Object-Oriented OOA, OOD, OOP, and Java

Instructor
Chi-Jen Wu
Postdoc@Sinica
Topics

• Introduction
• Object–Oriented analysis & design
• Object–Oriented programming
• Java tutorial
Anything for coding
Introduction

- Coding for a homework
  - Code size under 500 lines of code
- Coding for an E-campus system
  - ~ 10,000 lines of code
- Coding for a Google Chrome
  - ~7,037,783 lines of C++ code
- Coding for an operating system
  - ~ ?? lines of code
- Coding for a Facebook-like service
  - ?? lines of code
<table>
<thead>
<tr>
<th>Operating System</th>
<th>SLOC (Million)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debian 2.2</td>
<td>55-59</td>
</tr>
<tr>
<td>Debian 3.0</td>
<td>104</td>
</tr>
<tr>
<td>Debian 3.1</td>
<td>215</td>
</tr>
<tr>
<td>Debian 4.0</td>
<td>283</td>
</tr>
<tr>
<td>Debian 5.0</td>
<td>324</td>
</tr>
<tr>
<td>OpenSolaris</td>
<td>9.7</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>8.8</td>
</tr>
<tr>
<td>Mac OS X 10.4</td>
<td>86</td>
</tr>
<tr>
<td>Linux kernel 2.6.0</td>
<td>5.2</td>
</tr>
<tr>
<td>Linux kernel 2.6.29</td>
<td>11.0</td>
</tr>
<tr>
<td>Linux kernel 2.6.32</td>
<td>12.6</td>
</tr>
<tr>
<td>Linux kernel 2.6.35</td>
<td>13.5</td>
</tr>
<tr>
<td>Linux kernel 3.6</td>
<td>15.9</td>
</tr>
</tbody>
</table>
### Windows OS

<table>
<thead>
<tr>
<th>Year</th>
<th>Operating System</th>
<th>SLOC (Million)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1993</td>
<td>Windows NT 3.1</td>
<td>4-5</td>
</tr>
<tr>
<td>1994</td>
<td>Windows NT 3.5</td>
<td>7-8</td>
</tr>
<tr>
<td>1996</td>
<td>Windows NT 4.0</td>
<td>11-12</td>
</tr>
<tr>
<td>2000</td>
<td>Windows 2000</td>
<td>more than 29</td>
</tr>
<tr>
<td>2001</td>
<td>Windows XP</td>
<td>45</td>
</tr>
<tr>
<td>2003</td>
<td>Windows Server 2003</td>
<td>50</td>
</tr>
</tbody>
</table>
Why Object-oriented (OOX)?

Relative Costs of Development Process Phases
(from Boehm via Schach via Turner, slightly modified)

- Maintenance: 67%
- Integration: 6%
- Module testing: 7%
- Module coding: 5%
- Design: 6%
- Requirements: 2%
- Problem definition: 3%
- Planning: 4%
Relative cost to fix a fault that could have been fixed in requirements
The waterfall model
Object-Oriented Analysis & Design

- OOA is a collection of like-minded requirements modeling and analysis techniques for software systems
- Basic idea: streamline software development by making objects, classes, methods and the like atomic units out of which one builds requirements, designs and implementations.
Object–Oriented analysis & design

- Basic Principles of Object Orientation

Object Orientation

- Abstraction
- Encapsulation
- Modularity
- Hierarchy
Abstraction

Salesperson

Not saying
Which
salesperson –
just a
salesperson in
general!!!

Customer

Product

Manages Complexity
Encapsulation

• Hide implementation from clients
  – Clients depend on interface

How does an object encapsulate?
What does it encapsulate?

Improves Resiliency
Modularity

- The breaking up of something complex into manageable pieces

Order Processing System

Order Entry

Order Fulfillment

Billing

Manages Complexity
Hierarchy

• Levels of abstraction

Elements at the same level of the hierarchy should be at the same level of abstraction
Basic Concepts of Object Orientation

- Object
- Class
- Attribute
- Operation
- Interface (Polymorphism)
- Component
- Package
- Subsystem
- Relationships
What is an Object?

- Informally, an object represents an entity, either physical, conceptual, or software
  - Physical entity
  - Conceptual entity
  - Software entity
A More Formal Definition

• An object is a concept, abstraction, or thing with sharp boundaries and meaning for an application

• An object is something that has:
  – State
  – Behavior
  – Identity
Representing Objects

- An object is represented as rectangles with underlined names

: Professor

Class Name Only

ProfessorClark

Object Name Only

ProfessorClark : Professor

Class and Object Name

\[ a + b = 10 \]

Professor Clark

(stay tuned for classes)
What is a Class?

- A class is a description of a group of objects with common properties (attributes), behavior (operations), relationships, and semantics
  - An object is an instance of a class
- A class is an abstraction in that it:
  - Emphasizes relevant characteristics
  - Suppresses other characteristics

*OO Principle: Abstraction*
Sample Class

**Class**
**Course**

**Properties**
- Name
- Location
- Days offered
- Credit hours
- Start time
- End time

**Behavior**
- Add a student
- Delete a student
- Get course roster
- Determine if it is full

\( a + b = 10 \)
Representing Classes

• A class is represented using a compartmented rectangle

Professor

Professor Clark

\[ a + b = 10 \]
Class Compartments

• A class is comprised of three sections
  – The first section contains the class name
  – The second section shows the structure (attributes)
  – The third section shows the behavior (operations)
Classes of Objects

• How many classes do you see?
The Relationship Between Classes and Objects

- A class is an abstract definition of an object
  - It defines the structure and behavior of each object in the class
  - It serves as a template for creating objects
- Objects are grouped into classes
What is an Attribute?

Class

Attribute

CourseOffering

number
startTime
endTime

Object

Attribute Value

:CourseOffering

number = 101
startTime = 900
endTime = 1100

:CourseOffering

number = 104
startTime = 1300
endTime = 1500

What is an Attribute?
What is an Operation?

Class

Operation

CourseOffering

addStudent
deleteStudent
getStartTime
getEndTime
What is Polymorphism?

- The ability to hide many different implementations behind a single interface

*OO Principle: Encapsulation*
What is an Interface?

- Interfaces formalize polymorphism
- Interfaces support “plug-and-play” architectures

Realization relationship

(stay tuned for realization relationships)
What is a Component?

