CIS2571 – Intro to Java

Chapter11 → Inheritance and Polymorphism

Information consolidated from a variety of textbook and online resources

Topic Objectives

- Superclasses and Subclasses
- Inheritance
- super Keyword
- Overloading vs. Overriding Methods
- Object Class
- Polymorphism and Dynamic Binding
- Casting Objects
- Protected Access Modifier
- final Keyword
- Classes for Storing Objects

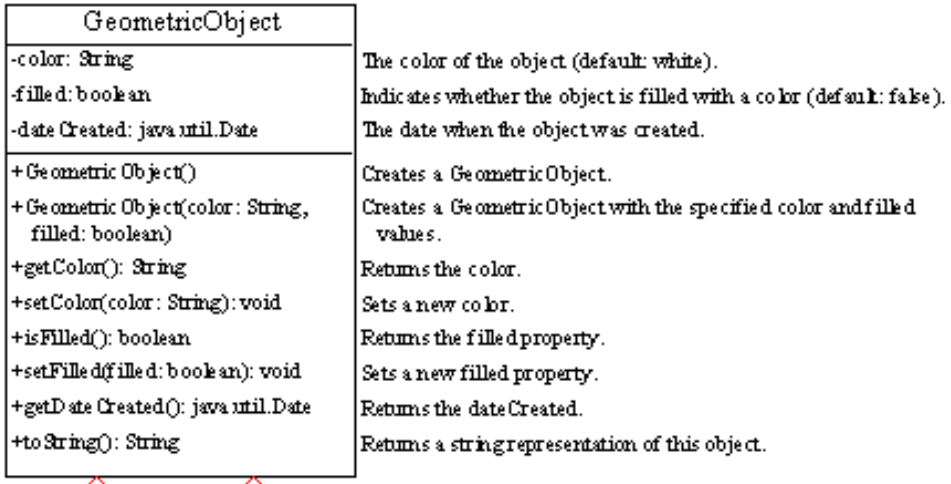
Object Oriented Programming

- When classes have **common** features, they should be designed to avoid redundancy
  - Common properties (fields) and behaviors (methods)
- Inheritance derives new classes from existing classes
  - Specialized class **inherits** properties and methods from general class
  - Models the is-a relationship
- Subclass (aka child class, extended class, derived class) is extended from superclass (aka parent class, base class)
  - Subclass inherits accessible data fields and methods from superclass
  - Subclass may also add new data fields and methods

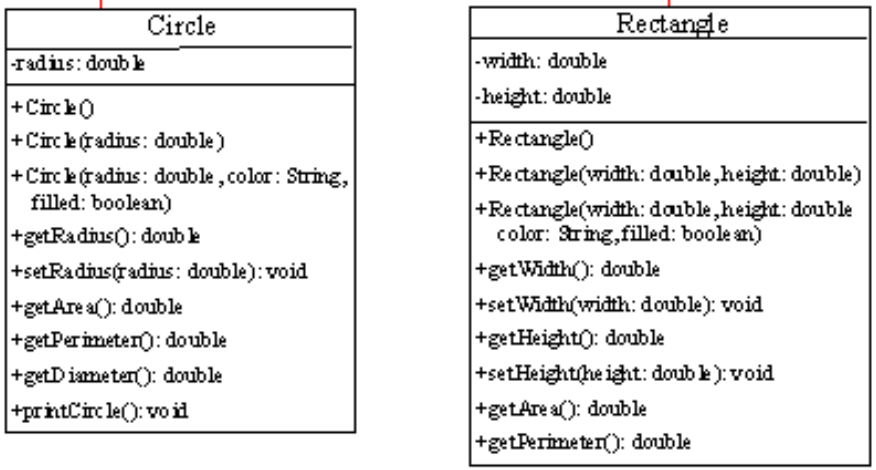
→Example

Superclasses and Subclasses (UML)

Superclass
(base class, parent)



Subclass
(derived class, child)



