Questions or Comments

• Lecture?
• Book?
• Tools?
Lecture 2

Intro to OO (Object Orientation)
and the Java language
OO

- Object Oriented Analysis and Design (OOAD)
  - Instead of functional AD (C, Cobol, Fortran, Basic, Assembly, …)
  - Instead of spaghetti coding
- OO Dates from
- Like software equivalent of “integrated circuits”
  - (Cox, 1986)
- **object** *n.*
  A noun - A person, place or thing
  (School House Rock 1970)
  Or a concept, too. (Ex: money, good will, intelligence, love, …)
Integrated Circuit (IC) Metaphor

- ICs that have well defined interfaces but encapsulate the implementation of how they do that functionality – you can’t see inside the IC!
- You connect them together and can easily replace them – modularity
- ICs send signals (messages) to each other through the connections between their interfaces
- Think about CPUs – you can substitute a AMD for an Intel because they use standard voltages, signals and clock signals
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[state] + Behavior the relates to that data**
  - Instance of a Class
  - Attributes / Variables (objects, references to objects or primitive data types)
  - Methods – Implement Behavior
- **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 (Parent) vs. Subclass (Child)
- **Overriding behavior**
  - Child can change the behavior inherited from the parent by overriding the method of a the parent in the child class. Only the child instances will use the newly defined behavior
- **Overloading**
  - Same method name, different parameter “signature”
  - Add(int,int), add(float,float),add(img,img)
- **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, based on type of the child object’s class. – *I’ll explain later*
Building “things” that have “behavior”, instead of designing behavior and then passing around things (like function oriented languages do).

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

**Objects – Nouns**
- **Nouns** - Person, Place or Thing (Concepts, too.)
- Have **Behavior** – they do things – Verbs - methods
- Have **Attributes** – Size, Color, Name, Location
- Have Individual **Identity**
- Can **contain** of the objects (generally of a different type)
Basic OO Concepts

- **Classes** “blueprint” for object instantiation, structure (attributes) and behavior (methods & constructors)
- **Inheritance** – Classes can inherit behavior and attributes. Generalization vs. Specialization
- **Encapsulation / “Shielding”** – modular, self contained units that hide the implementation from the outside
  - Information/state & Code Hiding
- **Send messages** (invoke methods) to other objects
- Object Maintain **State between** method calls
- Objects Have an unique **Identity** in runtime
- **Polymorphism** – behavior can change depending on the type of the object the message
- **Genericity** – <I’ll explain later>
Classes vs. Objects

- **Class definition** is a “blueprint” or “schematic”
- **Class** is definition of group of related “things”
- Terms “Class Definition” and “Class” are used seminterchangably.
- **Object** is the “building” built from the “blueprint” (class) or “IC”
- Objects are instantiated by following the class def
  - Instance of an object
OO Concepts - 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 creation.
- IC Analogy – circuit schematic
OO Concepts - 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 button objects of the *Button* class
  – From the *Class Invoice*, you might have 50 *invoice* objects
• Class also refers to the group that an object instance *type* belongs to
  – Objects are of a certain Class *type*
Inheritance

<<interface>>
ISteerable

float Turn(float degree);

SteeringWheel

turn(float degree):float

Is A

HandleBar

float turn(float degree);

Is A

Implements A

<<abstract>>
Vehicle

Wheel wheels[];
Seat seats[];

Is A

Car

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

Has A

Motorcycle

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

Is A

PickupTruck

CargoBed bed;
Wheels[4]
Seats seats[2];

Has A

<<abstract>>

Note: UML allows two styles for writing methods and attributes. The java style and the more general way.)
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
public class Car extends Vehicle {
    Trunk trunk;
    Color paintColor;
    Color interiorColor;
    Boolean convertible = false;
    SteeringWheel steeringWheel;
    static int TotalCars = 0;

    public Color getPaintColor() {
        return paintColor;
    } // windows style braces

    public boolean setPaintColor(Color newColor) {
        paintColor = newColor;
        return true;
    } // unix style braces

    public static int getTotalCars() {
        return TotalCars;
    }

    public static IncTotalCars() {
        TotalCars++;}

    public static IncTotalCars() {
        TotalCars++;}

}
OO Concepts - Encapsulation

- Definition of Attributes and Behavior are contained in the same place (a class definition) and in memory as an object instance.
- Ease of development, reuse, source management and management of code in runtime.
- Think “building blocks
  - all the components compose a single unit
  - Ex: a motherboard
OO Concepts - “Shielding”/”hiding”

