Today's Agenda

> To talk about object-oriented programming concepts

> To write a program in Processing together

> To show you your second assignment

Object-oriented

What does Object-oriented mean?

"Object-oriented" means that you can write little groups of code which mimic objects or processes in the real world.

Every program builds a certain model of the world where objects in that world interact.

Makes it easy to reuse code. You can create lots of different versions of that same object, but with different attributes.

If you think of an object as a Noun, you can use different Adjectives to describe it.

Object-oriented code helps you set-up a model of the world which is similar to the way humans think of it.

Object-oriented

You can think of programming in these two ways:

a) As a set of extremely detailed instructions.

b) As a set of simplified models of certain objects or processes in the world.

They go hand-in-hand: All instructions imply a model of the world. All models of the world imply a set of objects that have attributes and behaviors (that have the potential to be instructed).

Instructions are the “technical” part of programming. Modeling is the conceptual or “creative” part of programming.

Writing software is really an art as much as it is a science.

You have to be creative, flexible, able to approach things from different ways.

You get to create metaphors of reality.

You get to come up with tricky ways to write instructions efficiently and “elegantly”.

Modeling

“Everything should be made as simple as possible, but no simpler.”

-Albert Einstein

“I have a map of the United States... Actual size. It says, ‘Scale: 1 mile = 1 mile.’ I spent last summer folding it.”

-Stephen Wright

The whole point of a model is that it doesn't include everything. It is a vastly simplified version of reality, which is exactly why it's useful.

To make a model of something means you have to leave most stuff out.

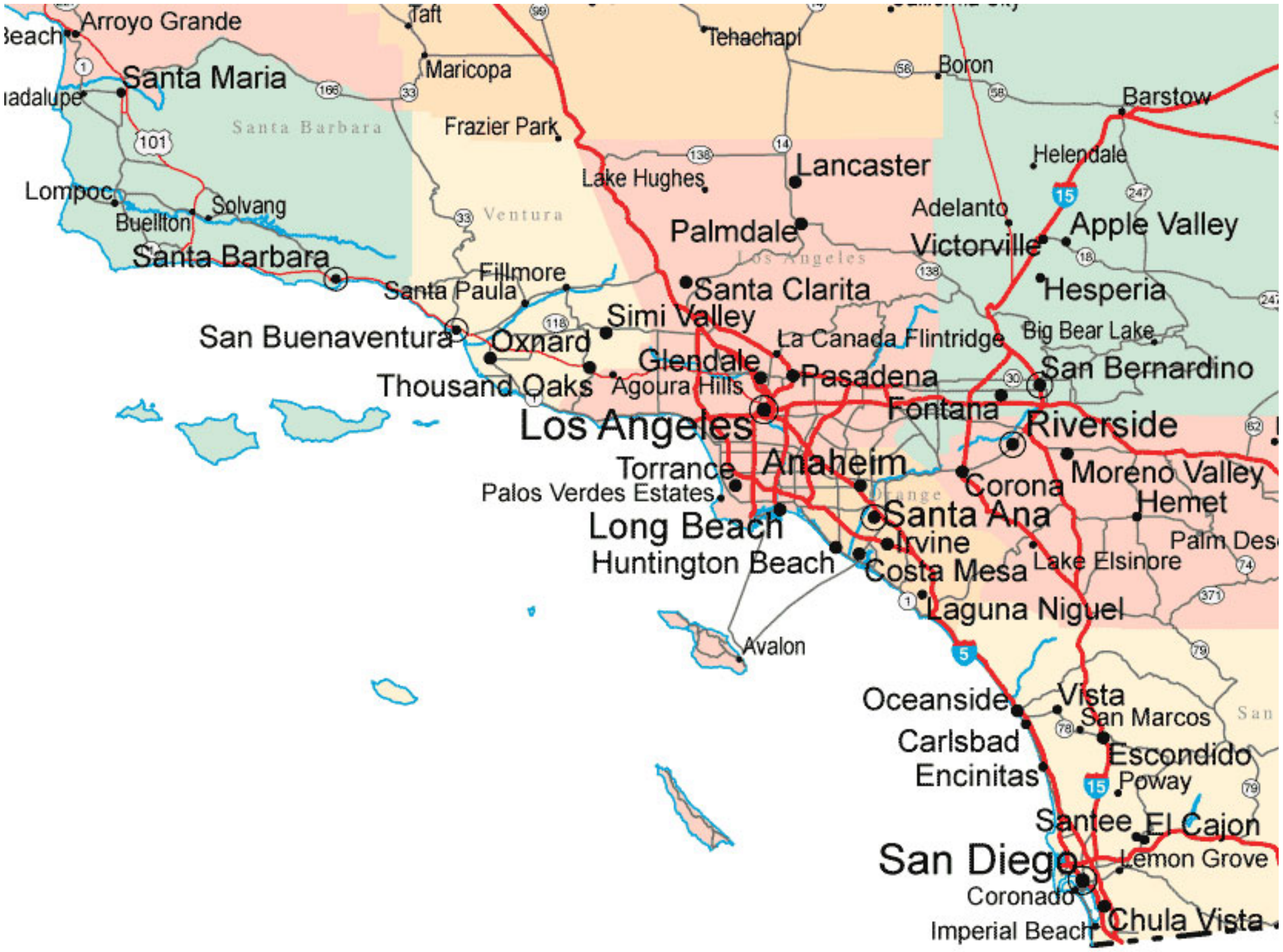
Here's a map (on next slide):

What aspects of reality does it include and exclude?

How do the features work together?

Could they work independently from the other features?