Superclasses and Subclasses (Java)

```
public class SimpleGeometricObject{
   . . .
}
```

```
public class CircleFromSimpleGeometricObject extends SimpleGeometricObject{
   . . .
}
```

Subclass

Superclass

```
public class RectangleFromSimpleGeometricObject extends SimpleGeometricObject{
   . . .
}
```

See 11.1 SimpleGeometricObject.java
See 11.2 CircleFromSimpleGeometricObjct.java
See 11.3 RectangleFromSimpleGeometricObject.java
See 11.4 TestCircleRectangle.java

OOP Fundamentals

Inheritance

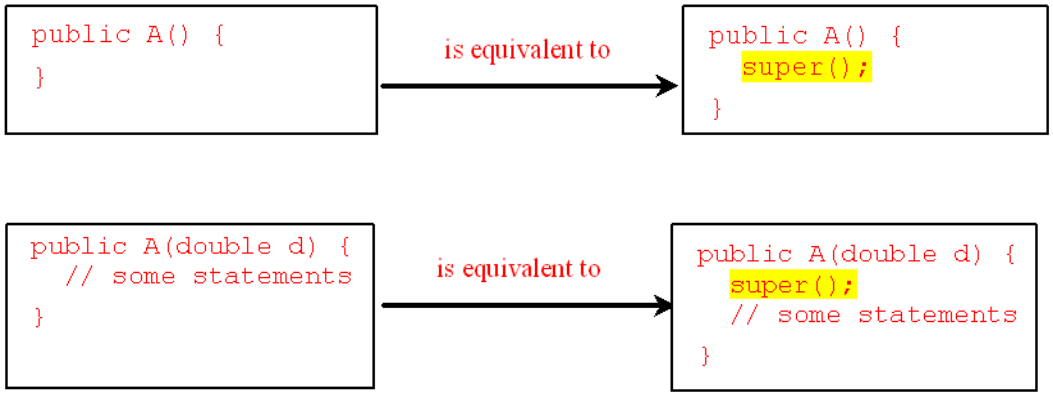
- Subclass usually contains **more** information and methods than its superclass
- Private data fields in superclass are not accessible outside class
  - Cannot be used directly in subclass
  - Can be accessed/mutated through public methods defined in superclass
- Inheritance should be used to model *is-a* relationships **only if there is the need** for extension and more detail in subclass
- Only **single** inheritance is supported in Java
  - Multiple inheritance can be achieved through interfaces

super Keyword

- Subclasses inherit fields and methods from superclass; however, they **do not** inherit superclasses constructors
- **super** keyword used to reference superclass
  - When calling superclass constructor
    - Must be first statement in constructor

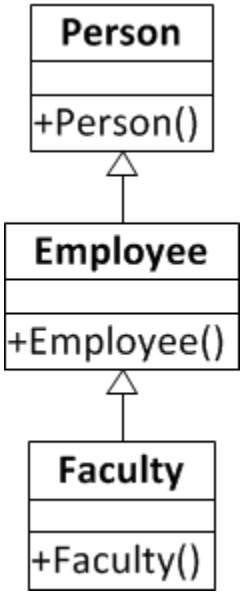
```
super();
super(parameters);
```

  - If no overloaded constructor or superclass constructor is **explicitly invoked**, the superclass no-arg constructor is **automatically invoked**

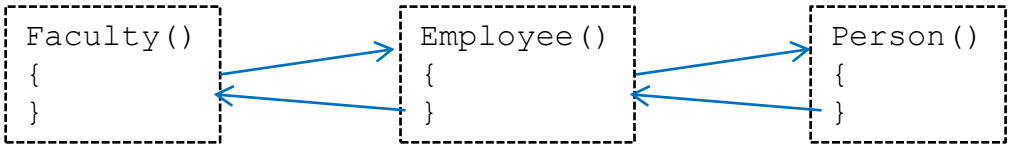


super Keyword

- Constructor chaining → Constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain

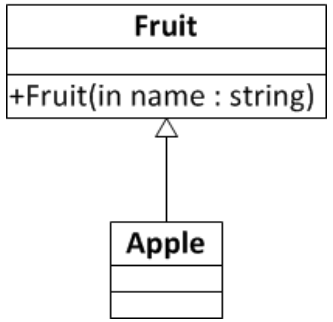


Create Faculty.java



super Keyword

- Important to provide no-arg constructor for class that will be extended to avoid programming errors




```
public class Apple extends Fruit {
// no-arg constructor implicitly defined and calls
// superclass Fruit no-arg constructor
}
class Fruit {
   // no-arg constructor not defined because explicit constructor defined
   public Fruit(String name) {
      System.out.println("Fruit's constructor is invoked");
   }
}
```

super Keyword

- **super** keyword used to reference superclass
  - When calling superclass method (if method not private)

```
super.method(parameters);
```

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

See 11.2 CircleFromSimpleGeometricObjct.java

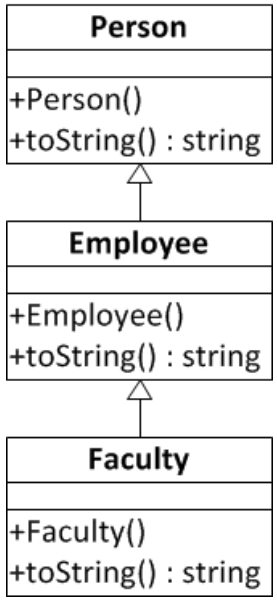
```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

Overriding Methods

- Method overriding → subclass provides new implementation of superclass method
  - Method must have **same signature** (method name and parameter list) and **return type**

```
public class SimpleGeometricObject {
// Other methods are omitted
/** Override the toString method defined in Object */
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }
}
```

See 11.1 SimpleGeometricObject.java



Update Faculty.java

Overriding Methods

- Method overriding → subclass provides new implementation of superclass method
  - Private methods **cannot** be overridden because they are **not accessible** outside the class

```
public class Test {
   public int A(int a) {
   ...
   }
   private int B(int b) {
   ...
   }
}
```

```
public class JavaTest extends Test {
   // overriding method
   public int A(int a) {
   ...
   }
   // unrelated method to
   // superclass Test method
   private int B(int b) {
   ...
   }
}
```

Overriding Methods

- Method overriding → subclass provides new implementation of superclass method
  - Static method is inherited; it **cannot** be overridden
    - If subclass redefines superclass static method, the superclass static method is hidden, but accessible with superclass name

```
public class Test {
   // this static method is tied
   // to Test class
   public static int A(int a) {
   ...
   }
}
```

```
public class JavaTest extends Test {
   // cannot override static method!
   // this static method is tied
   // to JavaTest class
   public static int A(int a) {
   ...
      // can invoke superclass static
      // method with superclass name
      Test.A(1);
   }
}
```

Overriding vs. Overloading

- Overloaded methods are multiple methods with **same** name but **different** signatures
- Overridden methods provide a **new** subclass implementation for **existing** method defined in superclass
  - Overridden method **must have same signature** as superclass

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

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

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

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

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

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

Object Class

- Every class in Java is descended from the java.lang.Object class
  - If no inheritance is specified when class is defined, superclass is Object by default
- toString() method returns a string representation of the object
  - default implementation returns a string consisting of
    - class name of which the object is an instance,
    - the at sign (@), and
    - memory address in hexadecimal

    java.lang.Object@42e816
  - should override in subclass for more readable representation of object
  - implicitly invoked when object is used in an expression requiring a String representation

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

Equivalent

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

Object Class

- equals() method checks whether two reference variables refer to the same object
  - should override in subclass to test for equal object content

```
// default implementation in Object class
public boolean equals(Object obj) {
   return (this == obj);
}
```

