Inheritance and Polymorphism

CSE 114: Introduction to Object-Oriented Programming

Paul Fodor

Stony Brook University

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

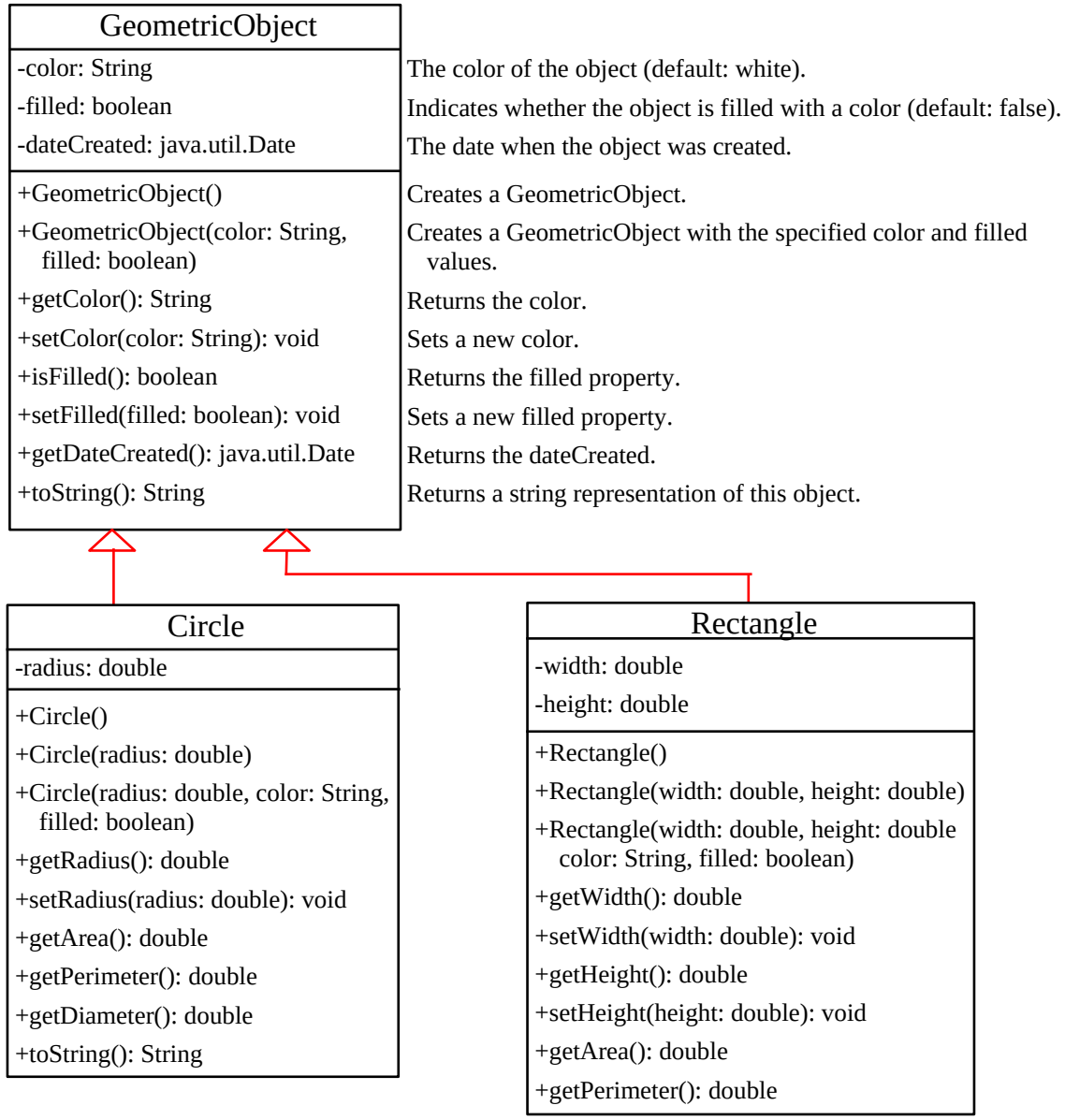
Contents

- Motivation: Model classes with similar properties and methods
- Declaring a Subclass
- Constructor Chaining
- Calling Superclass Methods with super
- Overriding Methods in the Superclass
- The Object Class and Its Methods: toString()
- Overloading vs. Overriding
- Method Matching vs. Binding
- Polymorphism, Dynamic Binding and Generic Programming
- Casting Objects and instanceof Operator
- The equals method
- The ArrayList Class
- The MyStack Class
- The protected and final Modifiers