- A non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture
- A component may be
  - A source code component
  - A run time components or
  - An executable component

**OO Principle:**
**Encapsulation**
Relationships: Generalization

- A relationship among classes where one class shares the structure and/or behavior of one or more classes.
- Defines a hierarchy of abstractions in which a subclass inherits from one or more superclasses:
  - Single inheritance
  - Multiple inheritance
- Generalization is an “is–a–kind of” relationship.
Example: Single Inheritance

- One class inherits from another
Example: Multiple Inheritance

- A class can inherit from several other classes

Use multiple inheritance only when needed, and always with caution!
OOP starting

- Objects as a formal concept in programming were introduced in the 1960s in Simula 67
- Alan Curtis Kay
- 2003: ACM Turing Award for work on object-oriented programming
- Defined the conceptual basics for laptop and tablet computers
- The modern overlapping windowing graphical user interface (GUI)
Install Java JDK and IDE

- www.eclipse.org/downloads/
Java Basics

**Step 1:** Write Source Codes

**Step 2:** Compile (Build)

**Step 3:** Run (Execute)
How it works…!

- Java is independent only for one reason:
  - Only depends on the Java Virtual Machine (JVM),
  - code is compiled to *bytecode*, which is interpreted by the resident JVM,
  - JIT (just in time) compilers attempt to increase speed.
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
Object–Oriented

- Java supports OOD
  - Polymorphism
  - Inheritance
  - Encapsulation
- Java programs contain nothing but definitions and instantiations of classes
  - Everything is encapsulated in a class!
Java Advantages

• Portable – Write Once, Run Anywhere
• Security has been well thought through
• Robust memory management
• Designed for network programming
• Multi-threaded (multiple simultaneous tasks)
• Dynamic & extensible (loads of libraries)
  – Classes stored in separate files
  – Loaded only when needed
Getting Started – Write your First Hello–world Java Program

Hello, world!

```java
Hello.java
1   /*
2    * First Java program, which says "Hello, world!"
3    */
4   public class Hello {  // Save as "Hello.java"
5       public static void main(String[] args) {
6         System.out.println("Hello, world!");  // print message
7       }
8   }
```
Java Program Template

```java
public class Classname { // Choose a meaningful Classname. Save as "Classname.java"
    public static void main(String[] args) {
        // Your programming statements here!
    }
}
```

```java
public class PrintTest { // Save as "PrintTest.java"
    public static void main(String[] args) {
        System.out.println("Hello, world!"); // Advance the cursor to the beginning of next line after printing
        System.out.println(); // Print a empty line
        System.out.print("Hello, world!"); // Cursor stayed after the printed string
        System.out.println("Hello,");
        System.out.print(" "); // Print a space
        System.out.print("world!");
        System.out.println("Hello, world!");
    }
}
```

Hello, world!

Hello, world!Hello, world!
Mark $\geq 50 \rightarrow$ print fail
Mark $< 50 \rightarrow$ print pass.

```java
public class CheckPassFail { // saved as "CheckPassFail.java"
    public static void main(String[] args) {
        int mark = 49; // set the value of mark here!
        System.out.println("The mark is " + mark);

        if ( ....... ) {
            System.out.println( ....... );
        } else {
            System.out.println( ....... );
        }
    }
}
```
Object-oriented Programming (OOP) Basics

- Traditional Procedural–Oriented languages

A function (in C) is not well-encapsulated
An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages.
Classes ARE Object Definitions

- OOP – object oriented programming
- code built from objects
- Java these are called *classes*
- Each class definition is coded in a separate .java file
- Name of the object must match the class/object name
The three principles of OOP

- **Encapsulation**
  - Objects hide their functions (**methods**) and data (**variables**)

- **Inheritance**
  - Each **subclass** inherits all variables of its **superclass**

- **Polymorphism**
  - Interface same despite different data types
Simple Class and Method

Class Fruit{
  int grams;
  int cals_per_gram;

  int total_calories() {
    return(grams * cals_per_gram);
  }
}

Methods

• A method is a named sequence of code that can be invoked by other Java code.
• A method takes some parameters, performs some computations and then optionally returns a value (or object).
• Methods can be used as part of an expression statement.

```java
public float convertCelsius(float tempC) {
    return( ((tempC * 9.0f) / 5.0f) + 32.0 );
}
```
Class & Instances

- **Name** (Identifier)
- **Variables** (Static attributes)
- **Methods** (Dynamic behaviors)

**Student**
- name
- grade
- getName()
- printGrade()

**Circle**
- radius
- color
- getRadius()
- getArea()

**SoccerPlayer**
- name
- number
- xLocation
- yLocation
- run()
- jump()
- kickBall()

**Car**
- plateNumber
- xLocation
- yLocation
- speed
- move()
- park()
- accelerate()

Examples of classes

Two instances of the class Student

- **paul:**
  - name="Paul Lee" grade=3.5
  - getName()
  - printGrade()

- **peter:**
  - name="Peter Tan" grade=3.9
  - getName()
  - printGrade()
Class Definition

• The syntax for class definition in Java is:

```java
[AccessControlModifier] class ClassName {
   // class body contains definition of variables and methods
   ...
}
```

```java
public class Circle {
   // class name
   double radius; // variables
   String color;

   double getRadius() {...} // methods
   double getArea() {...}
}
```

```java
public class SoccerPlayer {
   // class name
   int number; // variables
   String name;
   int x, y;

   void run() {...} // methods
   void kickBall() {...}
}
```
Creating Instances of a Class

// Declare 3 instances of the class Circle, c1, c2, and c3
Circle c1, c2, c3;
// Allocate and construct the instances via new operator
c1 = new Circle();
c2 = new Circle(2.0);
c3 = new Circle(3.0, "red");
// You can declare and construct in the same statement
Circle c4 = new Circle();

• Dot Operator

// Declare and construct instances c1 and c2 of the class Circle
Circle c1 = new Circle ();
circle c2 = new Circle ();
// Invoke member methods for the instance c1 via dot operator
System.out.println(c1.getArea());
System.out.println(c1.getRadius());
// Reference member variables for instance c2 via dot operator
c2.radius = 5.0;
c2.color = "blue";
Member Variables & Methods

• The formal syntax for variable definition in Java is:

```java
[AccessControlModifier] type variableName [= initialValue];
[AccessControlModifier] type variableName-1 [= initialValue-1] [, type variableName-2 [= initialValue-2]] ... ;
```

```java
private double radius;
public int length = 1, width = 1;
```

• Member Methods

```java
[AccessControlModifier] returnType methodName ( [argumentList] ) {
   // method body or implementation
   .......
}
```

```java
public double getArea() {
   return radius*radius*Math.PI;
}
```
Public/private

• Methods/data may be declared *public* or *private* meaning they may or may not be accessed by code in other classes ...

