Lecture 2

OO and Java

Intro
OO

• Object Oriented Analysis and Design (OOAD)
• Instead of functional AD
• Instead of spaghetti coding
• Like software version of integrated circuits. (Cox, 1986)

• object n.
  A noun - A person, place or thing – (School House Rock 1970)
  Or a concept, too.
**OO – Object Oriented**

- **Class**
  - A definition of how to construct the object, what its behavior is, what attributes it has and how to destroy the object.
  - Like a “template” or “blueprint” for creating objects of a given type
- **Object = Data + Behavior**
  - Attributes / Variables (objects, references to objects or primitive data types)
  - Methods
- **Encapsulation / Data Hiding**
  - Access to attributes and behavior can be restricted
    - Public, Private and Protected
  - Access to attributes only via methods not directly
- **Inheritance**
  - Children inherit the “characteristics” of Parents (attributes and behavior)
  - Superclass <- Subclass
- **Overriding behavior**
  - Child can change the behavior inherited from the parent by overriding the method of a the parent in the child class
- **Overloading**
  - Same method name, different parameter “signature”
- **Polymorphism**
  - Two sibling classes share a parent class, each overrides the same method in the parent, pass a parent reference for a object of either child class type, the method code that gets run is determined in runtime, dynamically.
Building things that have behavior, instead of designing behavior and then passing around things (function oriented languages).

- **OO** - `car.drive()`, `car.stop()`, `car.door[1].open()`
- **Function** - `drive(car1data)`, `stop(car1data)`

**Objects – Nouns**
- Person, Place or Thing (Concepts, too.)
- Have **Attributes** – Size, Color, Name, Location
- Have Individual **Identity**
- Have **Behavior** – do things - Verbs
OO Concepts

- Encapsulation
- “Shielding” – Information & Code Hiding
- Maintain State
- Send messages (invoke methods)
- Classes
- Inheritance – Generalization - Specialization
- Polymorphism
- Objects Have an unique Identity in runtime
- Genericity
Classes vs. Objects

- **Class definition** is a “blueprint”
- **Class** is group of related “things”
- Class definition and class are used semi-interchangeably.
- **Object** is the “building” built from the “blueprint”
Objects

• An Object is an *instantiation* of a class
• Many objects can be instantiated from a single class definition
  – *Button* is a class but a GUI might have many of the button object instances of the *Button* class
• Class also refers to the group that an object instance type belongs to
  – Objects are of a certain Class type
UML – symbology for static (class) diagrams

Class Name

instance attributes

Class Attributes

object methods

Class Methods

myObjectName:Class

inherits from (points to parent)

implements this interface

composition (exclusively “owns it”)

- contains it
- the diamond on the owner
- shows the multiplicity (0 or more)

aggregation

- references it
- knows about it, doesn’t own it
- the diamond is on the owner
- shows multiplicity (1 to 1 or more)
UML – symbology for diagrams

Visibility:
+ public
- private
# protected
~ protected

Stereotypes:
<<abstract>>
<<singleton>>
<<interface>>

A note or comment
Inheritance

- Vehicle is the **superclass** of Car
- Vehicle is the **superclass** of PickupTruck
- Vehicle is the **parent (super)** class
- Vehicle *has a* SteeringWheel
- Vehicle *has a* set of Wheel(s)
- Vehicle *has a* set of Seat(s)
- A car *is a* Vehicle
- A PickupTruck *is a* Vehicle
- A Car *has a* trunk
- A PickupTruck *has a* CargoBed
Inheritance

Maybe this is a more correct class diagram. A vehicle may have a steering wheel or a set of handle bars (a motorcycle is a vehicle).