Motivation

- Model classes with similar properties and methods:
  - Circles, rectangles and triangles have many common features and behaviors (i.e., data fields and methods):
    - **`color: String, isFilled: boolean, dateCreated:Date`**
    - **`getArea(): double`**
    - **`getPerimeter(): double`**
  - ***Inheritance*** is the mechanism of basing a ***sub-class*** on **extending** another ***super-class***
    - Inheritance will help us design and implement classes so to avoid redundancy

Superclasses and Subclasses



```
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) {
    this();
    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 "color: " + color + ", filled: " + filled
      + ", created on " + dateCreated;
  }
  /** Abstract method getArea */
  public abstract double getArea();
  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

```
public class Circle extends GeometricObject {
  private double radius;
  public Circle() {
    super();
  }
  public Circle(double radius) { // super();
    this.radius = radius;
  }
  public double getRadius() {
    return radius;
  }
  public void setRadius(double radius) {
    this.radius = radius;
  }
  public String toString() {
    return "Circle with radius is " + radius + ", " + super.toString();
  }
  public double getArea() {
    return radius * radius * Math.PI;
  }
  public double getPerimeter() {
    return 2 * radius * Math.PI;
  }
  public double getDiameter() {
    return 2 * radius;
  }
}
```

```
public class Rectangle extends GeometricObject {
  private double width;
  private double height;
  public Rectangle() {
    // super();
  }
  public Rectangle(double width, double height) {
    this();
    this.width = width;
    this.height = height;
  }
  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);
  }
}
```

```
public class TestGeometricObject1 {
  public static void main(String[] args) {
    // Declare and initialize two geometric objects
    GeometricObject geoObject1 = new Circle(5);
    GeometricObject geoObject2 = new Rectangle(5, 3);
    // Display circle
    displayGeometricObject(geoObject1);
    // Display rectangle
    displayGeometricObject(geoObject2);
    System.out.println("The two objects have the same area? " +
       equalArea(geoObject1, geoObject2));
  }

  /** A method for displaying a geometric object */
  public static void displayGeometricObject(GeometricObject object) {
    System.out.println(object); // object.toString()
    System.out.println("The area is " + object.getArea());
    System.out.println("The perimeter is " + object.getPerimeter());
  }

  /** A method for comparing the areas of two geometric objects */
  public static boolean equalArea(GeometricObject object1,
      GeometricObject object2) {
    return object1.getArea() == object2.getArea();
  }
}
```

Declaring a Subclass

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

Are superclass's Constructor Inherited?

- No. They are not inherited.
- They are invoked explicitly or implicitly:
  - Explicitly using the **`super`** keyword and the arguments of the superclass constructors as the first statement in the constructor
  - Implicitly: 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 with the keyword **`this`** (in this case, the last constructor in the chain will invoke the superclass constructor)

```
public A(args) {
  // some statements
}
```

is equivalent to

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

The Keyword `super`

- The keyword **`super`** refers to the superclass of the class in which **`super`** appears
- This keyword is used in two ways:
  - To call a superclass constructor (through ***constructor chaining***)
  - To call a superclass method (hidden by the overriding 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 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");
  }
}
```

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() { // super();
    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) { // super();
    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");
  }
}
```

```
(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 ove      ed constructor");
    System.out.println("(3) Emplo      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

(1) Person's no-arg constructor is invoked

(2) Invoke Employee's overloaded 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");
  }
}
```

8. Execute println

```
(1) Person's no-arg constructor is invoked
(2) Invoke Employee's overloaded constructor
(3) Employee'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");
  }
}
```

9. Execute println

```
(1) Person's no-arg constructor is invoked
(2) Invoke Employee's overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty'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) Person's no-arg constructor is invoked
(2) Invoke Employee's overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty's no-arg constructor is invoked"
```

Calling Superclass Methods with **super**

```
public abstract class GeometricObject {
  ...
  public String toString() {
    return "color: " + color + ", filled: " + filled
      + ", date created: " + getDateCreated();
  }
}
class Circle extends GeometricObject {
  ...
  public String toString() {
    return "Circle with radius " + radius
      + ", " + super.toString();
  }
}
```

Overriding Methods in the Superclass

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

```
public abstract class GeometricObject {
  ...
  public String toString() {
    return "color: " + color + ", filled: " + filled
      + ", date created: " + getDateCreated();
  }
}
class Circle extends GeometricObject {
  ...
  public String toString() {
    return "Circle with radius " + radius
      + ", " + super.toString();
  }
}
```

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 GeometricObject {
  ...
}
```

Equivalent

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

The `toString()` method in `Object`

- The **`toString()`** method returns a string representation of the object
- The default **`Object`** 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 **`Loan@12345e6`**
  - you should override the **`toString()`** method so that it returns an informative string representation of the object

Overriding is different than Overloading

- ***Method overloading*** (discussed in Methods) is the ability to create multiple methods of the same name, but with different signatures and implementations:

```
public class Overloading {
  public static int max(int num1, int num2) {
    if (num1 > num2)
      return num1;
    return num2;
  }
  public static double max(double num1, double num2) {
    if (num1 > num2)
      return num1;
    return num2;
  }
  public static void main(String[] args) {
    System.out.println(max(1, 2)); // 2 (as an int)
    System.out.println(max(1.5, 2.3)); // 2.3 (as a double)
  }
}
```

- Method overriding requires that the subclass has the same method signature as in the superclass.