• Good practice:
  – keep data private
  – keep most methods private

• well-defined interface between classes – helps to eliminate errors

• *public*: The class/variable/method is accessible and available to all the other objects in the system.

• *private*: The class/variable/method is accessible and available within this class only.
An OOP Example

Class Definition

Circle
- radius: double = 1.0
- color: String = "red"
+ getRadius(): double
+ getColor(): String
+ getArea(): double

Instances

<table>
<thead>
<tr>
<th>c1: Circle</th>
<th>c2: Circle</th>
<th>c3: Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>- radius = 2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- color = &quot;blue&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getRadius()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getColor()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getArea()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- radius = 2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- color = &quot;red&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getRadius()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getColor()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getArea()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- radius = 1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- color = &quot;red&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getRadius()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getColor()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ getArea()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
// Define the Circle class
public class Circle { // Save as "Circle.java"
    // Private variables
    private double radius;
    private String color;

    // Constructors (overloaded)
    public Circle() { // 1st Constructor
        radius = 1.0;
        color = "red";
    }
    public Circle(double r) { // 2nd Constructor
        radius = r;
        color = "red";
    }
    public Circle(double r, String c) { // 3rd Constructor
        radius = r;
        color = c;
    }

    // Public methods
    public double getRadius() {
        return radius;
    }
    public String getColor() {
        return color;
    }
    public double getArea() {
        return radius*radius*Math.PI;
    }
}
// Test driver program for the Circle class
public class TestCircle {
    // Save as "TestCircle.java"
    public static void main(String[] args) {
        // Execution entry point
        // Construct an instance of the Circle class called c1
        Circle c1 = new Circle(2.0, "blue"); // Use 3rd constructor
        System.out.println("Radius is " + c1.getRadius() // use dot operator to invoke member methods
            + " Color is " + c1.getColor()
            + " Area is " + c1.getArea());

        // Construct another instance of the Circle class called c2
        Circle c2 = new Circle(2.0); // Use 2nd constructor
        System.out.println("Radius is " + c2.getRadius()
            + " Color is " + c2.getColor()
            + " Area is " + c2.getArea());

        // Construct yet another instance of the Circle class called c3
        Circle c3 = new Circle(); // Use 1st constructor
        System.out.println("Radius is " + c3.getRadius()
            + " Color is " + c3.getColor()
            + " Area is " + c3.getArea());
    }
}

Radius is 2.0 Color is blue Area is 12.566370614359172
Radius is 2.0 Color is red Area is 12.566370614359172
Radius is 1.0 Color is red Area is 3.141592653589793
Constructors

- The name of the constructor method is the same as the class name, and by classname convention, begins with an uppercase.
- Constructor has no return type
- Constructor can only be invoked via the "new" operator.
- Constructors are not inherited
- A constructor with no parameter is called the default constructor

```java
Circle c1 = new Circle();
Circle c2 = new Circle(2.0);
Circle c3 = new Circle(3.0, "red");
```
Method Overloading

```java
// Example to illustrate Method Overloading
public class TestMethodOverloading {
    public static int average(int n1, int n2) { // A
        return (n1+n2)/2;
    }

    public static double average(double n1, double n2) { // B
        return (n1+n2)/2;
    }

    public static int average(int n1, int n2, int n3) { // C
        return (n1+n2+n3)/3;
    }

    public static void main(String[] args) {
        System.out.println(average(1, 2)); // Use A
        System.out.println(average(1.0, 2.0)); // Use B
        System.out.println(average(1, 2, 3)); // Use C
        System.out.println(average(1.0, 2)); // Use B - int 2 implicitly casted to double 2.0
        // average(1, 2, 3, 4); // Compilation Error - No matching method
    }
}
```

- Overloading Circle Class' Constructor

```java
Circle()
Circle(double r)
Circle(double r, String c)
```
Information Hiding and Encapsulation

• **Getters and Setters**
  – To allow other classes to read the value of a private variable

```java
// Setter for color
class Color {
    private String color;

    public void setColor(String c) {
        color = c;
    }
}

// Setter for radius
class Circle {
    private double radius;

    public void setRadius(double r) {
        radius = r;
    }
}
```
Keyword "this"

- You can use keyword "this" to refer to this instance inside a class definition.

```java
public class Circle {
    double radius; // Member variable called "radius"
    public Circle(double radius) { // Method's argument also called "radius"
        this.radius = radius;
        // "this.radius" refers to this instance's member variable
        // "radius" resolved to the method's argument.
    }
    ...
}
```
public class Aaa {
    // A private variable named xxx of type T
    private T xxx;

    // Constructor
    public Aaa(T xxx) {
        this.xxx = xxx;
    }

    // A getter for variable xxx of type T receives no argument and return a value of type T
    public T getXxx() {
        return xxx;
    }

    // A setter for variable xxx of type T receives a parameter of type T and return void
    public void setXxx(T xxx) {
        this.xxx = xxx;
    }
}
Method toString()

- Every well-designed Java class should have a public method called toString() that returns a string description of the object.

```java
// Return a short String description of this instance
public String toString() {
    return "Circle with radius = " + radius + " and color of " + color;
}
```

```java
Circle c1 = new Circle();
System.out.println(c1.toString());  // explicitly calling toString()
System.out.println(c1);            // implicit call to c1.toString()
System.out.println("c1 is: " + c1); // '+' invokes c1.toString() to get a String before concatenation
```
Constants (final)

- Constants are variables defined with the modifier final.
- A final variable can only be assigned once and its value cannot be modified once assigned.

```java
public final double XgetReference = 1.234;

private final int MAX_ID = 9999;
MAX_ID = 10000;  // error: cannot assign a value to final variable MAX_ID

// You need to initialize a final member variable during declaration
private final int SIZE;  // error: variable SIZE might not have been initialized
```
Putting Them Together in the Revised Circle Class

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>-radius:double = 1.0</td>
</tr>
<tr>
<td>-color:String = &quot;red&quot;</td>
</tr>
</tbody>
</table>