<<interface>>
ISteeringMethod
----------------------
float Turn(float degree);

Implement A

<<abstract>>
Vehicle
------------------------------------
Wheel wheels[4];
Seat seats[1..8];

Is A

SteeringWheel
------------------------
float turn(float degree);

Has A

Is A

HandleBar
-----------------------------
float turn(float degree);

Has A

Is A

Motorcycle
----------------------------------------
Wheels wheels[2];
Seat seats[1];

Is A

Car
-------------------------------
Trunk trunk;
Wheels wheels[4];
Seat seats[4];

Has A

Is A

PickupTruck
------------------------
CargoBed bed;
Wheels[4]
Seats seats[2];
public class Car extends Vehicle {
    Trunk trunk;
    Color paintColor;
    Color interiorColor;
    Boolean convertible = false;
    static int TotalCars=0;

    public Color getPaintColor() {
        return paintColor;
    }

    public boolean setPaintColor(Color newColor) {
        paintColor = newColor;
    }
}

Java code example

attributes of object instance

attributes of class (shared by all instances)

Keyword static

methods
OO Concepts (2)

- **Encapsulation** –
  - Definition of Attributes and Behavior are contained in the same place (a class definition) and in memory as an object instance.
  - Easy development, reuse, source management and management of code in runtime.
  - Think “building blocks”

- **“Shielding”/”hiding”** –
  - Behavior and attributes are able to be made non-visible through **visibility stereotype**.
  - Limits behavioral side effects caused by code somewhere else changing your “variables”.
  - Think “integrated circuit”
  - Public, protected, (package) or private visibility.

- **Maintain State**
  - Objects are not just collections of functions.
  - The state and changes to that state are part of the object.
  - The methods implicitly operate on the attributes that are associated with that instance of the object.
  - Objects share the code that is defined in the class def but the code when executed refers to the object instance

*Objects are ALWAYS allocated on the **heap** not the **stack** and so live longer than stack based memory “thingies”.*
OO Concepts (3)

- **Send messages to communicate (invoke methods)**
  - objects communicate with each other by sending and receiving messages that contain objects.
  - In Java these are parameters to methods.
  - Messages are interrogative (question), imperative (do something) or informative (tell something).

- **Classes**
  - objects are defined by a class definition
  - defines their attributes, behavior, how to build them (constructor), parent and initial state for attributes upon instantiation (creation in memory).
  - Like a “blueprint” for an object instance.

- **Inheritance**
  - Class def can be built on other class defs.
  - The inherit both the attributes of the parent class and the behavior of the parent class (and the parents of the parents class, and so on).
OO Concepts (3)

- **Send messages to communicate (invoke methods)**
  - objects communicate with each other by sending and receiving messages that contain objects.
  - Messages are interrogative (question), imperative (do something) or informative (tell something).

- **Classes**
  - objects are defined by a class definition
  - defines their attributes, behavior, how to build them (constructor), parent and initial state for attributes upon instantiation (creation in memory).
  - Like a “blueprint” for an object instance.

- **Inheritance**
  - Class def can be built on other class defs.
  - The inherit the definition of both the attributes of the parent class and the behavior (methods) of the parent class (and the parents of the parents class, and so on).
  - Do not inherit the constructors of a parent class, though.
OO Concepts (4)

- **Overriding** – a method with the same name can be created as another, as long as they have different method signatures. The method that matches the signature is the one that gets invoked. This is determined at *compile* time.
  Ex:
  ```java
  print(int a, int b)  //parent class method
  print(int a, int b)  //child redefines it with same signature,
                      //replacing that behavior in the child class
                      //(not in the parent though)
  ```

  Also, operators (+,-,/,*) can be overloaded (in some OO languages) to provide polymorphism e.g. ‘+’ can mean addition for integers and concatenation for strings.

- **Overloading** – the method that gets invoked is determined in runtime by determining the signature of the method.
  Ex:
  ```java
  add(int a, int b)       //adds the two parameters together
  add(int a, string b)   //converts the string to an int
  ```
OO Concepts (4)

- Polymorphism

```
Polygon
----------
numberOfSides : int =0
area : float = 0.0
----------
computeArea() : float <<abstract>>
```

```
Square
------
numberOfSides=4
------
Square(float)
computeArea() : float
```

```
Triangle
------
numberOfSides=3
------
Triangle(float, float, float)
computeArea() : float
```

```
```
**OO Concepts (4)**

- **Polymorphism** – the code that gets invoked gets determined in runtime (not compile time) by the type that the object is, not by the type that the reference is.

