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 Subclasses
• Constructor Chaining
  • The keyword super
    • Superclass constructors and calling superclass methods
• Overriding Methods
  • Overloading (Method Matching) vs. Overriding (Binding)
  • The Object Class and Its Methods: toString() and equals(Object)
• Polymorphism, Dynamic Binding and Generic Programming
  • Casting Objects and the instanceof operator
• The java.util.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

<table>
<thead>
<tr>
<th>GeometricObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>color: String</td>
</tr>
<tr>
<td>filled: boolean</td>
</tr>
<tr>
<td>dateCreated: java.util.Date</td>
</tr>
<tr>
<td>+GeometricObject()</td>
</tr>
<tr>
<td>+GeometricObject(color: String, filled: boolean)</td>
</tr>
<tr>
<td>+getColor(): String</td>
</tr>
<tr>
<td>+setColor(color: String): void</td>
</tr>
<tr>
<td>+isFilled(): boolean</td>
</tr>
<tr>
<td>+setFilled(filled: boolean): void</td>
</tr>
<tr>
<td>+getDateCreated(): java.util.Date</td>
</tr>
<tr>
<td>+toString(): String</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>radius: double</td>
</tr>
<tr>
<td>+Circle()</td>
</tr>
<tr>
<td>+Circle(radius: double)</td>
</tr>
<tr>
<td>+Circle(radius: double, color: String, filled: boolean)</td>
</tr>
<tr>
<td>+getRadius(): double</td>
</tr>
<tr>
<td>+setRadius(radius: double): void</td>
</tr>
<tr>
<td>+getArea(): double</td>
</tr>
<tr>
<td>+getPerimeter(): double</td>
</tr>
<tr>
<td>+getDiameter(): double</td>
</tr>
<tr>
<td>+toString(): String</td>
</tr>
</tbody>
</table>

The radius of the circle.
Creates a Circle.
Creates a Circle with the specified radius and filled values.
Returns the radius.
Sets a new radius.
Returns the area.
Returns the perimeter.
Returns the diameter.
Returns a string representation of this object.

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>-width: double</td>
</tr>
<tr>
<td>-height: double</td>
</tr>
<tr>
<td>+Rectangle()</td>
</tr>
<tr>
<td>+Rectangle(width: double, height: double)</td>
</tr>
<tr>
<td>+Rectangle(width: double, height: double, color: String, filled: boolean)</td>
</tr>
<tr>
<td>+getWidth(): double</td>
</tr>
<tr>
<td>+setWidth(width: double): void</td>
</tr>
<tr>
<td>+getHeight(): double</td>
</tr>
<tr>
<td>+setHeight(height: double): void</td>
</tr>
<tr>
<td>+getArea(): double</td>
</tr>
<tr>
<td>+getPerimeter(): double</td>
</tr>
</tbody>
</table>

The width and height of the rectangle.
Creates a Rectangle.
Creates a Rectangle with specified width, height, and filled values.
Returns the width.
Sets a new width.
Returns the height.
Sets a new height.
Returns the area.
Returns the perimeter.
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() {
    }
    public Circle(double radius) {
        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();
    }
}

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
Declaring Subclasses

- 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
  - 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.

```java
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) Invite 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");
    }
}
```
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");
    }
}
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");
    }
}
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");
    }
}
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");
    }
}
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");
    }
}
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
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");
    }
}
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");
    }
}
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");
    }
}
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");
    }
}
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:

```java
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();
    }
}
```
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.
  
  ```java
  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:

```java
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, 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

```java
public class Test {
    public static void main(String[] args) {
        B b = new A();
        b.p(10.0);
        b.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);
    }
}
```
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**.
The `equals()` method compares the contents of two objects - the default implementation of the `equals` method in the `Object` class is as follows:

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

Override the `equals()` method in other classes (e.g., `Circle`):

```java
public boolean equals(Object o) {
    if (o instanceof Circle)
        return radius == ((Circle)o).radius;
        //   && super.equals(o);
    else    return false;
}
```
Polymorphism, Dynamic Binding and Generic Programming

```java
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

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
Dynamic Binding

• Suppose an object \( o \) is an instance of classes \( C_1 \) (\( o = \text{new } C_1() \)) where \( C_1 \) is a subclass of \( C_2 \), \( C_2 \) is a subclass of \( C_3 \), \ldots, 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 \( C_1 \), \( C_2 \), \ldots, \( C_{n-1} \) and \( C_n \), **in this order**, until it is found, the search stops and the first-found implementation is invoked
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";
    }
}
Casting Objects

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

  \[
  \text{m(new Student());}
  \]

  is equivalent to:

  \[
  \text{Object o = new Student();} \quad // \text{Implicit casting}
  \]

  \[
  \text{m(o);} \quad // \text{Implicit casting}
  \]

  **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()`)

  ```java
  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

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

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

  ```java
  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:

```java
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 radius is " +
                                ((Circle)object).getRadius());
            System.out.println("The circle diameter is " +
                                ((Circle)object).getDiameter());
        }else if (object instanceof Rectangle) {
            System.out.println("The rectangle width is " +
                                ((Rectangle)object).getWidth());
        }
    }
}
General 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";
    }
}
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:

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ArrayList()</code></td>
<td>Creates an empty list.</td>
</tr>
<tr>
<td><code>add(o: Object) : void</code></td>
<td>Appends a new element o at the end of this list.</td>
</tr>
<tr>
<td><code>add(index: int, o: Object) : void</code></td>
<td>Adds a new element o at the specified index in this list.</td>
</tr>
<tr>
<td><code>clear(): void</code></td>
<td>Removes all the elements from this list.</td>
</tr>
<tr>
<td><code>contains(o: Object): boolean</code></td>
<td>Returns true if this list contains the element o.</td>
</tr>
<tr>
<td><code>get(index: int) : Object</code></td>
<td>Returns the element from this list at the specified index.</td>
</tr>
<tr>
<td><code>indexOf(o: Object) : int</code></td>
<td>Returns the index of the first matching element in this list.</td>
</tr>
<tr>
<td><code>isEmpty(): boolean</code></td>
<td>Returns true if this list contains no elements.</td>
</tr>
<tr>
<td><code>lastIndexOf(o: Object) : int</code></td>
<td>Returns the index of the last matching element in this list.</td>
</tr>
<tr>
<td><code>remove(o: Object): boolean</code></td>
<td>Removes the element o from this list.</td>
</tr>
<tr>
<td><code>size(): int</code></td>
<td>Returns the number of elements in this list.</td>
</tr>
<tr>
<td><code>remove(index: int) : Object</code></td>
<td>Removes the element at the specified index.</td>
</tr>
<tr>
<td><code>set(index: int, o: Object) : Object</code></td>
<td>Sets the element at the specified index.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>MyStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>- list: ArrayList</td>
</tr>
<tr>
<td>+ isEmpty(): boolean</td>
</tr>
<tr>
<td>+ getSize(): int</td>
</tr>
<tr>
<td>+ peek(): Object</td>
</tr>
<tr>
<td>+ pop(): Object</td>
</tr>
<tr>
<td>+ push(o: Object): void</td>
</tr>
<tr>
<td>+ search(o: Object): int</td>
</tr>
</tbody>
</table>

- 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
        MyStack s2 = new MyStack();
        s2.push("New York");
        s2.push("Washington");
        System.out.println(s2.pop()); // New York
        System.out.println(s2.pop()); // Washington
    }
}
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.

  
  
Visibility increases

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

- **We use # for protected in UML**
## Accessibility Summary

<table>
<thead>
<tr>
<th>Modifier on members in a class</th>
<th>Accessed from the same class</th>
<th>Accessed from the same package</th>
<th>Accessed from a subclass</th>
<th>Accessed from a different package</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
</tr>
<tr>
<td>default</td>
<td>✓</td>
<td>✓</td>
<td>−</td>
<td>−</td>
</tr>
<tr>
<td>private</td>
<td>✓</td>
<td>−</td>
<td>−</td>
<td>−</td>
</tr>
</tbody>
</table>
Visibility Modifiers

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

    protected void m() {
    }
}
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();
}
class C3 extends C1 {
    can access x;
    can access y;
    can access z;
    cannot access u;
    can invoke m();
}
class C4 extends C1 {
    can access x;
    can access y;
    cannot access z;
    cannot access u;
    can invoke m();
}
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

- 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 {
      ...
  }
  ```