+Circle(radius:double,color:String)  
+Circle(radius:double)  
+Circle()  
+getRadius():double  
+setRadius(radius:double):void  
+getColor():String  
+setColor(color:String):void  
+getArea():double  
+toString():String
```java
// The Circle class definition
public class Circle { // Save as "Circle.java"

    // Public constants
    public static final double DEFAULT_RADIUS = 8.8;
    public static final String DEFAULT_COLOR = "red";

    // Private variables
    private double radius;
    private String color;

    // Constructors (overloaded)
    public Circle() { // 1st Constructor
        radius = DEFAULT_RADIUS;
        color = DEFAULT_COLOR;
    }
    public Circle(double radius) { // 2nd Constructor
        this.radius = radius;
        color = DEFAULT_COLOR;
    }
    public Circle(double radius, String color) { // 3rd Constructor
        this.radius = radius;
        this.color = color;
    }
```

```
// Public getter and setter for private variables
public double getRadius() {
    return radius;
}
public void setRadius(double radius) {
    this.radius = radius;
}
public String getColor() {
    return color;
}
public void setColor(String color) {
    this.color = color;
}

// toString() to provide a short description of this instance
public String toString() {
    return "Circle with radius = " + radius + " and color of " + color;
}

// Public methods
public double getArea() {
    return radius*radius*Math.PI;
}
TestCircle.java

```java
// Test driver program for the Circle class
public class TestCircle {     // Save as "TestCircle.java"
    public static void main(String[] args) {     // Execution entry point
        // Construct an instance of the Circle class called c1
        Circle c1 = new Circle(2.0, "blue");     // Use 3rd constructor
        System.out.println("Radius is " + c1.getRadius() // use dot operator to invoke member methods
                        + " Color is " + c1.getColor()
                        + " Area is " + c1.getArea());

        // Construct another instance of the Circle class called c2
        Circle c2 = new Circle(2.0);     // Use 2nd constructor
        System.out.println("Radius is " + c2.getRadius()
                        + " Color is " + c2.getColor()
                        + " Area is " + c2.getArea());

        // Construct yet another instance of the Circle class called c3
        Circle c3 = new Circle();     // Use 1st constructor
        System.out.println("Radius is " + c3.getRadius()
                        + " Color is " + c3.getColor()
                        + " Area is " + c3.getArea());
    }
}
```

Radius is 2.0 Color is blue Area is 12.566370614359172
Radius is 2.0 Color is red Area is 12.566370614359172
Radius is 1.0 Color is red Area is 3.141592653589793
The MyTime class

- hour: between 00 to 23.
- minute: between 00 to 59.
- Second: between 00 to 59.
```java
// The MyTime class definition
public class MyTime {    // "MyTime.java"

    // Private member variables
    private int hour;    // 0-23
    private int minute;  // 0-59
    private int second;  // 0-59

    // Constructor
    public MyTime(int hour, int minute, int second) {
        setTime(hour, minute, second);
    }

    void setTime(int hour, int minute, int second) {
        setHour(hour);
        setMinute(minute);
        setSecond(second);
    }

    // Setters which validate input with exception handling
    void setHour(int hour) {
        if (hour >= 0 && hour <= 23) {
            this.hour = hour;
        } else {
            throw new IllegalArgumentException("Invalid hour!");
        }
    }

    void setMinute(int minute) {
        if (minute >= 0 && minute <= 59) {
            this.minute = minute;
        } else {
            throw new IllegalArgumentException("Invalid minute!");
        }
    }

    // Getters
    public int getHour() {
        return hour;
    }

    public int getMinute() {
        return minute;
    }

    public int getSecond() {
        return second;
    }

    // Return description in the format "hh:mm:ss" with leading zeros
    public String toString() {
        return String.format("%02d:%02d:%02d", hour, minute, second);
    }

    // Increment this instance to the next second, return this instance
    public MyTime nextSecond() {
        ++second;
        if (second == 60) {
            second = 0;
            ++minute;
        }
        if (minute == 60) {
            minute = 0;
            ++hour;
        }
        if (hour == 24) {
            hour = 0;
        }
        return this;    // Return this instance, to support cascaded operation
    }
}
```
Exception Handling

• In Java, instead of printing an error message, you can throw an so-called Exception object to the caller

```java
void setHour(int hour) {
    if (hour >= 0 && hour <= 23) {
        this.hour = hour;
    } else {
        throw new IllegalArgumentException("Invalid hour!");
    }
}
```