Ex:

```java
//Triangle and Square extend Polygon class
//Assume they have beem coded already
//Polygon has a method computeArea() that is overridden in
//the two child classes Triangle and Square in different ways
Polygon aRefToAPoly;
Square mySquare = new Square(2.0);
Triangle myTriangle = new Triangle(2.0, 3.0, 4.0);

aPoly = myTriangle;
aPoly.computeArea(); //invoke method in triangle

aPoly = mySquare;
aPoly.computeArea(); //invoke method in square
```
OO Concepts (4)

• **Object ID** –
  – Each object is unique in runtime
  – multiple instances of the same class area allowed
  – have their own state in memory (instance attributes.)
  – share some state via class attributes.
  – share code defined in the class.

• **Genericity** –
  – a class definition that has the types of objects manipulated defined in *runtime*.
  – *Parameterized* class definitions or use of a base object to define the class and using object introspection in runtime to determine the type of the object.
  – C++ templates, Java 1.5 Generics
OO Visibility Modifiers

+ Public — visible to all other classes

# Protected — visible to class and children of this class

- Private — visible only within the class not visible to its children

~ Package — (java specific) visible to all the classes in the package and nobody else
Questions?

:30
UML (Unified Modeling Language) – Class Diagram

**Class Name**

Vehicle
- theSteeringWheel: Wheel
- Tire tires[4];
- Door doors[4];
- Engine engine;

- pressGasPedal(int force)
- turnWheel(int degree)
- pressBrake()
- getCurrentVelocity();

F100Pickup
- Bed bed;
- Color externalColor;
- Color tnteriorColor;
- Seat seats[2];

- pressBrake()
- isDoorOpen(int doorNum);

Camero
- AlpineStereo stereo;
- Color externalColor;
- Color tnteriorColor;
- Seat seats[4];

- pressBrake()
- isDoorOpen(int doorNum);

**Attributes**

**Methods**

**Inheritance**

**UML style**

Vs

My Java style for writing attributes/method and return types

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Attributes</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vehicle</td>
<td>tires[4];</td>
<td>pressGasPedal(int force)</td>
</tr>
<tr>
<td></td>
<td>doors[4];</td>
<td>turnWheel(int degree)</td>
</tr>
<tr>
<td></td>
<td>engine;</td>
<td>pressBrake()</td>
</tr>
<tr>
<td>F100Pickup</td>
<td>bed;</td>
<td>pressBrake()</td>
</tr>
<tr>
<td>Camero</td>
<td>externalColor;</td>
<td>pressBrake()</td>
</tr>
<tr>
<td>AlpineStereo</td>
<td>tnteriorColor;</td>
<td>pressBrake()</td>
</tr>
<tr>
<td></td>
<td>seats[4];</td>
<td>isDoorOpen(int doorNum);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright Poelman & Associates Inc. 2002 - All Rights Reserved
UML – modifiers & stereotypes

private
protected
public
unspecified
package

<<abstract>>
Vehicle

---------
- Wheel theSteeringWheel;
- Tire tires[4];
- Door doors[4];
- Engine engine;

---------
#int pressGasPedal(int force)
+ int turnWheel(int degree)
  int pressBrake();
~ int getCurrentVelocity();

---
Camero

-------------
AlpineStereo stereo;
Color externalColor;
Color tnteriorColor;
Seat seats[4];

-------------
boolean openDoor(int doorNumber);
boolean isDoorOpen(int doorNum);

---
F100Pickup

-------------
Bed bed;
Color externalColor;
Color tnteriorColor;
Seat seats[2];

-------------
boolean openDoor(int doorNumber);
boolean isDoorOpen(int doorNum);
Quick analysis rules

• **Is A** – a member of a class or child of - **inheritance**
• **Has A** – contains an object but isn’t that type of object, Usually “owns” it. - **Containment**
• **Uses** – references an object but doesn’t “own” it – **aggregation without containment**
• **Implements A** or **Act As** or **Supports A** – implements an interface

A PC **is a** electronic appliance.
A PC **has a** MotherBoard, Case, PowerSupply and CPU.
A PC **uses** (knows about) 5 different print servers.
A PC **acts as** a input device, output device, calculation machine, telephone, email station, web server, ...
UML <<Stereotypes>>

- A stereotype is a comment about a class, object, etc, in UML that adds information about the item. Often giving hints about constraints on the implementation of the item.
- <<singleton>> – only one instance of a class is allowed to be instanciated. This single instance is shared by multiple client objects.
- <<abstract>> – a non-instanciable version of a class or method.
- <<interface>> – a specification of methods that must be supported by a class that implements the interface.
- <<utility>> – a collection of utility classes.
OO Example

