UCLA PIC 20A
Java Programming

- **Instructor:** Ivo Dinov,
  Asst. Prof. In Statistics, Neurology and Program in Computing
- **Teaching Assistant:** Yon Seo Kim, PIC

University of California, Los Angeles, Summer 2002
http://www.stat.ucla.edu/~dinov/

Chapter 2 – Object Oriented Programming

- What Is an Object?
- What Is a Message?
- What Is a Class?
- Objects versus Classes
- What Is Inheritance?
- What Is an Interface?
- Coding – objects, classes, inheritance, interfaces …

Objects

- An object is a software bundle of variables and related methods.
- Real-world objects: your dog, desk, your TV set, your bicycle.
- Objects two characteristics: ***state*** and ***behavior***.
- Ex., dogs have state (name, color, breed, hungry) and behavior (barking, fetching, and wagging tail).
- Ex., bicycles have state (current gear, current pedal cadence, two wheels, number of gears) and behavior (braking, accelerating, slowing down, changing gears).

Objects

- **(Software) Objects** are modeled after real-world objects in that they too have state and behavior.
- A software object maintains its state in one or more ***variables***. A variable is an item of data named by an identifier. A software object implements its behavior with ***methods***. A method is a function (subroutine) associated with an object.
- **Definition:** An object is a software bundle of variables and related methods.

Objects – guiding principles

- **Modularity:** The source code for **one object** can be written and maintained independently of the source code for **other objects**. Also, an object can be easily passed around in the system. You can give your bicycle to someone else, and it will still work.
- **Information hiding:** An object has a **public interface** that other objects can use to communicate with it. An object can maintain private information and methods that can be changed without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your bike to use it.

Messages

- A single object alone is generally not very useful. Instead, an object usually appears as a component of a larger program or application that contains many other objects.
- Your bicycle hanging from a hook in the garage is just a bunch of titanium alloy and rubber; by itself, the bicycle is incapable of any activity. The bicycle is useful only when another object (you) interacts with it (pedal).
- Software objects interact and communicate with each other by sending ***messages*** to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B.

Messages

- Sometimes, the receiving object needs more information so that it knows exactly what to do;
- E.x., when you want to change gears on your bicycle, you have to indicate which gear you want. This information is passed along with the message as ***parameters***.
- Components that comprise a message:
  - The object to which the message is addressed (**Bicycle**)
  - The name of the method to perform (**changeGear**)
  - Any parameters needed by the method (**lowerGear**)

Note the syntax

Messages

- Messages provide **two important benefits**.
  - An object's behavior is expressed through its methods, so (aside from direct variable access) message passing supports all possible interactions between objects.
  - Objects don't need to be in the same process or even on the same machine to send and receive messages back and forth to each other.
- Ex, `//since this view will also be changed, it will fire a property change to all the listeners`
  `firePropertyChange("ColorMap", null, _colorModel);`

C:\Ivo.dir\LONI_Viz\LONI_Viz_MAP_demo\src.dir\edu\ucla\loni\LONI_Viz\VolumeSliceView.java

Classes

- In the real world, you often have many objects of the same kind. For example, your bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that **your bicycle object is an instance of the class of objects known as bicycles**. Bicycles have some state (current gear, current size, two wheels) and behavior (change gears, brake) in common.
- However, each bicycle's state is independent of and can be different from that of other bicycles. When building bicycles, manufacturers take advantage of the fact that bicycles share characteristics, building many bicycles from the same blueprint. It would be very inefficient to produce a new blueprint for every individual bicycle manufactured.

Classes

- In object-oriented software, it's also possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. Like the bicycle manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects.
- A software blueprint for objects is called a ***class***.

Class vs. Instance Variables

- ***Class variables***: contain information that is shared by all instances of the class. For example, suppose that all bicycles had the same number of wheels (2 or 3). In this case, defining an instance variable to hold the number of wheels is inefficient; each instance would have its own copy of the variable, but the value would be the same for every instance. In such situations, you can define a class variable that contains the number of gears. All instances share this variable. If one object changes the variable, it changes for all other objects of that type. A class can also declare ***class methods***.
- **You can invoke a class method directly from the class, whereas you must invoke instance methods on a particular instance.** (E.g., System.out.println("Class Method");)

Objects vs. Classes

- Objects and Classes probably look very similar …
- A blue-print of a bicycle is not a bicycle.
- The term **object** is sometimes used to refer to both classes and instances.

Inheritance

- Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. Even if you don't know what a **pennyfarthing** is, if I told you it was a bicycle, you would know that it had two wheels, handle bars, and pedals.
- Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. Mountain bikes, racing bikes, and tandems are all ***sub-classes*** of the bicycle **(super-)class**.

Inheritance

- Each subclass ***inherits*** state (in the form of variable declarations) from the tandems share some states: cadence, speed, and size, etc. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and, each subclass inherits methods from the superclass. All bikes share some behaviors: braking and changing pedaling speed, for example.

Inheritance – how are sub- & super-classes different?

- Subclasses are not limited to the state and behaviors provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.
- Subclasses can also ***override*** inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the "change gears" method so that the rider could use those new gears.

Inheritance – how are sub- & super-classes different?

- You are not limited to just one layer of inheritance. Inheritance forms a tree, or class *hierarch*y,
- In general, the farther DOWN in the hierarchy a class appears, the more specialized its behavior. And the higher UP in the hierarchy a class is the more general it is!
- The **Object** class is at the top of class hierarchy, and each class is its descendant (directly or indirectly).
- For example, all classes inherit **Object**'s **toString()** method, which returns a string representation of the object.

