Java Object Oriented Programming

Course	**Java Programming**
Trainer	**Tuan Le**
Designed by	**TTC**
Last updated	**10-Apr-16**

Contents

Objectives

- Learn how classes are designed and built
- Learn about inheritance and polymorphism
- Illustrate the power of inheritance and polymorphism

Introduction

- Java is a full object-oriented programming language.
- Java supports:
  - Classes
  - Abstract Classes
  - Inner classes
  - Interfaces
- Java OOP is build on the single rooted hierarchy which means that all classes should be inherited from a single base class. In Java the name of this ultimate base class is simply “Object”.
- A class in Java represent a data type that *Encapsulates* data (attributes) and methods (behaviors) that are closely related.
- The class is the unit of Java programming.

Packages(1/2)

- Packages
  - Directory structures that organize classes and interfaces
  - Mechanism for software reuse
- Creating packages
  - Create a `public` class
    - If not `public`, can only be used by classes in same package
  - Choose a package name and add a `package` statement to source code file
  - Compile class (placed into appropriate directory)
  - Use Java standards in naming the packages.

Packages(2/2)

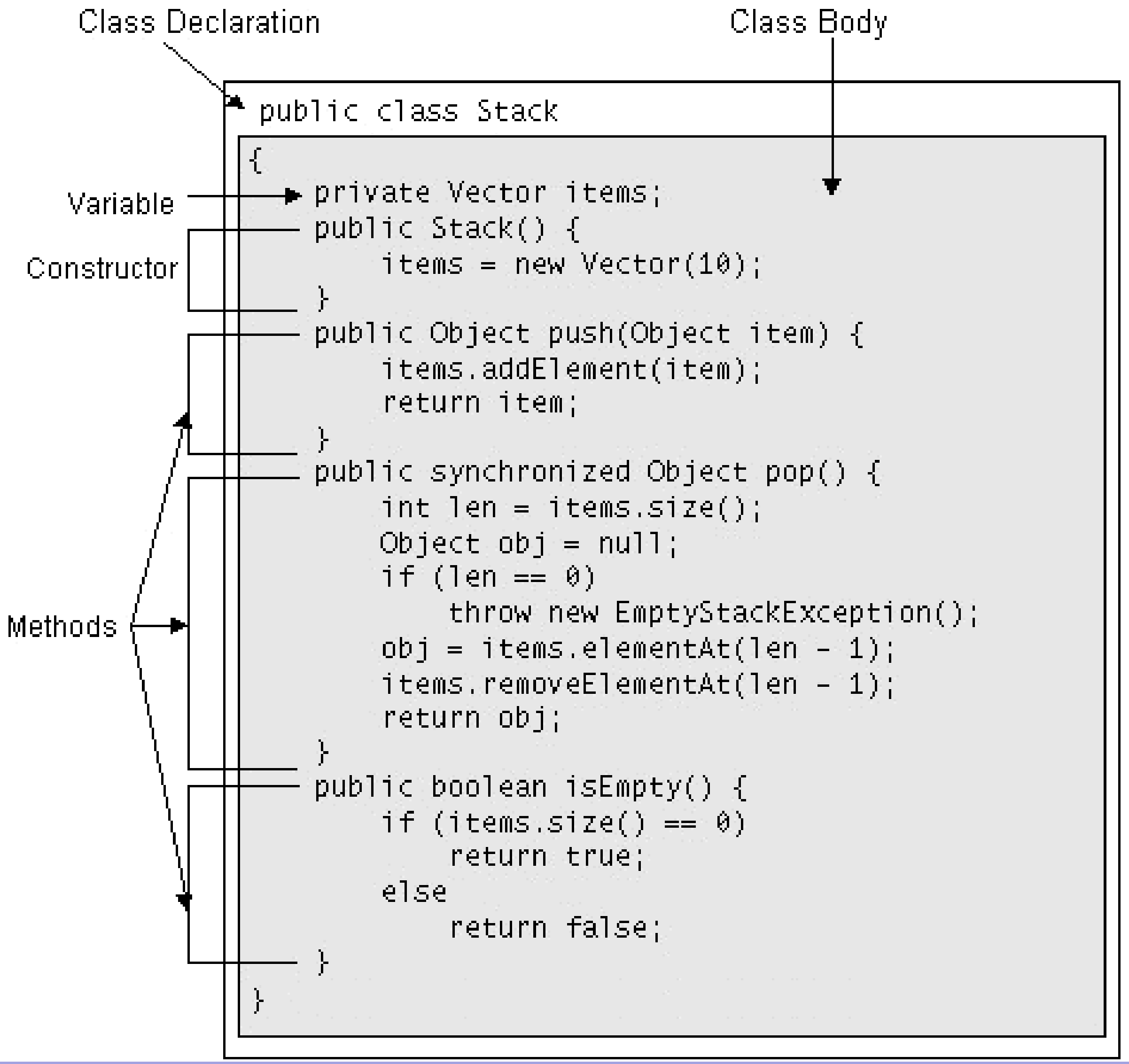
- `import`
  - Use `import` when classes are not of same package.
  - If no package specified for a class, it is put in the default package which includes compiled classes of the current directory
  - If class in same package as another, `import` not required
- Follow these steps to create a package:
  - Create a directory called `classes` inside directory `c:\jdk1.2\jre\`
  - Use the following command for compilation:

    ```
    javac -d c:\jdk1.2\jre\classes MyClasse.java
    ```
  - Once the package has been created you can use import statement to use its classes.

Java Classes(1/2)

- A class in Java is just a blueprint telling what the objects created from it will look and act like.
- Every class in Java is a subclass of the ultimate base class "Object"
- Every Class has three components:
  - Instance variables (Class Attributes).
  - Member methods (Class Behavior)
  - Constructors. (For initialization and consistency)
- Class body is delineated by braces { }

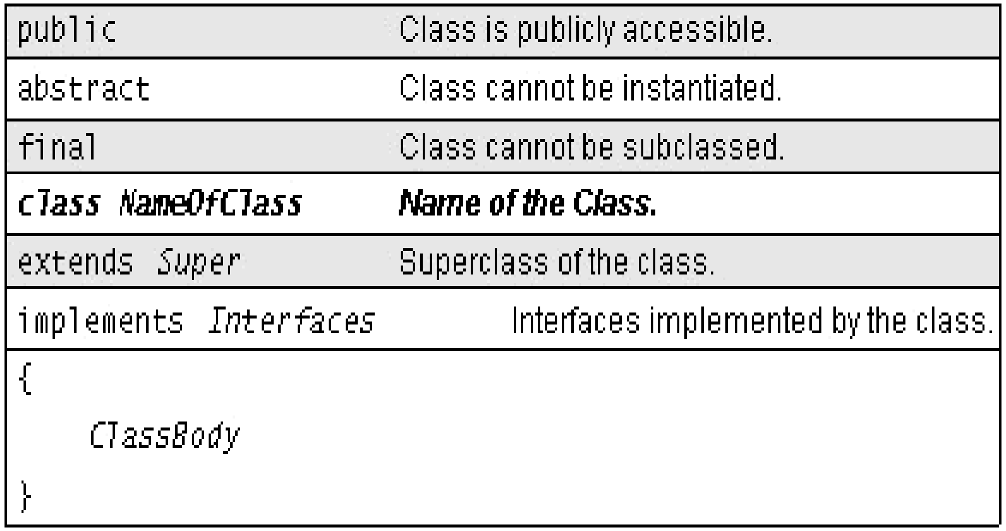
Java Classes(2/2)



```
public class Stack
{
    private Vector items;
    public Stack() {
        items = new Vector(10);
    }
    public Object push(Object item) {
        items.addElement(item);
        return item;
    }
    public synchronized Object pop() {
        int len = items.size();
        Object obj = null;
        if (len == 0)
            throw new EmptyStackException();
        obj = items.elementAt(len - 1);
        items.removeElementAt(len - 1);
        return obj;
    }
    public boolean isEmpty() {
        if (items.size() == 0)
            return true;
        else
            return false;
    }
}
```

Class Declarations(1/2)

- For class declaration, we use one or more of the following:
  - public = the class can be used by any class regardless of its package.
  - abstract = the class cannot be instantiated.
  - final = that the class cannot be subclassed.
  - class *NameOfClass* = to indicate to the compiler that this is a class declaration and that the name of the class is *NameOfClass*.
  - extends *Super* = to identify *Super* as the superclass of the class.
  - implements *Interfaces* = to declare that the class implements one or more interfaces.



public	Class is publicly accessible.
abstract	Class cannot be instantiated.
final	Class cannot be subclassed.
class NameOfClass	***Name of the Class.***
extends *Super*	Superclass of the class.
implements *Interfaces*	Interfaces implemented by the class.

```
{
    ClassBody
}
```

Class Constructors(2/2)