• A Telephone is a Class (<<abstract>>)
• It has attributes
• It implements several interfaces
• It defines behavior

*abstract means it isn’t instanciable as an object. It must be extended to become a concrete class that is instanciable into an object instance.
OO Example

• **A Telephone is a Class** (<<abstract>>)
• **attributes**
  - color, size, ...
• **It has several interfaces**
  - a soundInputDevice
  - a soundOutputDevice
  - a dialerDevice
• **The interfaces define the behavior that any children must provide implementations for:**
  - getAConnection()
  - Dial()
  - getSoundIn(Sound soundReceived)
  - putSoundOut(Sound sound);
  - changeVolume(float percentage)
  - mute(boolean muteOn);
Telephone class

```
<<abstract>>
Telephone
--------------
-Color color;
-String phoneNumOfTelephone;
-String phoneType;
-boolean inAPhoneCall=false;
--------------
getAConnection():boolean
boolean makeACall(String num);
boolean talk(String whatToSay);
String listen();
boolean hangUp();
```

```
<<interface>>
IDialerDevice
--------------
dial(String number): boolean;
```

```
<<interface>>
ISoundInputDevice
--------------
--------------
getSoundIn(Sound soundReceived)
```

```
<<interface>>
ISoundOutputDevice
--------------
--------------
putSoundOut(Sound sound);
changeVolume(float percentage)
mute(boolean muteOn);
```

UML style Vs My Java style

DeskPhone
----------
----------
CordlessPhone
----------
----------
Class vs. Object

Class

Objects (instances of a Class)
Class vs. Object

Class
(description of how to build an object)

Objects (instances of a Class)
Class vs. Object

Class

Objects (instances of a Class)
Subclasses of Telephone

• and instances of those subclasses
• OldPhone, DeskPhone, DisplayPhone, CordlessPhone and DialPhone
A subclass that adds attributes

- A PayPhone is a subclass of a Telephone
- It also has a CoinSlot and CoinReturn
PayPhone and TV classes

<<abstract>>
Telephone
-------------

<<interface>>
IDialerDevice
------------------------
dial(number)

<<interface>>
ISoundInputDevice
-------------
getSoundIn(Sound soundReceived)

<<interface>>
ISoundOutputDevice
-------------
void putSoundOut(Sound sound);
void changeVolume(float percentage);
void mute(boolean muteOn);

<<abstract>>
PayPhone
-------------
- float amountDeposited;
- CoinSlot myCoinSlot;
- CoinReturn myCoinReturn;

-------------

boolean insertCoin(float coin);
float pressCoinReturn();

<<abstract>>
TV
-------------
What about these?

• All are types of Telephone
• They may have additional attributes
• A SpeakerPhone is an <<abstract>> class that is a subclass of Telephone
Derivation – Subclass is derived from superclass

- CellPhone is derived from Telephone
- SpeakerPhone is derived from Telephone
- FaxMachine is derived from Telephone

- All are <<abstract>> classes

- MotorolaV100 is a <<concrete>> subclass of a CellPhone
Subclasses – abstract and concrete

<<abstract>> Telephone

<<abstract>> CellPhone

<<abstract>> Fax<Machine>

<<abstract>> SpeakerPhone

<<interface>> IDialerDevice

<<interface>> IVoiceInputDevice

<<interface>> IVoiceOutputDevice

MotorolaV100

myWifesCellPhone:MotorolaV100

Notation shows an instance of a class
Questions?

:60
OO Naming Conventions & Syntax

• **Class TitleCase with first letter uppercase**
  - Shoe, AnalysisEngine, ContextFactory

• **Objects camelCase with first letter lower**
  - myShoe, analysisEngine, myAnalysisEngine

• **Class Methods TitleCase with first letter upper**
  - TotalNumShoes(), GetTotalObjectInstances()

• **Instance Methods camelCase with first letter lower**
  - getShoeSize(), getContext(), isShoeBeingWorn()

• **Class Attributes …**
  - long TotalShoesInMemory;
    Url HostIDOfSingletonFactory;

• **Instance Attributes …**
  - int shoeSize, Color shoeColor, long contextID
Java

http://java.sun.com/features/2001/06/goslingduke.html

Copyright Poelman & Associates, Inc. 2002 - All Rights Reserved
Java is …

• An OO language
• Created in 1991-95! by James Gosling and others at SUN
• Originally called Green then Oak then Java
• Originally, Developed for mobile devices and set top TV boxes because C++ was too hard.

