Inheritance and Polymorphism

CSE 114, Computer Science 1
SUNY Korea

Pradipta De
pradipta.de@sunykorea.ac.kr

Acknowledgement for the slides: Dr. Paul Fodor (CS Stony Brook)
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-color: String</td>
<td>The color of the object (default: white).</td>
</tr>
<tr>
<td>-filled: boolean</td>
<td>Indicates whether the object is filled with a color (default: false).</td>
</tr>
<tr>
<td>-dateCreated: java.util.Date</td>
<td>The date when the object was created.</td>
</tr>
<tr>
<td>+GeometricObject()</td>
<td>Creates a GeometricObject.</td>
</tr>
<tr>
<td>+GeometricObject(color: String, filled: boolean)</td>
<td>Creates a GeometricObject with the specified color and filled values.</td>
</tr>
<tr>
<td>+getColor(): String</td>
<td>Returns the color.</td>
</tr>
<tr>
<td>+setColor(color: String): void</td>
<td>Sets a new color.</td>
</tr>
<tr>
<td>+isFilled(): boolean</td>
<td>Returns the filled property.</td>
</tr>
<tr>
<td>+setFilled(filled: boolean): void</td>
<td>Sets a new filled property.</td>
</tr>
<tr>
<td>+getDateCreated(): java.util.Date</td>
<td>Returns the dateCreated.</td>
</tr>
<tr>
<td>+toString(): String</td>
<td>Returns a string representation of this object.</td>
</tr>
</tbody>
</table>

### Circle

- radius: double

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

### Rectangle

- width: double
- height: double

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

```java
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

```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) 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
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");
    }
}
```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) 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
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 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");
    }
}
What is the error?

```java
public class Apple extends Fruit {
}

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

The default constructor in Apple is a no-arg Constructor. The default Apple constructor invokes constructor of superclass Fruit. The constructor of Fruit requires a parameter. Hence syntax error.

How to fix the error?
What are the outputs?

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

class B extends A {
}

public class C {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

A’s no arg constructor is invoked

```java
class A {
    public A(int x) {
    }
}

class B extends A {
    public B() {
    }
}

public class C {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

Syntax error
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:

```java
public class Circle extends GeometricObject {
    /** Override the toString method defined in GeometricObject */
    public String toString() {
        return super.toString() + "\nnradius 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

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

public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
    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);
  }
}
```
The **Object** Class and Its Methods

- Every class in Java is inherited from the `java.lang.Object` class by default
- If no inheritance is specified when a class is defined, the superclass of the class is `java.lang.Object`

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

Equivalent

```java
public class Circle extends Object {
  ...
}
```
The `toString()` method in `Object`

- The `toString()` method returns a string representation of the object
  - `public String toString() : returns a string representing 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
Polymorphism

- Polymorphism means that a variable of a supertype can refer to an object of the subtype

- Subtype: A type defined by the subclass is the subtype
  - Ex: Classes Circle, Rectangle are subtypes of class GeometricObjects

- Supertype: A type defined by the superclass (or parent class) is the supertype
  - Ex: GeometricObjects is the supertype of Classes of Circle, Rectangle, Triangle.

- Every object of subtype (or an instance of the subclass) is also an object of the supertype (or an instance of the superclass)
Polymorphic method invocation

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

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

- Same method can be implemented in several classes in the class relationship hierarchy. The Java compiler (or Java Virtual Machine – JVM) decides which method to invoke at runtime.

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

- Declared Type: Type that declares a variable
- Actual Type: Type that is assigned based on the Class whose constructor is used

- JVM searches for the implementation of the method starting from lowest child method in the hierarchy till it finds an implementation.
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 $C_n$
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 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

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.

```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";
    }
}
```
Practice: What is the output?