- All Java classes have constructors that are used to initialize a new object of that type.
- A constructor has the same name as the class. Example

```
public Stack() {
    items = new Vector(10);
}
```

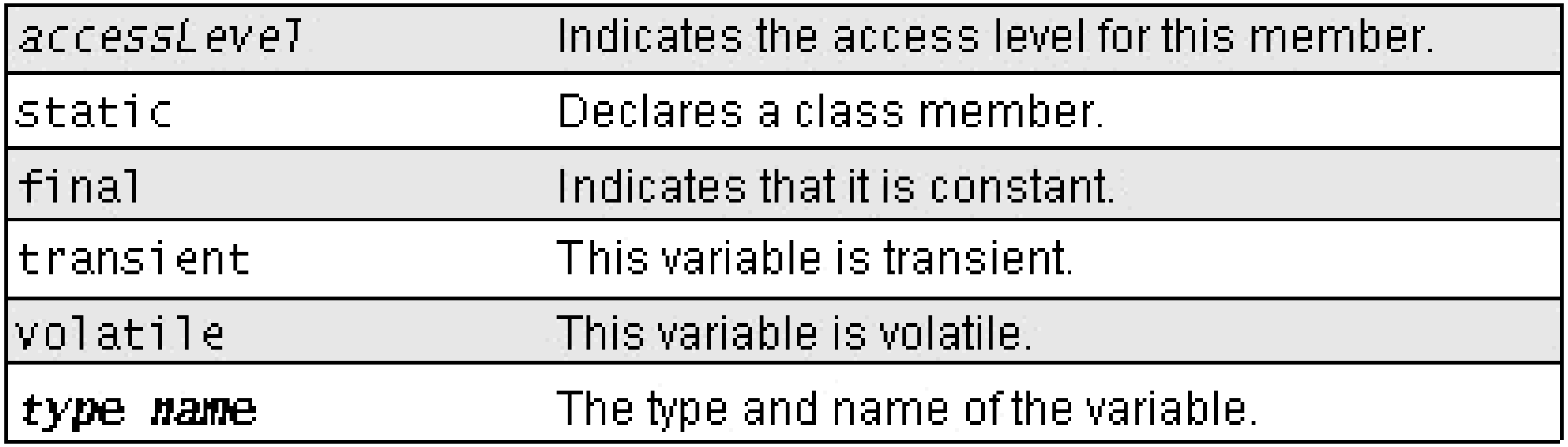
- Java supports name overloading for constructors so that a class can have any number of constructors. Example:

```
public Stack(int initialSize) {
    items = new Vector(initialSize);
}
```

- The compiler differentiates these constructors based on the number of parameters in the list and their types.
- Constructors cannot return values. There is no return type, not even `void`.

Member Variables Declaration

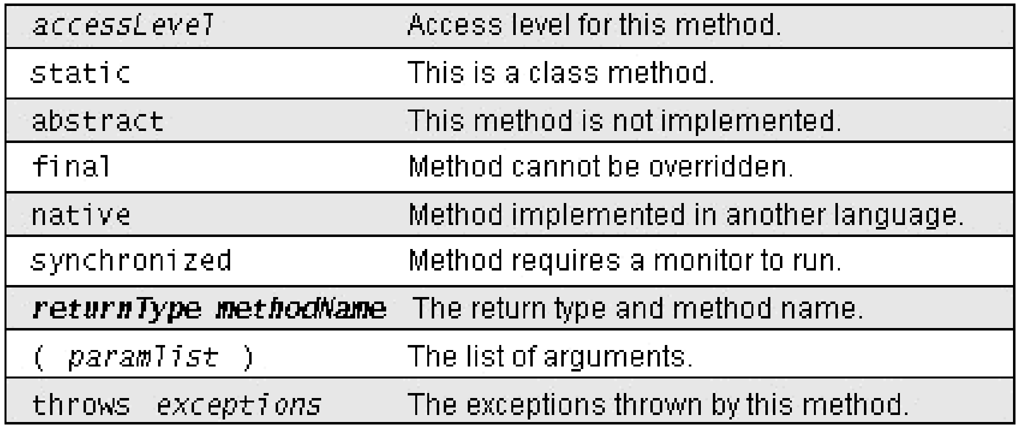
- Member variables represent the state of the object.
- They should be initialized in some way when creating the object to make sure that the object is in a consistent state.
- We use modifiers when declaring Member variables.



accessLevel	Indicates the access level for this member.
static	Declares a class member.
final	Indicates that it is constant.
transient	This variable is transient.
volatile	This variable is volatile.
type name	The type and name of the variable.