• tutorials - http://java.sun.com/docs/books/tutorial/index.html
Java is … (2)

• Platform Independent
• Object Oriented
• C++ like syntax
• Simple language structure
• Single inheritance
• Virtual Machine Based - compiler to pseudo machine code (byte codes), runtime interpreter of byte codes
Java is … (3)

- Automated **garbage collection** – objects in memory will get deleted when they are no longer utilized (as opposed to C/C++ which was manual memory management and the major source of bugs)

- **References** instead of pointers – power of C++ pointers without any of the confusion, unsafety, complexity.
  - You don’t pass objects on the stack, you pass a reference (on the stack – think of as a passing the address of a chunk of memory) to an object (on the heap)
Java is … (4)

- Everything is an **object** – have methods + attributes, have constructors, created on the heap
- Except – **int, long, byte, char, float, double** are primitive types – just data no methods, no constructor, created on stack (and heap) and passed on the **stack by value**.
- All classes are subclasses of a class called **Object**. (confusing, yes)
- Parameters in/out of method calls are by **reference** for objects (all non-primitive types)
  - Objects are pass by reference - that the method gets access to the callers object (not a copy) and can change its state permanently – pass by reference
  - primitives are pass by value - a copy is passed to the method code on the stack
Java is … (5)

• **Overriding** methods – a child can redefine a method defined in the parent.
  – The method signature has to be exactly the same as the parent.
    
    ```java
    parent.add(int a, int b)
    child.add(int a, int b)
    ```
  
  – In child objects only the child version of the method gets called. In parent objects the parent version gets called

• **Overloading** methods – an class can define multiple versions of a method, say:

  ```java
  add(int a, int b)
  add(int a, int b, int c)
  add(String A, String B)
  ```

  the correct method to invoke is determined by matching the parameters in the invocation (the method signature) at compile time.
Java is … (5)

- **Polymorphic** methods – behavior is determined in runtime.

```java
class Phone
//defines an abstract dial(String num2Dial) method

class CellPhone extends Phone
//defines a dial(String num2dial)

class DeskPhone extends Phone
//defines a dial(String num2dial)

CellPhone aCellPhone = CellPhone.new();
DeskPhone aDeskPhone = DeskPhone.new();

Phone myPhone = null;  //refernce to the parent class Phone

myPhone = aCellPhone;
myPhone.dial();  //invoke the method defined in CellPhone

myPhone = aDeskPhone;
myPhone.dial();  //invoke the method defined in DeskPhone
```

Great for building code that can be extended later on or “pluggable” in runtime. Just need new classes to be based on the parent class and use a parent class reference.
Java is … (6)

- No preprocessors. No macros.
- No constants.
- No **user defined** operator overloading.
- No global variables.
- De facto standard (not an official open standard – SUN controls it).
- Getting really fast – tests with profiling JIT JVMs have found it to be up to **2x as fast** as C++ code! (and getting faster.)
- Supported by all major vendors (except Microsoft) – IBM, Oracle, HP, Borland, …
Java History

- 1991: Oak is created for consumer electronics
- 1993: Oak used in interactive TV system dev
- 1994: Mosaic browser released
- 1995: Becomes Java – oriented to the Internet
- 1996: Netscape releases Navigator 2.0 (Java-enabled browser). Sun makes JDK freely available.
- 1997: EJB specification
- see http://java.sun.com/features/1998/05/birthday.html

Why did they pick Java as the name?
Java Technology Releases

• http://java.sun.com/products/
• J2SE – Standard Edition
• J2EE – Enterprise Edition (Servlets/EJB)
• J2ME – Micro Edition (PDAs, CellPhone)
• other releases like Jini, Jiro,
Java J2SE – Standard Edition

- I/O – Files, Streams, Pipes
- Swing – GUI toolkit
- Applets – run in a web page on the client
- RMI (Remote Method Invocation) – remotely calling other object’s methods
- Math
- JavaBeans – client side component model
- Security
- 2D Graphics
- Internationalization / Localization
J2EE – Enterprise Edition