```java
try {
    MyTime t3 = new MyTime(12, 69, 69);
    // skip remaining statements in try, goto catch
    System.out.println(t1);
} catch (IllegalArgumentException ex) {
    ex.printStackTrace();
} // after catch, continue next statement
```
// A test driver program for MyTime
public class TestMyTime {
    public static void main(String[] args) {
        MyTime t1 = new MyTime(23, 59, 58);
        System.out.println(t1);
        System.out.println(t1.nextSecond());
        System.out.println(t1.nextSecond().nextSecond().nextSecond());

        // MyTime t2 = new MyTime(12, 69, 69); // abrupt termination
        // NOT continue to next statement

        // Handling exception gracefully
        try {
            MyTime t3 = new MyTime(12, 69, 69);
            // skip remaining statements in try, goto catch
            System.out.println(t1);
        } catch (IllegalArgumentException ex) {
            ex.printStackTrace();
        } // after try or catch, continue next statement

        System.out.println("Continue after exception!");
    }
}
There are two ways to reuse the existing classes, namely,

- **Composition**
- **Inheritance**

<table>
<thead>
<tr>
<th>Point</th>
</tr>
</thead>
</table>
| - x:int  
- y:int |
| +Point() |
| +Point(x:int, y:int) |
| +getX():int |
| +setX(x:int):void |
| +getY():int |
| +setY(y:int):void |
| +toString():String |
Re-use the Point class via composition

- A line is composed of two points
- A line has two points
public class Point {
    private int x, y; // (x, y) co-ordinates

    // Constructors
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Point() { // default (no-arg) constructor
        x = 0;
        y = 0;
    }

    // Public getter and setter for private variables
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }

    // toString() to describe itself
    public String toString() {
        return "(" + x + "," + y + ")";
    }
}
// The Line class definition

public class Line {

    // Private member variables
    private Point begin, end; // Declare begin and end as instances of Point

    // Constructors
    public Line(int x1, int y1, int x2, int y2) {
        begin = new Point(x1, y1); // Construct Point instances
        end = new Point(x2, y2);
    }
    public Line(Point begin, Point end) {
        this.begin = begin; // Caller constructed Point instances
        this.end = end;
    }

    // Public getter and setter for private variables
    public Point getBegin() {
        return begin;
    }
    public Point getEnd() {
        return end;
    }
    public void setBegin(Point begin) {
        this.begin = begin;
    }
    public void setEnd(Point end) {
        this.end = end;
    }
    public int getBeginX() {
        return begin.getX();
    }
    public void setBeginX(int x) {
        begin.setX(x);
    }
    public int getBeginY() {
        return begin.getY();
    }

    public void setBeginXY(int x, int y) {
        begin.setX(x);
        begin.setY(y);
    }

    public int getEndX() {
        return end.getX();
    }
    public void setEndX(int x) {
        end.setX(x);
    }

    public int getEndY() {
        return end.getY();
    }
    public void setEndY(int y) {
        end.setY(y);
    }

    public void setEndXY(int x, int y) {
        end.setX(x);
        end.setY(y);
    }

    public String toString() {
        return "Line from " + begin + " to " + end;
    }

    public double getLength() {
        int xDiff = begin.getX() - end.getX();
        int yDiff = begin.getY() - end.getY();
        return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
    }
}

TestLine.java

// A test driver program for the Line class
public class TestLine {
    public static void main(String[] args) {
        Line l1 = new Line(0, 3, 4, 0);
        System.out.println(l1); // toString()
        System.out.println(l1.getLength());
        l1.setBeginXY(1, 2);
        l1.setEndXY(3, 4);
        System.out.println(l1);
        Point p1 = new Point(3, 0);
        Point p2 = new Point(0, 4);
        Line l2 = new Line(p1, p2);
        System.out.println(l2);
        System.out.println(l2.getLength());
        l2.setBegin(new Point(5, 6));
        l2.setEnd(new Point(7, 8));
        System.out.println(l2);
    }
}

Line.java

// Constructors
public Line(int x1, int y1, int x2, int y2) {
    begin = new Point(x1, y1); // Construct Point instances
    end = new Point(x2, y2);
}
public Line(Point begin, Point end) {
    this.begin = begin; // Caller constructed Point instances
    this.end = end;
}
// A test driver program for the Line class
public class TestLine {
    public static void main(String[] args) {
        Line l1 = new Line(0, 3, 4, 0); // toString()
        System.out.println(l1);
        System.out.println(l1.getLength());
        l1.setBeginXY(1, 2);
        l1.setEndXY(3, 4);
        System.out.println(l1);

        Point p1 = new Point(3, 0);
        Point p2 = new Point(0, 4);
        Line l2 = new Line(p1, p2);
        System.out.println(l2);
        System.out.println(l2.getLength());
        l2.setBegin(new Point(2, 5)); // Construct Point instances
        l2.setEnd(new Point(7, 8));
        System.out.println(l2);
    }
}
Inheritance

- Often organize classes in hierarchy to avoid duplication and reduce redundancy
- A class in the lower hierarchy is called a subclass (or derived, child, extended class)
- A class in the upper hierarchy is called a superclass (or base, parent class)
Inheritance

• It is important to note that a subclass is not a "subset" of a superclass
• Subclass is a "superset" of a superclass
  – A subclass inherits all the variables and methods of the superclass
• class Goalkeeper extends SoccerPlayer
  {......}
An Example on Inheritance

```java
// Define Cylinder class, which is a subclass of Circle
public class Cylinder extends Circle {
    private double height; // Private member variable

    public Cylinder() {   // constructor 1
        super(); // invoke superclass' constructor Circle()
        height = 1.0;
    }

    public Cylinder(double radius, double height) {  // Constructor 2
        super(radius); // invoke superclass' constructor Circle(radius)
        this.height = height;
    }

    public double getHeight() {  // Use Circle's getArea()
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public double getVolume() {
        return getArea()*height; // Use Circle's getArea()
    }
}
```
Circle Class

```java
public class Circle {  // Save as Circle.java
   // Public constants
   public static final double DEFAULT_RADIUS = 1.0;
   public static final String DEFAULT_COLOR = "red";

   // Private variables
   private double radius;
   private String color;

   // Constructors (over-loaded)
   public Circle() {  // 1st Constructor
      radius = DEFAULT_RADIUS;
      color = DEFAULT_COLOR;
   }
   public Circle(double radius) {  // 2nd Constructor
      this.radius = radius;
      color = DEFAULT_COLOR;
   }
   public Circle(double radius, String color) {  // 3rd Constructor
      this.radius = radius;
      this.color = color;
   }

   // Public getter and setter for private variables
   public double getRadius() {
      return radius;
   }
   public void setRadius(double radius) {
      this.radius = radius;
   }
   public String getColor() {
      return color;
   }
   public void setColor(String color) {
      this.color = color;
   }

   // toString() to provide a short description of this instance
   public String toString() {
      return "Circle with radius = " + radius + " and color of " + color;
   }

   // Public methods
   public double getArea() {
      return radius * radius * Math.PI;
   }
}
```

Circle

- radius:double = 1.0
- color:String = "red"

+Circle(radius:double,color:String)
+Circle(radius:double)
+Circle()
+getRadius():double
+setRadius(radius:double):void
+getColor():String
+setColor(color:String):void
+getArea():double
+toString():String
public class TestCylinder {
  public static void main(String[] args) {
    Cylinder cy1 = new Cylinder(); // Use constructor 1
    System.out.println("Radius is " + cy1.getRadius() + " Height is " + cy1.getHeight() + " Color is " + cy1.getColor() + " Base area is " + cy1.getArea() + " Volume is " + cy1.getVolume());
    Cylinder cy2 = new Cylinder(5.0, 2.0); // Use constructor 2
    System.out.println("Radius is " + cy2.getRadius() + " Height is " + cy2.getHeight() + " Color is " + cy2.getColor() + " Base area is " + cy2.getArea() + " Volume is " + cy2.getVolume());
  }
}
But….  

- Overriding the inherited method getArea()

```java
public class Cylinder extends Circle {
    ......
    // override the getArea() method inherited from superclass Circle
    @Override
    public double getArea() {
        return 2*Math.PI*getRadius()*height + 2*super.getArea();
    }
    // need to change the getVolume() as well
    public double getVolume() {
        return super.getArea()*height;  // use superclass' getArea()
    }
    // override the inherited toString()
    @Override
    public String toString() {
        return "Cylinder: radius = " + getRadius() + " height = " + height;
    }
}
```
// A test driver program for Cylinder class
public class TestCylinder {
    public static void main(String[] args) {
        Cylinder cy1 = new Cylinder(); // Use constructor 1
        System.out.println("Radius is " + cy1.getRadius() + " Height is " + cy1.getHeight() + " Color is " + cy1.getColor() + " Base area is " + cy1.getArea() + " Volume is " + cy1.getVolume());

        Cylinder cy2 = new Cylinder(5.0, 2.0); // Use constructor 2
        System.out.println("Radius is " + cy2.getRadius() + " Height is " + cy2.getHeight() + " Color is " + cy2.getColor() + " Base area is " + cy2.getArea() + " Volume is " + cy2.getVolume());
    }
}

output

Radius is 1.0 Height is 1.0 Color is red Base area is 3.141592653589793 Volume is 3.141592653589793
Radius is 5.0 Height is 2.0 Color is red Base area is 78.53981633974483 Volume is 157.07963267948966
But….