Method Declaration

- Methods are the ways through which objects communicate with each other
- A method's declaration provides a lot of information about the method to the compiler, to the runtime system, and to other classes and objects
- We use modifiers when declaring Methods.



accessLevel	Access level for this method.
static	This is a class method.
abstract	This method is not implemented.
final	Method cannot be overridden.
native	Method implemented in another language.
synchronized	Method requires a monitor to run.
returnType methodName	The return type and method name.
(*paramlist*)	The list of arguments.
throws *exceptions*	The exceptions thrown by this method.

Methods Overriding

- Overriding a method in a subclass means re-writing or modifying its code so that it acts differently from what it used to.
- Method overriding is related to a very important feature of OOP known as “polymorphism”
- ***Example***: Overriding methods `init()` or `paint()` in applets.

Controlling Access to Class Members(1/4)

- You can use access specifiers to protect both a class's variables and its methods when you declare them.
- The Java language supports four distinct access levels for member variables and methods: private, protected, public, and, if left unspecified, package.
- Private:
  - The most restrictive access level is private.
  - A private member is accessible only to the class in which it is defined. Inheritance does not apply on the private members. They are Just like secrets.
  - Use `private` keyword to create private members.

Controlling Access to Class Members (2/4)

- Protected:
  - Allows the class itself, subclasses, and all classes in the same package to access the members.
  - Use the protected access level when it's appropriate for a class's subclasses to have access to the member, but not unrelated classes. Protected members are like family secrets.
  - To declare a protected member, use the keyword protected

Controlling Access to Class Members (3/4)

- Public:
  - The easiest access specifier is **public**.
  - Any class, in any package, has access to a class's public members.
  - Declare public members only if such access cannot produce undesirable results if an outsider uses them.
  - There are no personal or family secrets here; this is for stuff you don't mind anybody else knowing.
  - To declare a public member, use the keyword **public**.

Controlling Access to Class Members (4/4)

- Package
  - The package access level is what you get if you don't explicitly set a member's access to one of the other levels.
  - This access level allows classes in the same package as your class to access the members. This level of access assumes that classes in the same package are trusted friends.
- To summarize:

Specifier	class	subclass	package	world
private	X			
protected	X	X	X	
public	X	X	X	X
package	X		X	

Class Scope

- Class scope
  - Includes Instance variables and methods
  - Class members are accessible to class methods. They can be referenced simply by name.
  - Outside scope, cannot be referenced by name
  - Visible (`public`) members accessed through a handle

    `objectReferenceName.variableName or .methodName()`
  - Static public members can be accessed through class name like:

    `Color.red, Font.PLAIN, System.out.print("");`
- Block scope
  - Variables defined in a method known only to that method
  - If variable has same name as class variable, class variable is hidden in that method.

`final` Variables, Methods, and Classes

- Declaring variables `final`
  - Indicates they cannot be modified after declaration
  - Indicate that they are constants
  - Must be initialized when declared. Can not be changed after that
  - Use all-caps identifiers. Example:

    ```
    private final int INCREMENT = 5;
    ```
- Declaring methods `final`
  - Cannot be overridden in a subclass
  - `static` and `private` methods are implicitly `final`
- Declaring classes `final`
  - Cannot be a super-class (cannot inherit from it)
  - All methods in class are implicitly `final`

Static Class Members(1/2)

- Static variables
  - Class data members are divided into two groups:
    - Instance variables: every object of the class has its own copies of them.
    - Class variables: they are allocated once for the class. All objects of class share the same variable.
  - Keyword `static` is used to create class variables.
  - `static` class variables shared among all objects of class
    - One copy for entire class to use
    - `static` class variables exist even when no objects do
  - `public static` members
    - Accessed through references or class name and dot operator
    - `MyClass.myStaticVariable`
  - `private static` members
    - Accessed through class methods.