- **Servlets** – server side active web pages written in java. Runs on web server.
- **JSP (Java Server Pages)** – HTML + java mingled together that gets compiled into servlet automatically by the web server.
- **JDBC** – database access API
- **JNDI (Java Naming and Directory Interface)** – finding other objects and database tables, etc. Looking up distributed things in a central repository.
- **EJB (Enterprise Java Beans)** – server side component model. Business logic and data logic components.
- **JavaMail** – email API
- **JMS (Java Message Service)** – message oriented middleware
- **JCA (Java Connector Architecture)** – standard adapters to legacy systems like mainframes or SAP r/3
Java Technologies

copyright SUN © 2002

Copyright Poelman & Associates, Inc. 2002 - All Rights Reserved
Java Technologies

copyright SUN © 2002
J2ME – Micro Edition

• Targeted at consumer electronics and embedded devices.
• Virtual machine and a minimal set of core libraries
• Extending the capabilities of the minimal configuration by adding additional libraries
Java API Relationships

copyright SUN © 2002

Copyright Poelman & Associates, Inc. 2002 - All Rights Reserved
Java API Relationships

copyright SUN © 2002
Questions?

:90
JVM – Java Virtual Machine

• Software layer on top of a physical processor
• abstraction
• Standard instruction set (bytecodes)
• runtime translation to architecture/processor specific operands (machine code).
• JVMs for specific platform
• Win/x86, Solaris/SPARC, Linux/x86, IBM mainframe, cellphone, PDA
• On windows invoked by running java.exe
JVM

- **Bytecodes** are **verified**- checked to insure no illegal operations, pointer based bugs, etc. – bytecode verifier
- Interpreted in runtime
- Or **JIT** (just in time) Compiled to machine specific instructions. Faster execution.
- Or in runtime, code can be **profiled** and the most frequently executed code can be precompiled and cached in memory to improve performance with minimum memory utilization. (HotSpot from SUN) – can’t do that in C++!
Simple Java Program

```java
public class DemoJavaProgram {
    public static void main(String[] cmdlineArgs) {
        System.out.println("Hello World");
    }
}
```

compile program
run .class in JVM
the Java compiler
javac.exe

- Converts java code into a compiled class file (java byte codes in a .class file)
- JVM executes byte codes
- Byte codes are for a “pseudo” or “virtual” microprocessor.
- Translated into the machine specific/architecture specific instructions on the fly by the JVM.
The java JVM

java.exe

• To run a program in the JVM
• Use java.exe <classname>
• That class must have a method named “main()” with the exact signature
  
  ```
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
  ```

• Has to be a public static void method.
• Parameters accepted are commandline arguments (if any)
• JVM jumps to that “main” method and executes from there.
Using an IDE to create

Create a new NetBeans object, for example:

- A Swing Form
- An Applet
- A JavaBean
- A Java Server Page

Or, select from the complete set of object templates.

Tip of the Day

You can use VCS Groups to keep track of which version-controlled files have changed in your working directory by automatically adding all modified files to the default group.

Choose VCS Groups from the Versioning menu in the main window. Right-click the VCS Groups node and choose Properties. Then set
Select “main”
name it “MyFirstClass”
click next, next, next, ...
Generated code
Add line and then press execute

```java
    System.out.println("Hello Class.");
```

Java Packages

- A hierarchical way of storing related classes.
- `package` statement tells compiler where to associate the class with.
- `import` statement tells compiler where to look for class definitions. Avoids having to fully qualify every thing e.g. use `String` instead of `java.lang.String`
Looking up java classes at
http://java.sun.com/j2se/1.4/docs/api/

```java
java.util

Class Date

java.lang.Object
    \_java.util.Date

All Implemented Interfaces:
    Cloneable, Comparable, Serializable

Direct Known Subclasses:
    Date, Time, Timestamp

public class Date
    extends Object
    implements Serializable, Cloneable, Comparable

The class Date represents a specific instant in time, with millisecond precision.

Prior to JDK 1.1, the class Date had two additional functions. It allowed the interpretation of dates as year, month, day, hour,
minute, and second values. It also allowed the formatting and parsing of date strings. Unfortunately, the API for these functions was
not amenable to internationalization. As of JDK 1.1, the Calendar class should be used to convert between dates and time fields
and the DateFormat class should be used to format and parse date strings. The corresponding methods in Date are deprecated.
```
Java Coding Conventions