```java
public class Cylinder extends Circle {
    ......
    // override the getArea() method inherited from superclass Circle
    @Override
    public double getArea() {
        return 2*Math.PI*getRadius()*height + 2*super.getArea();
    }
    // need to change the getVolume() as well
    public double getVolume() {
        return super.getArea()*height;  // use superclass' getArea()
    }
    // override the inherited toString()
    @Override
    public String toString() {
        return "Cylinder: radius = " + getRadius() + " height = " + height;
    }
}
```
Single Inheritance

• Java does **NOT** support multiple inheritance
• Multiple inheritance permits a subclass to have *more than one* direct superclasses
  – C++
• A serious drawback if the superclasses have conflicting implementation for *the same* method
More Examples on Inheritance

- **Point3D**

```java
// Define Point3D, subclass of Point
public class Point3D extends Point {
    // Private member variable
    private int z;

    // Constructors
    public Point3D() { // default no-arg constructor
        super();       // Call superclass' no-arg constructor Point()
        z = 0;
    }
    public Point3D(int x, int y, int z) {
        super(x, y); // Call superclass' Point(x, y)
        this.z = z;
    }

    // Public getter/setter for private variable
    public int getZ() {
        return z;
    }
    public void setZ(int z) {
        this.z = z;
    }

    // toString() to describe itself
    @Override
    public String toString() {
        return "(" + super.getX() + "," + super.getY() + "," + z + ")";
    }
}
```
Person and its subclasses

**Person**
- name: String
- address: String

+ Person(name: String, address: String)
+ getName(): String
+ getAddress(): String
+ setAddress(address: String): void
+ toString(): String

**Student**
- numCourses: int
- courses: String[]
- grades: int[]

+ Student(name: String, address: String)
+ toString(): String
+ addCourseGrade(course: String, grade: int): void
+ printGrades(): void
+ getAverageGrade(): double

**Teacher**
- numCourses: int
- courses: String[]

+ Teacher(name: String, address: String)
+ toString(): String
+ addCourse(course: String): boolean
+ removeCourse(course: String): boolean
Composition vs. Inheritance

- What are stacks
  - A stack is just an object that stores data following the **LIFO** principle (last in, first out). It typically has methods called push and pop.

- Diagram of a stack:
  - E
  - F
  - D
  - C
  - B
  - A
Composition vs. Inheritance

- What are stacks
  - A stack is just an object that stores data following the LIFO principle (last in, first out). It typically has methods called push and pop.
Composition vs. Inheritance

• What are stacks
  – A stack is just an object that stores data following the **LIFO** principle (last in, first out). It typically has methods called push and pop
public class Stack
{
    private ArrayList items = new ArrayList();
    private int stackSize = 0;

    public void push(Object item)
    {
        items.add(stackSize++, item);
    }

    public Object pop()
    {
        return(items.remove(--stackSize));
    }
}
public class Stack extends ArrayList
{
    private int stackSize = 0;

    public void push(Object item)
    {
        add(stackSize++, item);
    }

    public Object pop()
    {
        return(remove(--stackSize));
    }
}
Stack Class Using Inheritance

- All the public ArrayList methods (19 in all) are visible to the Stack class user! No encapsulation
  - For example-- the user could call the clear() method which would clear out the items in the stack but the clear() method would NOT reset the stackSize field to zero
  - stackSize would get out of sync with the number of items in the stack

- The class user would also be able to call lots of other methods that don't make sense for a "stack"
Composition or Inheritance

• In this Stack class example, composition is better than inheritance

• You only want to extend an existing class when your subclass needs
  – all the functionality of the superclass
  – to add its own "specialization" code
• Advantages of class inheritance
  – source code can be more concise and easier to understand
  – easily supports polymorphism

• Disadvantage of class inheritance
  – The subclass becomes dependent on the superclass implementation. This gives you less flexibility in reusing the subclass, especially if part of the inherited implementation is no longer desirable
  – The "fragile superclass" problem--a change in the superclass may "break" the subclasses
Polymorphism

- Substitutability
- Cylinder is a subclass of Circle
- We can say that Cylinder "is-a" Circle (actually, it "is-more-than-a" Circle)

Circle c1 = new Cylinder(5.0);

does not know about methods defined in the subclass Cylinder
Upcasting & Downcasting

• **Upcasting a Subclass Instance to a Superclass Reference**

Circle c1 = new Cylinder();  // Compiler checks to ensure that R-value is a subclass of L-value.
Circle c2 = new String();    // Compilation error: incompatible types

• **Downcasting a Substituted Reference to Its Original Class**

Cycle c1 = new Cylinder(5.0);  // upcast is safe
Cylinder aCylinder = (Cylinder) c1;  // downcast needs the casting operator

Circle c1 = new Circle(5);
Point p1 = new Point;

c1 = p1;  // compilation error: incompatible types (Point is not a subclass of Circle)
c1 = (Circle)p1;  // runtime error: java.lang.ClassCastException: Point cannot be casted to Circle
The "instanceof" Operator

```java
anObject instanceof aClass

Circle c1 = new Circle();
System.out.println(c1 instanceof Circle); // true

if (c1 instanceof Cirlce) { ....... }

An instance of subclass is also an instance of its superclass. For example,

Circle c1 = new Circle(5);
Cylinder cy1 = new Cylinder(5, 2);
System.out.println(c1 instanceof Circle); // true
System.out.println(c1 instanceof Cylinder); // false
System.out.println(cy1 instanceof Cylinder); // true
System.out.println(cy1 instanceof Circle); // true