Overloading vs. Overriding

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10.0);
    a.p(10);
  }
}

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);
  }
}
```

```
20.0
10
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10.0);
    a.p(10);
  }
}

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
```

Method Matching vs. Binding

- For overloaded methods, the compiler finds a ***matching*** method according to parameter type, number of parameters, and order of the parameters at compilation time.
- For overridden methods, the Java Virtual Machine ***dynamically binds*** the implementation of the most specific **overridden** method implementation at runtime.

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";
  }
}
```

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

The method **`m`** takes a parameter of the **`Object`** type, so can be invoked with any object.

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

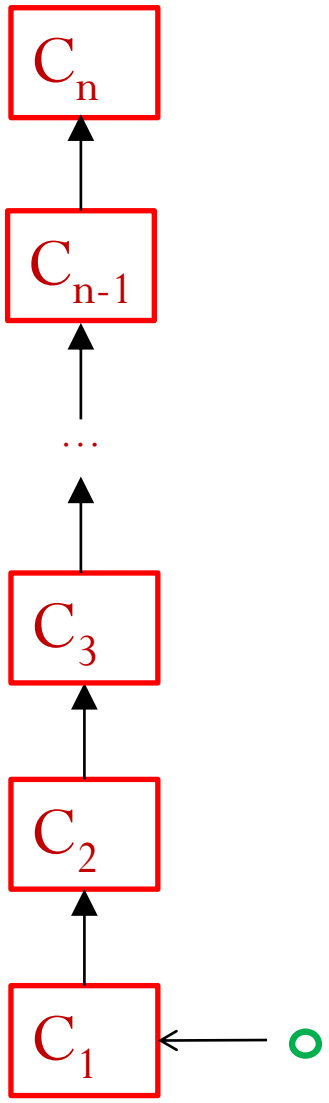
When the method **`m(Object x)`** is executed, the argument **`x`**'s most specific **`toString()`** method is invoked.

Output:

Student
Student
Person
java.lang.Object@12345678

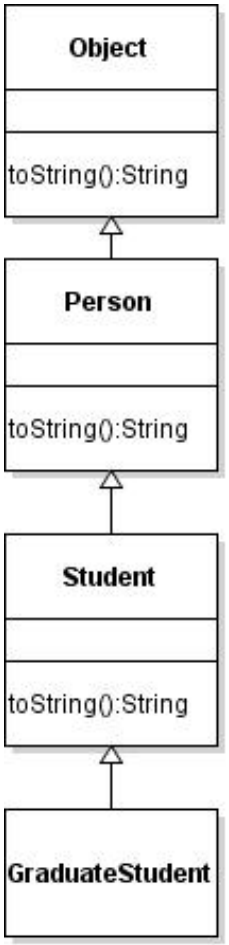
Dynamic Binding

- Suppose an object o is an instance of classes C_1, C_2, …, C_{n-1}, and C_n 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
  - C_n is the most general class (i.e., Object), and C_1 is the most specific class (i.e., the concrete type of o)
  - ***Dynamic Binding***: if o invokes a method m, the JVM searches the implementation for the method m 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



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

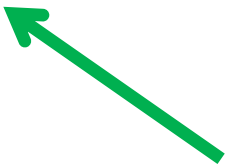
Casting Objects

- ***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 Explicit Casting Is Necessary?

- Sometimes we need to cast down, so we can use methods of the subclass (e.g., **`getGPA()`**)

```
Student b = o; // Syntax Error
```

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

- We must use **explicit casting** to tell the compiler that **`o`** is a **`Student`** object

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

  - the explicit casting syntax is similar to the one used for casting among primitive data types:

```
int i = (int)1.23;
```

The `instanceof` Operator

- Explicit casting may not always succeed (i.e., if the object is not an instance of the subclass)
  - We could use the **`instanceof`** operator to test whether an object is an instance of a class:

```
Object myObject = new Student();
...
if (myObject instanceof Student) {
 System.out.println(
   "The student GPA is "
    + ((Student)myObject).getGPA());
}
```

```
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 (e.g., **`Circle`**):

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

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
- very useful for data-structures

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 finite 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() );
```



```
}
```

```
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() ); // no casting needed
```

// Generics: eliminates warnings


```
}
```

Our MyStack Class – Custom stack

A stack to hold any objects.