• <<Somewhat related to Encapsulation>>
• Behavior and attributes are able to be made non-visible through **visibility stereotype**.
  – Public, protected, (package) or private visibility.
• Can’t access attributes except through **methods** on object
• Limits **behavioral side effects** caused by code somewhere else changing your “variables” without you knowing it in the code that “owns” them.
• Allows changing the implementation without “**breaking**” the code that uses your class.
• Analogy: “integrated circuit”
  – can’t see the components inside the case, only the external pins (**interface**) like the methods on an object

Vehicle

- Wheel wheels[4];
- SteeringWheel steering;
- Seat seats[4];
- currentSpeed float;

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

+ turnWheet(float);
+ setSpeed(float);
OO Concepts - Inheritance

- Class def can be built on other class defs (inherits from).
- 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).
- Do not inherit the constructors, generally in OO languages
- SubClass “Is A” Parent Class

```
PickupTruck
-------------
Wheel wheels[4];
Seat seats[2];

Is A

FordF150
---------
CargoBed bed;
TowHitch towHitch;
```
OO Concepts – Objects Maintain State

- Objects are **not** just collections of functions.
- The **state (attributes)** and **changes (method)** to that state that are part of that object.
- The **methods implicitly operate** on the attributes that are associated with that instance of the object. (the **this** reference in java)
- Object **share the code** that is defined in the class def but the code when executed refers to the object instance variables
Object send messages to communicate

- Objects communicate with each other by sending and receiving messages that contain objects
  - In Java, they invoke methods on other object to communicate with them. These are parameters to methods and returned objects

- Messages (methods) are:
  - **interrogative** (ask a question)
  - **imperative** (please do something)
  - or **informative** (let me tell you something).

- IC Analogy – ICs communicate via signals
OO Concepts - Object ID

- Each object is **unique** in runtime
- Multiple instances of the same class are allowed
- Have **their own** individual state in memory
  - Each object has its via instance attributes (variables)
- **Share** class state
  - Via class (static) attributes.
- **Share** code defined in the class.
**OO Concepts - Overriding**

- A method in the child class with the **same name** as the parent method. Thus the child behavior overrides the parents definition of that behavior. They **must have the same method signatures** (name and parameter types, not return type or the actual names of the parameters). This is determined at **compile** time.

- **Ex:**

  ```java
  print(int a, int b) //parent class method - prints in RED
  print(int x, int b) //child redefines it with same signature, //replacing that behavior in the child class //(not in the parent though) //prints in GREEN
  ```

  **Note the signatures are the same in the parent and the child even though parameter names are different.**

OO Concepts - Overriding

• Also, operators (+,-,/,*) can be overloaded in some OO languages to provide polymorphism
  • Ex: ‘+’ can mean addition for integers and concatenation for strings.
  • **User definition** of overloaded operators not **allowed** in Java
  • Java does overload the “+”, “+=“ operator for numbers and strings
    − 1.0 + “ Dollars” -> ”1.0 dollars
OO Concepts - Overloading

- In a class, multiple methods have the same name but different parameter signatures.
- The method that gets invoked is determined in compile time by determining the signature of the method.

Ex:

```java
print(float)   { }
print(int)     { }
print(String)  { }

print("Hello world");
print(1.00);   
print(300)    
```

RedPrinter
-------------
print(int)
print(float)
print(String)
OO Concepts - 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 been 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 - Polymorphism

Polygon

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

Square

- numberOfSides = 4
- computeArea() : float

Triangle

- numberOfSides = 3
- computeArea() : float

Triangle(float, float, float)

Square(float)
OO Concepts - Genericity

- 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.
- “Metaclass” – a blueprint for a class
- Java 1.5 Generics, C++ templates,
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
  - Attributes:
    - theSteeringWheel: Wheel
    - Tire tires[4];
    - Door doors[4];
    - Engine engine;
  - Methods:
    - int pressGasPedal(int force)
    - int turnWheel(int degree)
    - int pressBrake();
    - int getCurrentVelocity();

- **Class Name**: F100Pickup
  - Attributes:
    - Bed bed;
    - Color externalColor;
    - Color interiorColor;
    - Seat seats[2];
  - Methods:
    - int pressBrake();
    - boolean isDoorOpen(int doorNum);
UML – modifiers & stereotypes