Circle c2 = new Cylinder(5, 2);
System.out.println(c2 instanceof Circle); // true
System.out.println(c2 instanceof Cylinder); // true
```
// Define superclass Shape
public class Shape {
    // Private member variable
    private String color;

    // Constructor
    public Shape (String color) {
        this.color = color;
    }

    @Override
    public String toString() {
        return "Shape of color=" + color + ";";
    }

    // All shapes must have a method called getArea()
    public double getArea() {
        System.err.println("Shape unknown! Cannot compute area!");
        return 0;  // Need a return to compile the program
    }
}
// Define Rectangle, subclass of Shape
public class Rectangle extends Shape {
    // Private member variables
    private int length;
    private int width;

    // Constructor
    public Rectangle(String color, int length, int width) {
        super(color);
        this.length = length;
        this.width = width;
    }

    @Override
    public String toString() {
        return "Rectangle of length=\" + length + \" and width=\" + width + \", subclass of \" + super.toString();
    }

    @Override
    public double getArea() {
        return length * width;
    }
}
Triangle.java

```java
// Define Triangle, subclass of Shape
public class Triangle extends Shape {
    // Private member variables
    private int base;
    private int height;

    // Constructor
    public Triangle(String color, int base, int height) {
        super(color);
        this.base = base;
        this.height = height;
    }

    @Override
    public String toString() {
        return "Triangle of base=" + base + " and height=" + height + ", subclass of " + super.toString();
    }

    @Override
    public double getArea() {
        return 0.5*base*height;
    }
}
```
// A test driver program for Shape and its subclasses
public class TestShape {
    public static void main(String[] args) {
        Shape s1 = new Rectangle("red", 4, 5);
        System.out.println(s1);
        System.out.println("Area is " + s1.getArea());

        Shape s2 = new Triangle("blue", 4, 5);
        System.out.println(s2);
        System.out.println("Area is " + s2.getArea());
    }
}

public class TestShape {
    public static void main(String[] args) {
        // Constructing a Shape instance poses problem!
        Shape s3 = new Shape("green");
        System.out.println(s3);
        System.out.println("Area is " + s3.getArea());
    }
}
// A test driver program for Shape and its subclasses
public class TestShape {
    public static void main(String[] args) {
        Shape s1 = new Rectangle("red", 4, 5);
        System.out.println(s1);
        System.out.println("Area is " + s1.getArea());

        Shape s2 = new Triangle("blue", 4, 5);
        System.out.println(s2);
        System.out.println("Area is " + s2.getArea());
    }
}

public class TestShape {
    public static void main(String[] args) {
        // Constructing a Shape instance poses problem!
        Shape s3 = new Shape("green");
        System.out.println(s3);
        System.out.println("Area is " + s3.getArea());
    }
}
Abstract Classes

- abstract method
- without implementation

```java
abstract public class Shape {
    ......
    public abstract double getArea();
    public abstract void draw();
}
```
abstract public class Shape {
  // Private member variable
  private String color;

  // Constructor
  public Shape (String color) {
    this.color = color;
  }

  @Override
  public String toString() {
    return "Shape of color=\"" + color + "\"";
  }

  // All shapes must has a method called getArea()
  public abstract double getArea();
}

// Define superclass Shape
public class Shape {
  // Private member variable
  private String color;

  // Constructor
  public Shape (String color) {
    this.color = color;
  }

  @Override
  public String toString() {
    return "Shape of color=\"" + color + "\"";
  }

  // All shapes must has a method called getArea()
  public abstract double getArea() {
    System.err.println("Shape unknown! Cannot compute area!");
    return 0; // Need a return to compile the program
  }
}
Abstract class

- is incomplete in its definition
- cannot be instantiated
- you have to derive a subclass from the abstract class

```java
public class TestShape {
    public static void main(String[] args) {
        Shape s1 = new Rectangle("red", 4, 5);
        System.out.println(s1);
        System.out.println("Area is " + s1.getArea());

        Shape s2 = new Triangle("blue", 4, 5);
        System.out.println(s2);
        System.out.println("Area is " + s2.getArea());

        // Cannot create instance of an abstract class
        Shape s3 = new Shape("green"); // Compilation Error!!
    }
}
```
Another example

```java
public abstract class Animal {

    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void makeNoise();

} // end of Animal
```
Extending Abstract Classes

```java
public class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }
    public void makeNoise() {
        System.out.println("Mewwwwww");
    }
} // end of Cat
```
public class Main {

    public static void main(String[] args) {

        Cat kitty = new Cat("Paola");

        System.out.println("Want to hear my lovely " +
                          kitty.getName() + "?");
        kitty.makeNoise();

    } // main()

} // end of driver class
public class SiameseCat extends Cat {

    public SiameseCat(String name) {
        super(name);
    }

    @Override
    public void makeNoise() {
        System.out.println("Meeeeewwwwweeeeee");
    }

} // end of SiameseCat
public class Main {

    public static void main(String[] args) {

        Animal [] zoo = new Animal[5];

        zoo[0] = new Cow("Matilda");
        zoo[1] = new SiameseCat("Mussy");
        zoo[2] = new Birdie("Tweety");
        zoo[3] = new Dog("Toby");
        zoo[4] = new Cat("Frisby");

        for(int i = 0; i < zoo.length; i++)
            zoo[i].makeNoise();

    } // main()

} // end of driver class
An Even More Powerful Example

This is the program output:

```
run:
Moooooooooo
Meeeeewwweeeee
Tweet Tweet
Wof Wof Wof
Meeeeooooowwwwww
BUILD SUCCESSFUL (total time: 0 seconds)
```
Another example---maybe not good

class Animal...

private int _type;
static final int Cat = 0;
static final int Dog = 1;
static final int Cow = 2;
static final int Birdie = 3;

Animal(int type) {
    _type = type;
}

```java
Animal create(int type) {
    return new Animal(type);
}

private Animal (int type) {
    _type = type;
}

Animal makeNoise (int type) {
    switch (type) {
        case Cat:
            return "Meeeeeooooooowwwwwwww";
        case Dog:
            return "Wof Wof Wof";
        case Birdie:
            return new "Tweet Tweet";
        default:
            throw new IllegalArgumentException("Incorrect type code value");
    }
}
```
Polymorphism

• From the Greek
  – “poly” = many
  – “morph” = form, figure, silhouette

• The ability of a class method to do different things based on the object it is acting upon.
  – Example: Animal.makeNoise();
  – Dog: barks
  – Cat: mews
Three types of polymorphism

1. Method Overloading (Ad-hoc polymorphism):
   - Multiples implementations of the same method occur in the same class.
   - Each differs in the number and types of the method arguments.
   - Java invokes the closest one that matches the actual arguments being passed to the method.
     - System.out.println ( int );
     - System.out.println( char );
     - System.out.println ( String ) ;
     - System.out.println ( boolean );
Polymorphism

Parametric polymorphism:
– a function or a data type to be written generically, so that it can handle values identically without depending on their type