Modeling


Arroyo Grande
Santa Maria
Taft
Maricopa
Tehachapi
Boron
Barstow
Frazier Park
Santa Barbara
Lompoc
Buellton
Solvang
Santa Barbara
Ventura
Lake Hughes
Lancaster
Helendale
Adelanto
Apple Valley
Palmdale
Victorville
Los Angeles
Hesperia
Fillmore
Santa Paula
Santa Clarita
Simi Valley
San Buenaventura
Oxnard
La Canada Flintridge
Big Bear Lake
Glendale
San Bernardino
Thousand Oaks
Agoura Hills
Pasadena
Fontana
Los Angeles
Riverside
Torrance
Anaheim
Moreno Valley
Palos Verdes Estates
Orange
Corona
Hemet
Santa Ana
Long Beach
Irvine
Palm Des
Huntington Beach
Costa Mesa
Lake Elsinore
Laguna Niguel
Avalon
Oceanside
Vista
San Marcos
Carlsbad
Escondido
Encinitas
Poway
Santee
El Cajon
San Diego
Lemon Grove
Coronado
Chula Vista
Imperial Beach

Classes & Objects

The main modeling tool in object-orient programming is the *class* definition.

A **class** is a *description* of a discrete entity within your model.

It describes the attributes that make up a particular thing, and it describes the ways you can interact with that thing.

And an **object** is an actual *instance* of that entity within your model.

For example, your model of a highway system might have a single general Car class. But when you are investigating traffic patterns you might need to create thousands of instances of that Car class.

You can think of an an object as a Noun (an *instance* of the class), which has various Adjectives (*fields*) that are used to describe it, and which can be manipulated by various Verbs (*methods*).

Classes & Objects

```
//class definition
public class Car {
  //important properties
  private String brand;
  private boolean isAutomatic;
  private int year;
  private double maxSpeed;

  //constructor -- turns the class definition into an object you can manipulate
  public Car(String brand, boolean isAutomatic, int year, double maxSpeed)
  {
    //set the "state" of the object
    this.brand = brand;
    this.isAutomatic = isAutomatic;
    this.year = year;
    this.maxSpeed = maxSpeed;
  }
  //accessor methods...
  public String getBrand()
  {
     return brand;
  }
}
```

Object-oriented

The particular ways in which object-oriented code models your problem domain includes these 3 interrelated concepts:

Encapsulation

Inheritance

Polymorphism

Encapsulation

Encapsulation means that each object is a kind of “capsule” of data.

You instantiate an object of a specific class.

Your object has data (instance variables).

You access the data through the specific methods defined in the object.

There is no other way to access the data, except through these methods.

The data is “hidden”, private to the object, and it is clearly defined how we can access and change the data.

For example, if you were modeling a bank account, your Account class might have a getBalance and a setBalance method to access and change the data. You could add a check inside the setBalance method so that it wouldn’t let you set the balance to a negative number.

By constraining the way in which you can get to data, you simplify your programming model and hopefully reduce unwanted side-effects and bugs that crash your program.

Inheritance

Inheritance is a powerful feature of object-oriented languages which makes it easy to re-use and extend existing code-- either from a library or that you have created yourself.

For example, you might want to model birds for a scientific or artistic simulation of flocking behavior.

Inheritance lets you first model a "superset" of general qualities that are shared amongst all kinds of birds, and then modify them as needed for particular individuals or subsets of those birds.

The root of your inheritance model is called the *superclass* or the *base* class.

```
abstract public class Bird
{
    Point currentPosition = new Point(0,0);
    String typeOfBird;
    public Bird(String typeOfBird) {
      this.typeOfBird = typeOfBird;
    }
    abstract public void nextPosition();

    public String toString() { return typeOfBird + ":" + currentPosition; }
}
```

Inheritance

```
The children of the superclass are called subclasses. In this example, SocialBird is a
    subclass of Bird, and Robin is a subclass of SocialBird.
```

```
abstract public class SocialBird extends Bird {
    int numFriends;
    public SocialBird(String typeOfBird, int numFriends) {
          super(typeOfBird);
          this.numFriends = numFriends;
    }
}

public class Robin extends SocialBird {
    public Robin() {
          super("Robin", 10);
    }

    public void nextPosition() {
          //figure out next point based on current position and nearby friends
          currentPostion = ... //some complicated logic...
    }
}
```

Inheritance

```
abstract public class IconoclastBird  extends Bird {
    public IconoclastBird(String typeOfBird) {
          super(typeOfBird);
    }
}


public class Eagle extends IconoclastBird {
    public Eagle() {
          super("Eagle");
    }

    public void nextPosition() {
          //the Eagle has no friends and always goes straight
          currentPostion = new Point(currentPosition.x + 1, currentPostion.y + 1);
    }
}

(draw inheritance on board...)
```

Polymorphism

In the above example, the nextPostion method was polymorphic.

Polymorphism means that you can call an operator on a super type of the object and the language will, on the fly, figure out the most specific thing it could possible mean and execute that method.

Since our base class, Bird, has a method called *nextPosition*, all classes that extend Bird are guarenteed to have that class as well. They can also override it. If they do override it, the language will always know to use that version of it, even if you aren't referring to the specific type explicitly.

```
public class BirdSimulation {
    public static void main(String[] args) {
     new BirdSimulation();
    }
    public BirdSimluation() {
     List birds = new ArrayList();
     birds.add(new Robin());
     brids.add(new Eagle());

     for (int i = 0; i < birds.size(); i++) {
          Bird b = (Bird) birds.get(i);
          b.nextPosition(); //automatically figures out what subtype this Bird is
          println(b);
     }
}
```