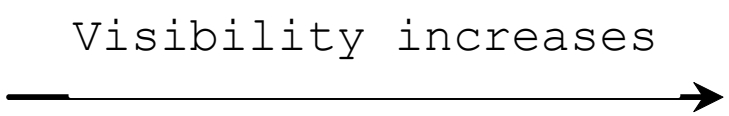
MyStack	
-list: ArrayList	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: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

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

```
public class TestMyStack {
  public static void main(String[] args){
    MyStack s = new MyStack();
    s.push(1);
    s.push(2);
    System.out.println(s.pop()); // 2
    System.out.println(s.pop()); // 1
  }
}
```

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

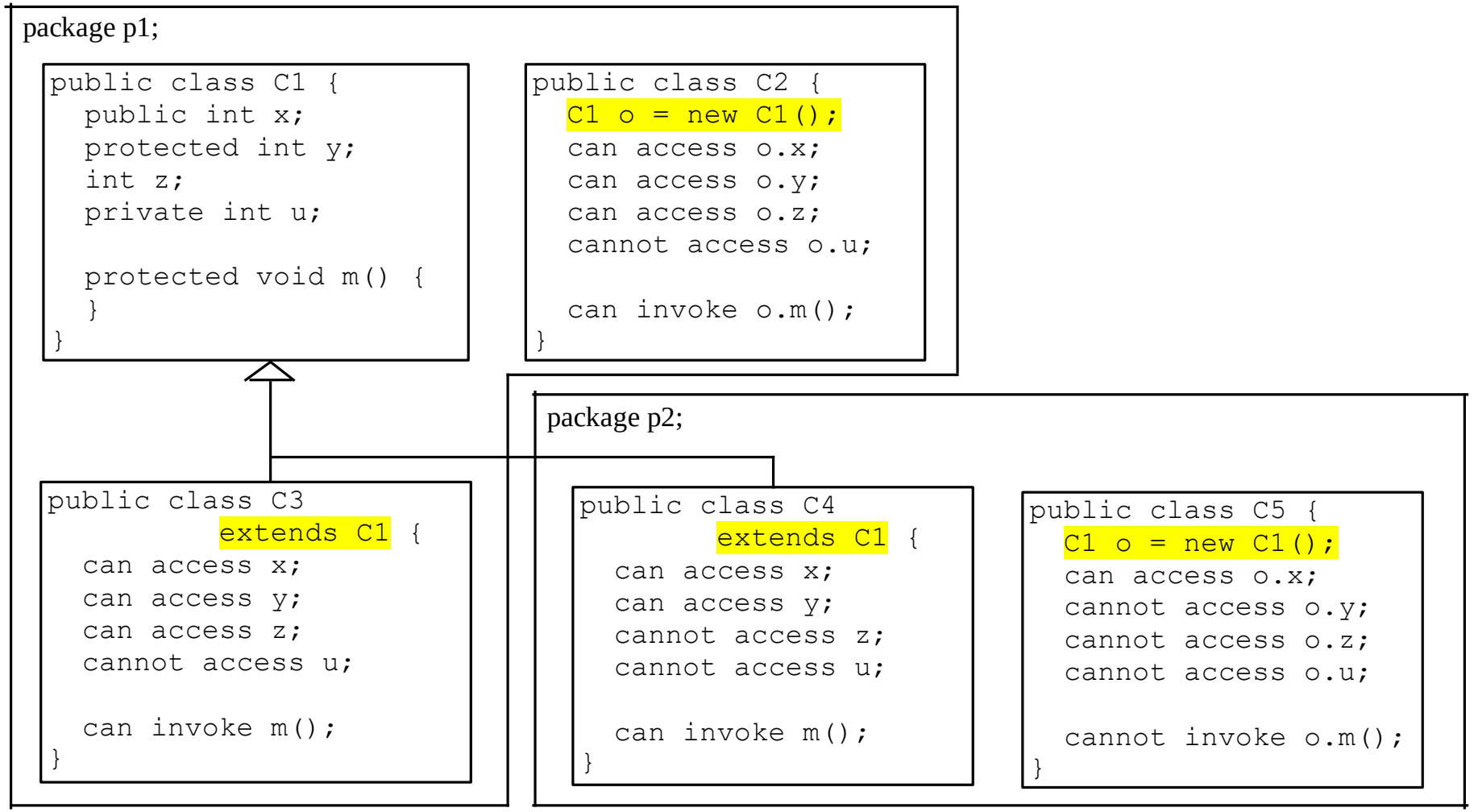


`private, default (if no modifier is used), protected, public`

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



UML Class Diagram

- **Visibility:**
  - + = public
  - - = private
  - ~ = default/package
  - # = protected

  - underlined = static

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.

Overriding Methods in the Subclass

- 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

The `final` Modifier

- Remember that 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 {
   ...
}
```