<<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);

F150Pickup
------------
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
- **Able** or **Ability to** - 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, …
A Keyboard is Able to input data –
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>>
IDialerDevice

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

<<interface>>
ISoundInputDevice

getSoundIn(Sound soundReceived)

<<interface>>
ISoundOutputDevice

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

putSoundOut(Sound sound);
changeVolume(float percentage)
mute(boolean muteOn);

UML style
Vs
My Java style

DeskPhone

CordlessPhone
PayPhone and TV classes

<<abstract>>
Telephone
-------------------------
float amountDeposited;
CoinSlot myCoinSlot;
CoinReturn myCoinReturn;
-------------------------
boolean insertCoin(float coin);

<<abstract>>
PayPhone
-------------------------

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

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

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

<<abstract>>
TV
-------------------------
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
- **Blackberry is a** <<concrete>> subclass of a **CellPhone**
Subclasses – abstract and concrete

Telephone

FaxMachine

IAbilityToDial

IAbilityToInputSound

IAbilityToOutputSound

IAbilityToConnectToWiring

CellPhone

Blackberry

SpeakerPhone

myBlackberry

Notation shows an instance of a class
Questions?

:60
Java 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

- **Interfaces ...**
  - IRunnable, ISerializable, IAbilityToSampleAudio, IResizable
Java

The Duke logo™ is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries.

http://java.sun.com/features/2001/06/goslingduke.html
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 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 is … (2)

- **Platform Independent**
  - Java code runs *unchanged* on many platforms
  - Win, Unix, Mac, PDA, Phones, Routers, Cable Boxes, …

- **Object Oriented**
- **C / C++ like syntax**
- **Simple language structure**
- **Single inheritance**

- **Virtual Machine Based**
  - compiled to pseudo machine code (byte codes)
  - runtime interpreter of byte codes JVM
  - JVM (Java Virtual Machine) written for each platform supported
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)
  – the memory gets reused (goes back as free in the heap)

• **References** instead of pointers – power of C++ pointers without any of the confusion, unsafety, complexity.
  – In C/C++ you could do pointer ‘arithmetic” and access things you shouldn’t – source of many viruses e.g. buffer overflows
  – You don’t pass objects on the stack, you pass a reference to that object on the stack to an object on the heap (think of it as a passing the address of a chunk of memory)
Java is ... (4)

- Everything is an **object**
  - have methods, attributes, constructors and are **created on the heap**
- Except **primitives**
  - int, long, byte, char, float, double are primitive types
  - just data no methods, no constructor, created on stack (and heap if an attribute of an object) 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
  - Objects are **pass by reference**
    - the method that gets access to the callers object (not a copy) by the reference and can change that objects state permanently
  - 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
class parent {
    void add(int a, int b) {
    }
}
class child {
    void add(int a, int b) {
    }
}
```
  - In child objects, 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 that have the same name:

  ```java
  class parent {
    void add(int a, int b) {
    }
    void add(int a, int b, int c) {
    }
    void 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;  //reference 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. (not really true anymore)
- No constants. ( Enums in Java 5.0)
- 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 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
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
Java API Relationships

copyright SUN © 2002
Questions?

:90
JVM – Java Virtual Machine

- **Software layer** on top of a physical processor
- **abstraction** of a microprocessor
- 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** Compiled (**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.
  - Ex: HotSpot from SUN
  - can’t do that in C++ or statically compiled languages!
Simple Java Program

```java
public class DemoJavaProgram {
    public static void main(String[] args) {
        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”

Using this template, you can create a new Java class with a main method permitting it to be run as a console application.

If you want to design a visual application, you might prefer to use the JFrame template under SwingForms, or an application skeleton under Sample Forms.
name it “MyFirstClass”
click next, next, next, ...
Generated code
Add line and then press execute
Java Packages

• A way of storing related classes.
• **package** statement tells compiler where to find the associated the class files.
• Looks like a hierarchical space but actually its **flat** – I’ll explain later
• **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/1.5/docs/api/
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;
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);
    }
}

Sample Class
Javadoc

- Builds a HTML file that extracts comments and class structure from your code to produce documentation
- Special “commands” are place in the documentation comments. These get extracted from you 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)
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
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