```java
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}

List map(Func<A,B> f, List<A> xs) {
    ...
}
```
Polymorphism

Method Overriding (Subtype polymorphism):

- Multiples implementations of the same method occur in different classes along the same hierarchy.
- A child class “overrides” the implementation of a method provided by its base class.
- Examples:
  - `Cat.makeNoise()` overrides `Animal.makeNoise()`
  - `SiameseCat.makeNoise()` overrides `Cat.makeNoise()`
Overriding vs. Overloading

• Do not confuse overriding with overloading

  ▪ **Overriding** takes place in the **subclass** – new method with **same signature**

  ▪ **Overloading** takes place in the **same class** – new method with **different signature**
The interface

• A Java interface is a 100% abstract superclass which define a set of methods its subclasses must support
• An interface contains only public abstract methods
  – methods with signature and no implementation
Example: Movable Interface and its Implementation

- Suppose that our application involves many objects that can move
- define an interface called movable
• Similar to an abstract class, an interface cannot be instantiated; because it is incomplete (the abstract methods' body is missing)
public class MovablePoint implements Movable {
    // Private member variables
    private int x, y;  // (x, y) coordinates of the point

    // Constructor
    public MovablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "Point at (" + x + "," + y + ")";
    }

    // Implement abstract methods defined in the interface Movable
    @Override
    public void moveUp() {
        y--;
    }

    @Override
    public void moveDown() {
        y++;
    }

    @Override
    public void moveLeft() {
        x--;
    }

    @Override
    public void moveRight() {
        x++;
    }
}

public class TestMovable {
    public static void main(String[] args) {
        Movable m1 = new MovablePoint(5, 5);  // upcast
        System.out.println(m1);
        m1.moveUp();
        System.out.println(m1);
        m1.moveRight();
        System.out.println(m1);
    }
}

Copyright © 2012 Addison-Wesley. All rights reserved.
Implementing Multiple interfaces

- Java supports only single inheritance
- however, can implement more than one interfaces

```java
public class Circle extends Shape implements Movable, Displayable {
    // One superclass but implement multiple interfaces
    .......
}
```
A Stack class hierarchy could also be implemented using interfaces.

```java
interface Stack {
    public void push(Object item);
    public Object pop();
}
```

Thus any class implementing the Stack interface must implement the push() and pop() methods.
This stack class uses an ArrayList for storage

```java
class ArrayListStack implements Stack {
    private int stackSize = 0;
    ArrayList items = new ArrayList();

    public void push(Object item) {
        items.add(stackSize++, item);
    }

    public Object pop() {
        return (items.remove(--stackSize));
    }
}
```
public class ArrayStack implements Stack
{
  private int stackSize = 0;
  private Object[] items = new Object[100];

  public void push(Object item)
  {
    items[stackSize++] = item;
  }

  public Object pop()
  {
    return(items[--stackSize]);
  }
}
The ArrayListStack and ArrayStack classes both implement the Stack interface and thus can be cast as Stack types.

For example, suppose we had a StackList class which is just a container for Stack objects. We could write code like so:

```java
StackList stackList = new StackList();

Stack ArrayListStack = new ArrayListStack();
stackList.add(ArrayListStack);

Stack ArrayStack = new ArrayStack();
stackList.add(ArrayStack);
```
Why Interface

- Thus by designing our stack classes using interfaces, we have **greater flexibility** than by designing by composition alone.
- As an example of the **inflexibility** of using composition alone, consider these two stack classes:

```java
public class ArrayListStack {
    private ArrayList items = new ArrayList();
    ...
}
```

```java
public class ArrayStack {
    private Object[] items = new Object[100];
    ...
}
```
Why Interface

- These classes are less flexible because there is no common type to which they can be cast.
- For example, we can't do the following

```java
StackList stackList = new StackList();

Stack arrayListStack = new ArrayListStack();
stackList.add(arrayListStack);  //error, wrong type!
```

- It is convenient to be able to cast related classes to a common type
What if we wanted to add a new method to the Stack interface which pushes several objects onto the stack at one time. We could call this method: `pushItems()`.

```java
interface Stack {
    public void push(Object item);
    public Object pop();
    public void pushItems(Object[] items);
}
```

This change will break our `ArrayListStack` and `ArrayStack` classes which implement the Stack interface!

We would be forced to rewrite these classes in order to implement the new method.
Another example

```java
public interface Animal
{
    void Sleep();
}

public interface Animal
{
    int Age{get;set;}
    void Sleep();
}

All the code that every one wrote against the first version of the interface
```
Question

How could we design a flexible object hierarchy which would allow us to add a method like `pushItems()` without breaking existing classes?
Solution

- By using class inheritance
- For example, we could create an abstract class called AbstractStack and implement the pushItems() method there
  - This wouldn't break the code in its subclasses (ArrayListStack and ArrayStack)

```java
public abstract class AbstractStack
{
    public void pushItems(Object[] items)
    {
        for (int i=0; i < items.length; i++)
            push(items[i]);
    }

    ...
```
public class ArrayStack extends AbstractStack implements Stack
{
    private int stackSize = 0;
    private Object[] items = new Object[100];

    public void push(Object item)
    {
        items[stackSize++] = item;
    }

    public Object pop()
    {
        return(items[--stackSize]);
    }
}
Implementing Multiple interfaces

```java
interface Stack {
    public void push(Object item);
    public Object pop();
}
interface StackPushItems {
    public void pushItems(Object[] items);
}
```
public class ArrayStack implements Stack, StackPushItems
{
    private int stackSize = 0;
    private Object[] items = new Object[100];

    public void push(Object item)
    {
        items[stackSize++] = item;
    }

    public Object pop()
    {
        return(items[--stackSize]);
    }

    public void pushItems(Object[] items)
    {
        for (int i=0; i < items.length; i++)
            push(items[i]);
    }
}
interface versus abstract class

```java
public abstract class Figure {
    /* because this is an abstract method the body will be blank */
    public abstract float getArea();
}

public class Circle extends Figure {
    private float radius;
    public float getArea() {
        return (3.14 * (radius * 2));
    }
}

public class Rectangle extends Figure {
    private float length, width;
    public float getArea(Figure other) {
        return length * width;
    }
}
```

**IS-A relationship**
public interface Dog
{
    public boolean Barks();
}

public class Teacher implements Dog
{
    public boolean Barks{
        // method definition here
    }
}