to:

```
Object o = new Student();    // Implicit casting
m(o);
```

Legal because an instance of Student is automatically an instance of Object

Why Casting Is Necessary?

```
    Student b = o;
```

- A compilation error would occur because an Object o is not necessarily an instance of Student

- We use **explicit casting** to tell the compiler that o is a Student object - syntax is similar to the one used for casting among primitive data types

```
    Student b = (Student)o;
```

- This type of casting may not always succeed (check this with **`instanceof`** operator)

The **instanceof** Operator

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

```
Object myObject = new Circle();
...
if (myObject instanceof Circle) {
 System.out.println("The circle diameter is "
    + ((Circle)myObject).getDiameter());
  ...
}
```

```
public class CastingDemo{
  public static void main(String[] args){
       Object object1 = new Circle(1);
       Object object2 = new Rectangle(1, 1);
       displayObject(object1);
       displayObject(object2);
  }
  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());
       }
  }
}
```

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);
}
```

- Override the **`equals()`** method in other classes:

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

The ArrayList Class

You can create arrays to store objects - But the array's size is fixed once the array is created. Java provides the java.util.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.

```
public class TestArrayList {
  public static void main(String[] args) {          // Warnings
        java.util.ArrayList cityList = new java.util.ArrayList();
        cityList.add("London");cityList.add("New York");cityList.add("Paris");
        cityList.add("Toronto");cityList.add("Hong Kong");
        System.out.println("List size? " + cityList.size());
        System.out.println("Is Toronto in the list? " +
                                 cityList.contains("Toronto"));
        System.out.println("The location of New York in the list? " +
                                 cityList.indexOf("New York"));
        System.out.println("Is the list empty? " + cityList.isEmpty()); // false
        cityList.add(2, "Beijing");
        cityList.remove("Toronto");
        for (int i = 0; i < cityList.size(); i++)
                System.out.print(cityList.get(i) + " ");
        System.out.println();
        // Create a list to store two circles
        java.util.ArrayList list = new java.util.ArrayList();
        list.add(new Circle(2));
        list.add(new Circle(3));
        System.out.println( ((Circle)list.get(0)).getArea() );
}
```

// Generics: eliminates warnings

```
public class TestArrayList {
  public static void main(String[] args) {
        java.util.ArrayList<String> cityList=new java.util.ArrayList<String>();
        cityList.add("London");cityList.add("New York");cityList.add("Paris");
        cityList.add("Toronto");cityList.add("Hong Kong");
        System.out.println("List size? " + cityList.size());
        System.out.println("Is Toronto in the list? " +
                                 cityList.contains("Toronto"));
        System.out.println("The location of New York in the list? " +
                                 cityList.indexOf("New York"));
        System.out.println("Is the list empty? " + cityList.isEmpty()); // false
        cityList.add(2, "Beijing");
        cityList.remove("Toronto");
        for (int i = 0; i < cityList.size(); i++)
                System.out.print(cityList.get(i) + " ");
        System.out.println();
        // Create a list to store two circles
        java.util.ArrayList<Circle> list = new java.util.ArrayList<Circle>();
        list.add(new Circle(2));
        list.add(new Circle(3));
        System.out.println( list.get(0).getArea() );
```

```
}
```

The MyStack Class – Custom stack

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.