```
// overridden in Circle class
// must have same signature
public boolean equals(Object o) {
  // test to see if of type Circle before
  // accessing Circle object members
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

OOP Fundamentals

Polymorphism

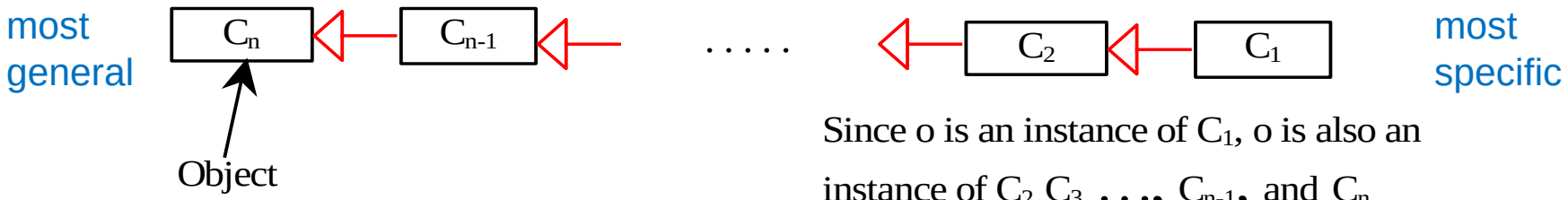
- Subtype defined by subclass
- Supertype defined by superclass
- Every instance of subclass is also instance of superclass
  - But every instance of superclass is not also an instance of subclass
- Polymorphism → instance of subclass can be used wherever instance of superclass is required (aka **generic programming**)

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    displayObject(new CircleFromSimpleGeometricObject(1, "red", false));
    displayObject(new RectangleFromSimpleGeometricObject(1, 1, "black", true));
  }
  public static void displayObject(SimpleGeometricObject object) {
    System.out.println("Created on " + object.getDateCreated() +
      ". Color is " + object.getColor());
  }
}
```

See 11.5 PolymorphismDemo.java

Dynamic Binding

- When method is defined in superclass and overridden in subclass, the **actual method called is determined at runtime**
- Declared type is determined by **variable declaration**
  - Determines matching method at **compile time**
- Actual type is determined by **variable assignment**
  - Determines binding method at **run time**
  - JVM searches from most specific to general when locating method to execute
  - First found implementation is executed

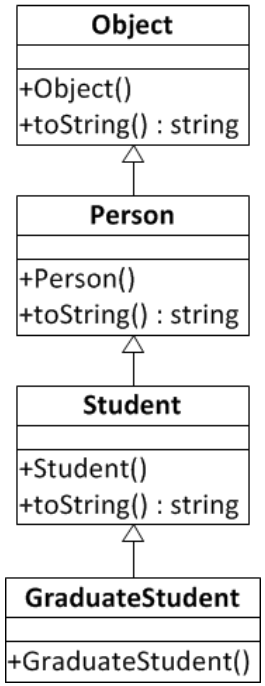


Since o is an instance of C_1, o is also an instance of C_2, C_3, . . ., C_{n-1}, and C_n

→Example

Dynamic Binding

- Dynamic Binding → Java Virtual Machine determines which method to invoke at runtime based upon the actual type.




```
public class DynamicBindingDemo {
   public static void main(String[] args) {
      m(new GraduateStudent());
      m(new Student());
      m(new Person());
      m(new Object());
   }
   public static void m(Object x) {
      System.out.println(x.toString());
   }
}
```

actual type

declared type

See 11.6 DynamicBindingDemo.java

Casting Objects

- Casting can be used to convert object of one type to another within inheritance hierarchy
  - Implicit casting

```
Object o = new Student();
```

    - Okay because instance of subclass is also instance of superclass