Interfaces

- An **interface** is a device or a system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the
- English language is an interface between two people, and the protocol of behavior enforced in the military is the interface between people of different ranks.
- In Java, an **interface** is a device that unrelated objects use to interact with each other. An interface is probably most analogous to a protocol (an agreement on behavior).
- **Definition:** An interface is a device that unrelated objects use to interact with each other. An object can implement multiple interfaces.

Interfaces

- The bicycle class and its class hierarchy define what a bicycle can and cannot do in terms of its bicycleness. But bicycles interact with the world on other terms. For example, a bicycle in a store could be managed by an inventory program. An inventory program doesn't care what class of items it manages, as long as each item provides certain information, such as price and tracking number.
- Instead of forcing class relationships on otherwise unrelated items, the inventory program sets up a protocol of communication. **The inventory interface would define, but not implement, methods that set and get** the retail price, assign a tracking number, and so on.

Interfaces

- Interfaces are useful for the following reasons:
  - Capturing similarities among unrelated classes without artificially forcing a class relationship
  - Declaring methods that one or more classes are expected to implement
  - Revealing an object's programming interface without revealing its class

Coding using these concepts

- Ex., ClickMe.java class (C:\Ivo.dir\UCLA_Classes\Summer2002\PIC20A\Examples\JavaTutorialExamples\java\concepts\example-1dot1\ClickMe.java)
- **Local Objects**: ClickMe, Spot, the three colors used in the applet (black, white, and red); an event object represents the user action of clicking the mouse, and so on.
- Classes: ClickMe & Spot

Coding using these concepts

```
public class Spot {
        //instance variables
    public int size;
    public int x,y;

        //constructor
    public Spot(int intSize)    {
        size =intSize;
        x =-1;
        y =-1;
    }
}
```


Coding using these concepts

- The applet creates a new spot object when the applet is initialized. Here's the relevant code from the applet class:

```
private Spot spot =null;
private static final int RADIUS =7;
...
spot   =   new Spot(RADIUS);
```

- The first line shown declares a variable named spot whose data type is Spot , the class from which the object is created, and initializes the variable to null. The second line declares an int variable named RADIUS = 7. Finally, the last line shown creates the object; **new** allocates memory space for the object.

Coding using these concepts

- Messages:

```
g.setColor(Color.white);
g.fillRect(0,0,getSize().width -1,getSize().height -1);
```

- Both are messages from the applet to an object named **g** — a Graphics object that knows how to draw simple on-screen shapes and text. This object is provided to the applet when the browser instructs the applet to draw itself.
- The first line sets the color to white;
- The second fills a rectangle the size of the applet, thus painting the extent of the applet's area white.

Coding using these concepts

- **Inheritance**:
- To run in a browser, an object must be an applet. The ClickMe applet object is an instance of the ClickMe class, which is declared like this:

```
public class ClickMe extends Applet implements
MouseListener {
...
}
```

- The **extends Applet** clause makes ClickMe a subclass of Applet . ClickMe inherits a lot of capability from its superclass, including the ability to be initialized, started, and stopped by the browser; to draw within an area on a browser page; and to register to receive mouse events.

Coding using these concepts

- **Inheritance**:
- Along with the benefits of being **Applet**'s subclass, the ClickMe class has certain obligations: Its painting code must be in a method called paint , its initialization code must be in a method called init, etc.

```
public void init(){
    ... //   ClickMe 's initialization code here
}

public void paint(Graphics g){
    ... //   ClickMe 's painting code here
}
```


Coding using these concepts

- **Interfaces**:
- The ClickMe applet responds to mouse clicks by displaying a red spot at the click location. If an object wants to be notified of mouse clicks, the event system requires that the object implement the MouseListener interface.
- The object must also register as a mouse listener. The **MouseListener interface declares five methods**, each of which is called for a different kind of mouse event: when the mouse is clicked, moves outside of the applet, and so on. Even though the applet is interested only in mouse clicks, it must implement all five methods. The methods for the events that the applet isn't interested in are empty.

Coding using these concepts

- **Interfaces**:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class ClickMe extends Applet implements
MouseListener {
    private Spot spot =null;
    private static final int RADIUS =7;
    public void init(){
        addMouseListener(this);
    }
```


Coding using these concepts

- **Interfaces**:

```
public void mousePressed(MouseEvent event){
    if (spot ==null){
        spot =new Spot(RADIUS);
    }
    spot.x =event.getX();
    spot.y =event.getY();
    repaint();
}
public void mouseClicked(MouseEvent event){}
public void mouseReleased(MouseEvent event){}
public void mouseEntered(MouseEvent event){}
public void mouseExited(MouseEvent event){}
}
```


Coding using these concepts

- **API Docs – EXTREMELY IMPORTANT**:
- The ClickMe applet inherits a lot of capability from its superclass. To learn more about how ClickMe works, you need to learn about its superclass, **Applet** .
- How do you find that information? You can find detailed descriptions of every class in the API documentation, which constitutes the specification for the classes that make up the Java platform.
- The API documentation for the Java 2 Platform is online at java.sun.com . The documentation is also included on the CD that comes with our book. When appropriate, this book has footnotes providing the URLs to the API documents for particular classes. It's helpful to have the API documentation for all releases you use book marked in your browser.
- **http://java.sun.com/j2se/1.3/docs/api/index.html**