```java
public class MyStack {
  private java.util.ArrayList list = new java.util.ArrayList();
  public boolean isEmpty() {
    return list.isEmpty();
  }
  public int getSize() {
    return list.size();
  }
  public Object peek() {
    return list.get(getSize() - 1);
  }
  public Object pop() {
    Object o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }
  public void push(Object o) {
    list.add(o);
  }
  public int search(Object o) {
    return list.lastIndexOf(o);
  }
  public String toString() {
    return "stack: " + list.toString();
  }
}
```

The `protected` Modifier

- 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

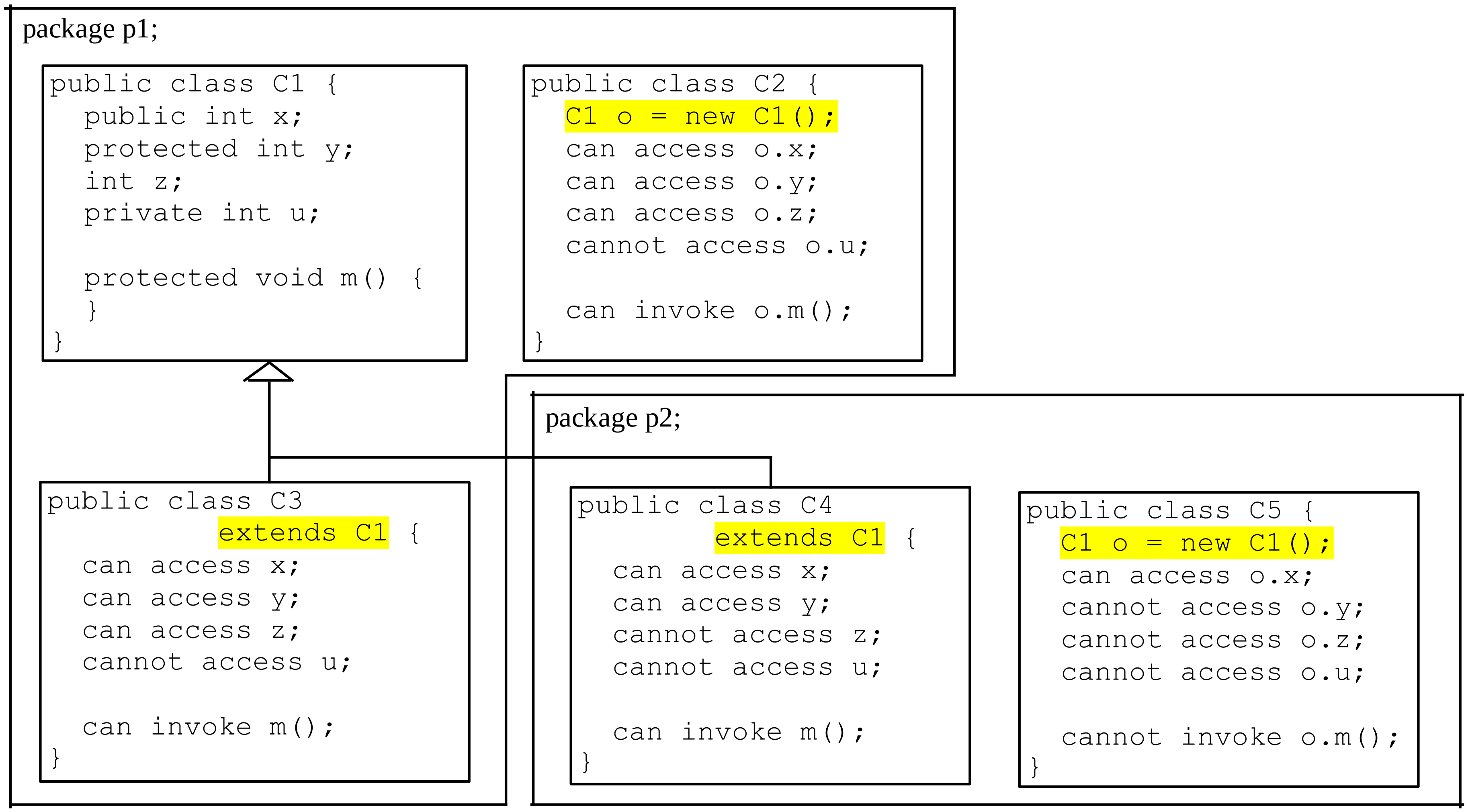
$$\xrightarrow{\texttt{Visibility increases}}$$

`private, default (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 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 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

- A **`final`** variable is a constant:

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

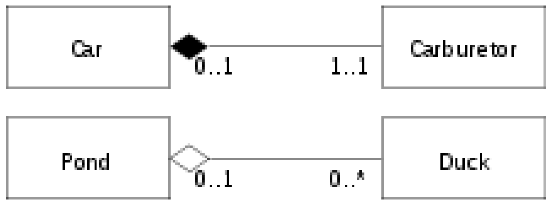
- A **`final`** method cannot be overridden by its subclasses

- A **`final`** class cannot be extended:

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

UML Class Diagram

- **Visibility:**
  - + = Public
  - - = Private
  - # = Protected
  - ~ = Package
  - <u>underline</u> = Static
- **Generalization Relationship**
- **Instance Level Relationships**
  - **Association**
  - **Composition**
  - **Aggregation**
    - if the container is destroyed, its contents are not


Car
0..1
1..1
Carburetor
Pond
0..1
0..*
Duck