```java
class Person {
    public String getInfo() {
        return "Person";
    }

    public void printPerson() {
        System.out.println(getInfo());
    }
}

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

class Test {
    public static void main(String[] args) {
        new Person().printPerson();
        new Student().printPerson();
    }
}
```
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
## 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

package p1;

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 **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:
  ```java
  final static double PI = 3.14159;
  ```

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

- A **final** class cannot be extended:
  ```java
  final class Math {
      ...
  }
  ```
Casting Objects

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

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

is equivalent to:

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

*Legal* because an instance of Student is automatically an instance of Object.
Why Casting Is Necessary?

\[
\text{Student } b = o;
\]

- A \text{compilation error} would occur because an Object \(o\) is not necessarily an instance of Student

- We use \text{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

\[
\text{Student } b = (\text{Student}) o;
\]

- This type of casting may not always succeed (check this with \text{instanceof} operator)
The `instanceof` Operator

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

```java
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());
    }
  }
}

Note:

- The object member access operator (.) precedes the casting operator. Hence parentheses is used to ensure casting before compiler uses the . Operator

  ```java
  ((Circle)object).getArea();
  ```
Exercise: Error in the code

```java
public class Test {
    public static void main(String[] args) {
        Object fruit = new Fruit();
        Object apple = (Apple)fruit;
    }
}

class Apple extends Fruit {
}

class Fruit {
}
```

Object apple cannot access any method in the class Apple
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:

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

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

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

Use the equals method to compare if two objects have same data values.
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:

<table>
<thead>
<tr>
<th>java.util.ArrayList</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ArrayList()</td>
</tr>
<tr>
<td>+add(o: Object) : void</td>
</tr>
<tr>
<td>+add(index: int, o: Object) : void</td>
</tr>
<tr>
<td>+clear(): void</td>
</tr>
<tr>
<td>+contains(o: Object): boolean</td>
</tr>
<tr>
<td>+get(index: int) : Object</td>
</tr>
<tr>
<td>+indexOf(o: Object) : int</td>
</tr>
<tr>
<td>+isEmpty(): boolean</td>
</tr>
<tr>
<td>+lastIndexOf(o: Object) : int</td>
</tr>
<tr>
<td>+remove(o: Object): boolean</td>
</tr>
<tr>
<td>+size(): int</td>
</tr>
<tr>
<td>+remove(index: int) : Object</td>
</tr>
<tr>
<td>+set(index: int, o: Object) : Object</td>
</tr>
</tbody>
</table>

- Creates an empty list.
- Appends a new element o at the end of this list.
- Adds a new element o at the specified index in this list.
- Removes all the elements from this list.
- Returns true if this list contains the element o.
- Returns the element from this list at the specified index.
- Returns the index of the first matching element in this list.
- Returns true if this list contains no elements.
- Returns the index of the last matching element in this list.
- Removes the element o from this list.
- Returns the number of elements in this list.
- Removes the element at the specified index.
- 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() );
    }
}
<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>ArrayList</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating an array/ArrayList</td>
<td><code>String[] a = new String[10]</code></td>
<td><code>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;();</code></td>
</tr>
<tr>
<td>Accessing an element</td>
<td><code>a[index]</code></td>
<td><code>list.get(index);</code></td>
</tr>
<tr>
<td>Updating an element</td>
<td><code>a[index] = &quot;London&quot;;</code></td>
<td><code>list.set(index, &quot;London&quot;);</code></td>
</tr>
<tr>
<td>Returning size</td>
<td><code>a.length</code></td>
<td><code>list.size();</code></td>
</tr>
<tr>
<td>Adding a new element</td>
<td></td>
<td><code>list.add(&quot;London&quot;);</code></td>
</tr>
<tr>
<td>Inserting a new element</td>
<td></td>
<td><code>list.add(index, &quot;London&quot;);</code></td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td><code>list.remove(index);</code></td>
</tr>
<tr>
<td>Removing an element</td>
<td></td>
<td><code>list.remove(Object);</code></td>
</tr>
<tr>
<td>Removing all elements</td>
<td></td>
<td><code>list.clear();</code></td>
</tr>
</tbody>
</table>