Static Class Members(2/2)

- `static methods`
  - Can only access `static` members
  - Have no `this` reference
    - `static` variables are independent of objects
  - They can be called even if no object is created.
- Examples:
  - The main method in
    `public static void main(String []args)`
  - Method exit() in `System.exit();`
  - Method `showMessageDialog()` in `JOptionPane.showMessageDialog(. . .);`

Initializing Objects

- Initialization is a must every time an object is created out of a class.
- Java compiler will initialize all class members and creating default constructors that initialize data members as follows: `0` for primitive numeric types, `false` for `boolean`, `null` for references.
- Initialization mainly happened in the class constructors.
- A class could have more than one way (constructor) of initialization.
- Initializers are passed as arguments to constructor

Object Instantiating

- Classes only describe data types. To put then in use at runtime, objects have to be created (instantiated) from them.
- "`new`" keyword is used to instantiate an object from a class. Example:

```
public Font myFnt = new
    Font("Arial", Font.ITALIC, 12);
```

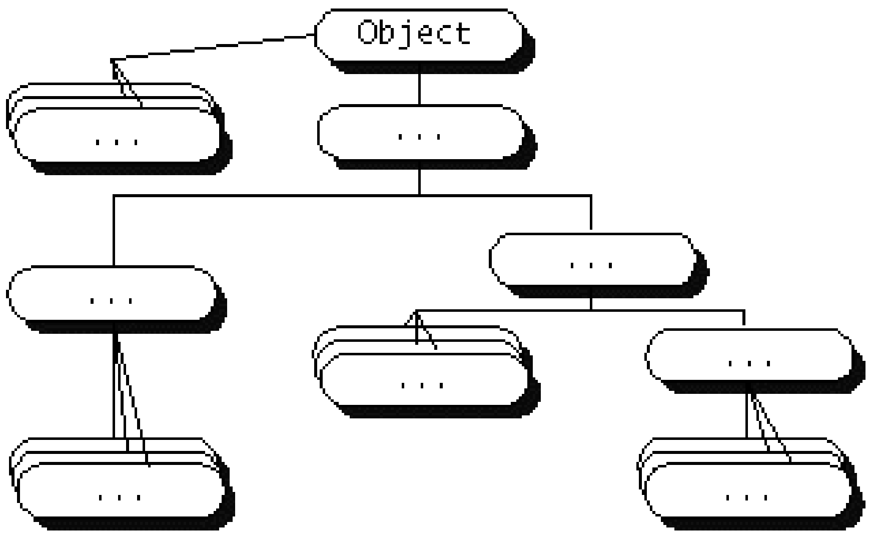
- No objects can be instantiated from Abstract classes. You can not use new here.

Using `this` Reference

- Each object has a reference to itself . It is called `this` reference
- Implicitly used to refer to instance variables and methods
- Used inside methods
  - If a parameter or a local variable has the same name as an instance variable, use `this.variableName` to explicitly refer to the instance variable. Use `variableName` to refer to the parameter
- It helps clarify the program logic.

Inheritance(1/2)

- Inheritance is a form of software reusability where
  - New classes created from existing ones
  - Subclasses absorb super-classes' attributes and behaviors, and add in their own.
- The Object class defines and implements behavior that every class in the Java system needs. The following will be the hierarchy that every class in Java will end up part of.



Inheritance(2/2)

- A Subclass is more specific than a superclass
- Every subclass can be described by its superclass, but not vice-versa
- Unlike C++, Java does not support multiple inheritance.
- To inherit from a class use keyword `extends`

```
class TwoDimensionalShape extends
  Shape
{ ... }
```

- Inheritance does also apply to Java interfaces.

Composition(1/2)

- Composition means that a class has references to other objects as members
    - These objects have to be initialized.
    - Default constructor or available constructors are used to initialize these objects.
- Composition is a powerful way of software re-use
- Composition is related to the “has a” relationship in the OOP model.

Composition(2/2)

- Using instance variables that are references to other objects. For example:

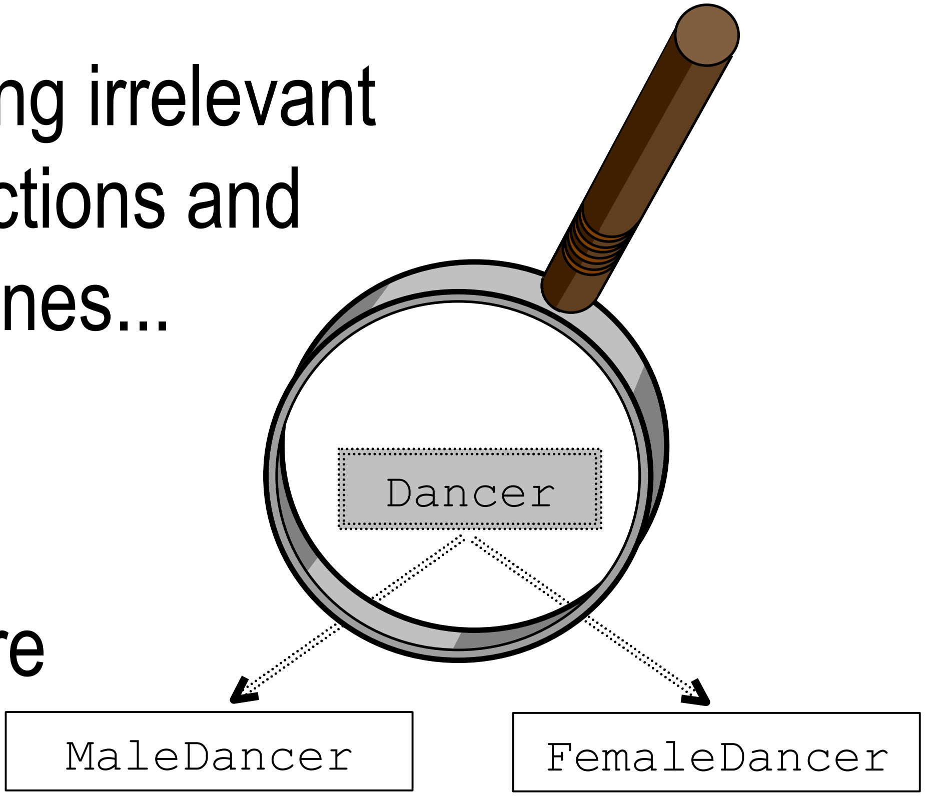
```
class Fruit {
    //...
}

class Apple {
    private Fruit fruit = new Fruit();
    //...
}
```

Abstraction(1/2)

- Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones...

- ... relevant to the given project (with an eye to future reuse in similar projects).



Abstraction(2/2)

```
public abstract class Foot
{
  private static final int footWidth = 24;

  private boolean amLeft;
  private int myX, myY;
  private int myDir;
  private boolean myWeight;

  // Constructor:
  protected Foot(String side, int x, int y, int dir)
  {
    amLeft = side.equals("left");
    myX = x;
    myY = y;
    myDir = dir;
    myWeight = true;
  }
```

All fields are `private`

Continued

Abstract methods

- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

- If a class includes abstract methods, the class itself must be declared abstract, as in:

```
public abstract class GraphicObject {
    // declare fields
    // declare non-abstract methods
    abstract void draw();
}
```

Encapsulation

- Encapsulation means that all data members (*fields*) of a class are declared private. Some methods may be private, too.
- The class interacts with other classes (called the *clients* of this class) only through the class's constructors and public methods. Constructors and public methods of a class serve as the *interface* to class's clients.

Polymorphism(1/2)

- Polymorphism ensures that the appropriate method is called for an object of a specific type when the object is disguised as a more general type
- Polymorphism is already supported in Java — all you have to do is use it properly.

Polymorphism(2/2)

- Polymorphism replaces old-fashioned use of explicit object attributes and if-else (or switch) statements, as in:

```
public abstract class Foot{
  ...
  public void draw(Graphics g){
    ...
    if (isLeft())
      drawLeft(g);
    else
      drawRight(g);
    ...
  }
}
```

Inner Classes

- Inner classes are classes defined inside other classes.
- Inner classes have access to all the members of the outer classes.
- Usually we use inner classes as helper classes of adapter classes.
- Inner classes can be anonymous (without names)
- They are used intensively to write event listeners such as ActionListener, MouseListener, KeyListener, and the like.

Java Interfaces(1/2)

- An *interface* defines a protocol of behavior that can be implemented by any class anywhere in the class hierarchy.
- An interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.
- Two elements are required in an interface declaration--the `interface` keyword and the name of the interface. The `public` access specifier indicates that the interface can be used by any class in any package.

Java Interfaces(2/2)

- An interface can inherit from one or more comma-separated superinterfaces using keyword `extends`.
- The interface body contains method declarations ; each followed by a semicolon (;). All methods declared in an interface are implicitly public and abstract.
- An interface can also contain constant declarations. All constant values defined in an interface are implicitly public, static, and final.

Implementing an Interface

- An interface defines a protocol of behavior. A class that implements an interface adheres to the protocol defined by that interface.
- To declare a class that implements an interface, include an `implements` clause in the class declaration.
- Your class can implement more than one interface (the Java platform supports multiple inheritance for interfaces. For instance,

```
public class StockApplet extends Applet
    implements StockWatcher {
    public void valueChanged(String
        tickerSymbol, double newValue){…}
    ...
}
```

Document Revision History

Date	Version	Description	Revised by
27 Jul 2011	1.0	First version	Tuan Le