• Class name Title Case with first letter Caps.
  - AutoPilotSystem
  - Order
  - Book
  - BookShelf

• Method and Attribute names in camel case with first letter lower case.
  - String getPilotName(); //method
  - String pilotName;     //attribute
  - String socialSecurityNumber;
  - Int    age;
  - Book   myBook;
Sample Class

package GVOnlineCommandline;

public class Book
{
    int              numOfPages  =0;
    String           title;
    String           authorName;
    String           pages;
    public void book()
    { //constructor
    }

    public void setTitle(String newTitle)
    {
        title=newTitle;
    }
    public String getTitle()
    {
        return (title);
    }
}

Copyright Poelman & Associates,
Inc. 2002 - All Rights Reserved
Javadoc

• Builds a HTML file that extracts comments and class structure from your code to produce documentation

• Special “commands” are placed in the documentation comments. These get extracted from your code and formatted into the javadoc.

• Comments must immediately precede the immediately before class, interface, constructor, method, or field declarations

• Can include HTML fragments
Example

/** *
 * A class representing something.
 * This is a <b>doc</b> comment.
 * For example:
 * <pre>
 * Window win = new Window(parent);
 * win.show();
 * </pre>
 * @author Logan Poelman
 * @version %I%, %G%
 * @see java.awt.BaseWindow
 * @see java.awt.Button
 */

class MyClass extends MyParentClass
{
    ...
}
javadoc syntax

* @author (classes and interfaces only, required)
* @version (classes and interfaces only, required)
  (see footnote 1)
* @param (methods and constructors only)
* @return (methods only)
* @exception
* @throws (is a synonym added in Javadoc 1.2)
* @see
* @since
* @serial (or @serialField or @serialData)
* @deprecated (see How and When To Deprecate APIs)
GVBooksOnline.GVBooksOnline

Class MainAppFrame

java.lang.Object
    |  
    |  ---java.awt.Component
    |   |  
    |   |  ---java.awt.Container
    |   |  |  
    |   |  ---java.awt.Window
    |   |  |  |  
    |   |  ---java.awt.Frame
    |   |  |  |  |  
    |   |  ---javax.swing.JFrame
    |   |  |  |  |  |  
    |   |  ---GVBooksOnline.GVBooksOnline.MainAppFrame

All Implemented Interfaces:
    javax.accessibility.Accessible, java.awt.image.ImageObserver, java.awt.MenuContainer, javax.swing.RootPaneContainer,
    java.io.Serializable, javax.swing.WindowConstants

Deprecated. (see How and When To Deprecate APIs)

public class MainAppFrame
extends javax.swing.JFrame
running the javadoc processor

```
C:\GUBooksOnline\GUBooksOnline\GUBooksOnline>javadoc *.java
Loading source file AddCustomerDialog.java...
Loading source file MainAppFrame.java...
Constructing Javadoc information...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
javadoc: warning - GUBooksOnline.GUBooksOnline.MainAppFrame: <methods and constructors only> tag not on method.
javadoc: warning - GUBooksOnline.GUBooksOnline.MainAppFrame: @throws is a synonym added in Javadoc 1.2, tag not on method.
javadoc: warning - GUBooksOnline.GUBooksOnline.MainAppFrame: @see tag has no arguments.
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating index.html...
Generating packages.html...
Generating GUBooksOnline\GUBooksOnline\AddCustomerDialog.html...
Generating GUBooksOnline\GUBooksOnline\MainAppFrame.html...
Tag @see: Malformed:
Generating serialized-form.html...
Generating package-list...
Generating help-doc.html...
Generating stylesheet.css...
3 warnings

C:\GUBooksOnline\GUBooksOnline\GUBooksOnline>
```
Questions?

:110
Summary

• OO
  – Encapsulation
  – Polymorphism
  – Inheritance
  – Encapsulation
  – Classes
  – “Shielding”
  – Maintain State
  – Send messages
  – Objects Identity
  – Genericity

• Java
  – OO
  – Platform Independent
  – Compiled and Interpreted
  – JVM
  – Garbage Collection
  – J2SE, J2EE, J2ME
  – NetBeans IDE