```
Student b = o;
```

    - !Okay because instance of superclass is not always instance of subclass (*generates compile error*)

Casting Objects

- Explicit casting → enclose the target object type in parentheses and place it before the object to be cast

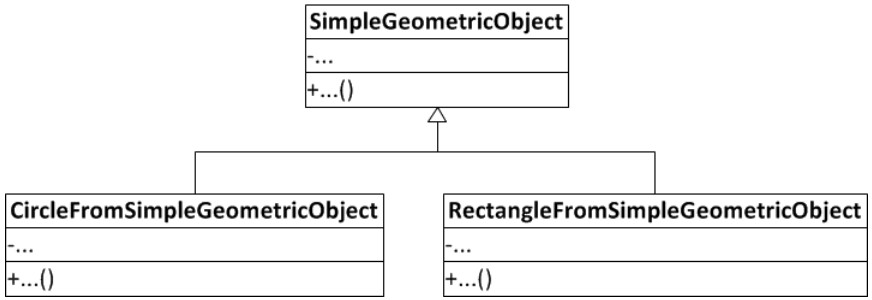
```
Student b = (Student)o;
```

  - When using member access operator (.) with cast, need to enclose casting in parentheses because of higher precedence of . Operator

```
(Student)o.studentMethod();   // !okay
((Student)o).studentMethod();
```

Casting Objects

- Upcasting → **implicitly** casting instance of subclass to variable of superclass is **always allowed** (polymorphism)
  - Okay because instance of subclass is always instance of superclass
- Downcasting → casting instance of superclass to variable of subclass must be done **explicitly** for compiler
  - Instance of superclass is not always instance of subclass
  - Source object to cast **must be** instance of target class
    - Or ClassCastException error occurs during runtime
    - Use instanceof operator used to ensure object is instance of given class



See 11.7 CastingDemo.java

protected Data and Methods

- A protected data, or a protected method, in a public class can be accessed by any subclass in the **same or different package**
  - UML notation uses # for protected access
- Subclass may override a protected method defined in superclass and change its visibility to public
  - Subclasses cannot weaken accessibility of superclass method
- increasing visibility →

```
private, none (no modifier or default), protected, public
```

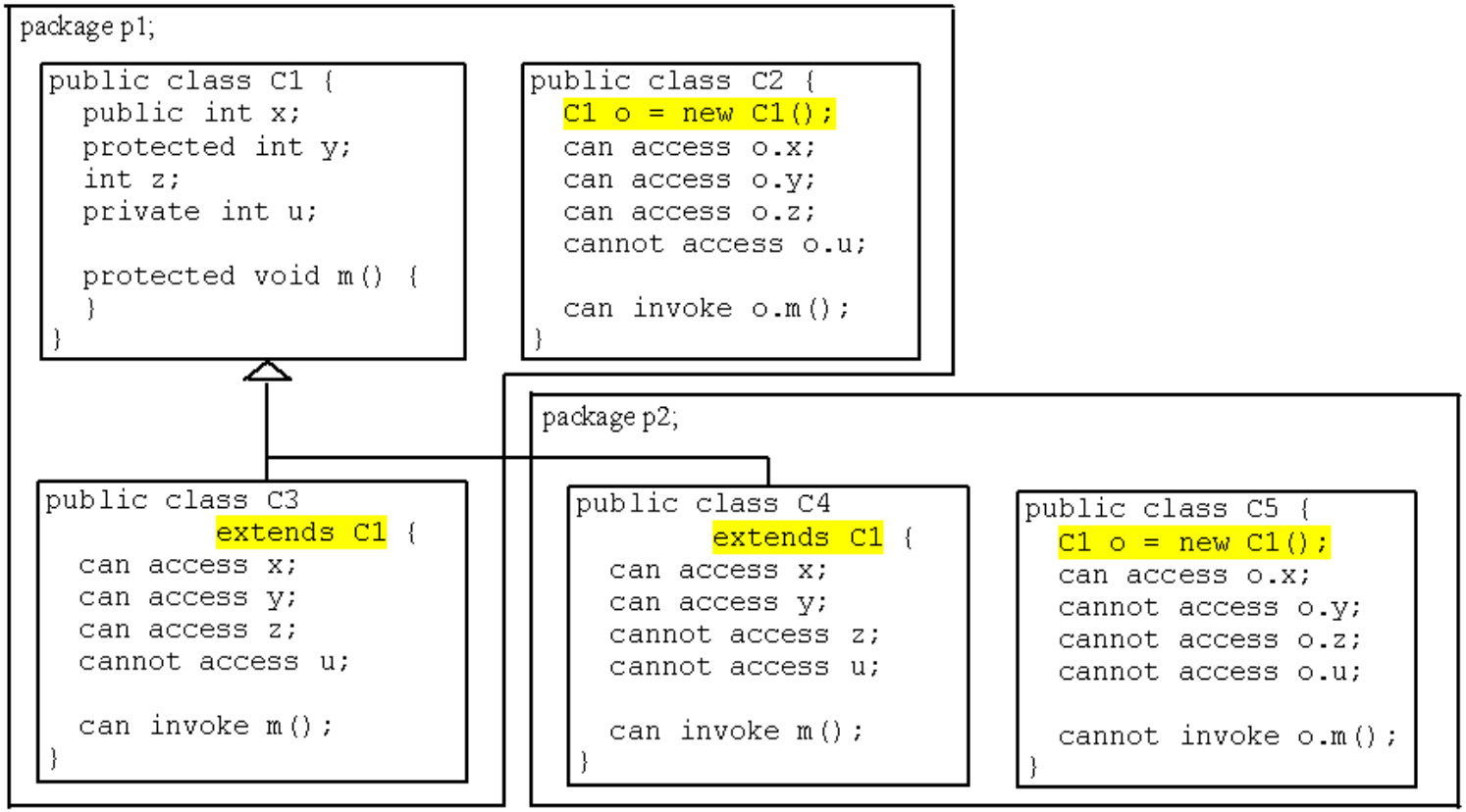
Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	-
default	✓	✓	-	-
private	✓	-	-	-

Access Modifier Guidelines

- private → hides member of class completely so they **cannot** be accessed directly from outside the class
  - available only for **class members**
- no modifiers (default) → allows the class, or member of the class, to be accessed directly from **any** class within the same package but **not** from other packages
  - available for **class** and **class members**
- protected → enables the member of the class to be accessed by the subclasses in **any** package **or** classes in the same package
  - available only for **class members**
- public → enables the class, or member of the class, to be accessed by **any** class
  - available for **class** and **class members**

→Example

Access Modifier Example



Classes and Subclasses

- Classes are used for
  - Creating instances
  - Defining subclasses by extending the class
- Use final keyword to prevent **data fields** from being **modified**
- Use final keyword to prevent **classes** from being **extended**
- Use final keyword to prevent **class methods** from being **overridden**

```
// cannot be superclass
public final class A {
  // data fields, constructors, and methods
}
```

```
public class Test {
  // data fields, constructors, and methods
  // cannot be overridden in subclass
  public final void m() {
    // do something
  }
}
```

Classes For Storing Objects

- ArrayList Class
  - Store an unlimited number of objects
  - Introduced in JDK 1.2 and intended to replace Vector of JDK 1.1
  - Generic class since JDK 1.5
  - Type inferencing allowed in JDK 1.7

```
ArrayList<AConcreteType>  list = new
  ArrayList<AConcreteType>();
// infers type from variable declaration
ArrayList<AConcreteType>  list = new ArrayList<>();
```

- Part of Java Collections Framework

See 11.8 TestArrayList.java
NOTE: Need to compile with -Xlint:unchecked option

→Example

Classes For Storing 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.

- Custom Stack Class
  - Stack class to store objects
  - Implements methods similar to 10.8 StackOfIntegers.java
  - Uses ArrayList to hold data

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.

See 11.10 MyStack.java
NOTE: Need to compile with -Xlint:unchecked option