TODAY’S TOPIC:

- Last time, we talked about differences between \textit{procedural} and \textit{object oriented} programming styles.

- Today, we will cover \textit{objects} and \textit{classes}. 
WHAT IS AN OBJECT?

• Just like the real world, objects are things.

• They have names, attributes, and behaviors.

• They can be used along with other objects to complete tasks.

• They maintain their own data

WHAT IS AN OBJECT?

• An object is an entity that contains both data and behavior.

  – The data represents the current state of the object.
    • The data are called attributes, and they differentiate between objects.

  – The behaviors are what the object can do.
    • These behaviors are called methods and are invoked by sending a message to the object.
ON TO OBJECT ORIENTED PROGRAMMING

- In this class we will focus on OO programming and design.

- An OO approach is best utilized in building large scale systems that require many programmers to develop.

- In some cases it may feel like overkill on smaller projects to use OO, but it can still be effective.

- It is perfectly normal to feel uneasy or confused when learning to “think” in objects.

OBJECTS ARE INSTANCES

- It is important to differentiate between objects and classes.
  - An object is a specific instance of a class.
  - Each object has its own data and state.

- There are two steps to creating and using an object:
  - Declare the object. (give it a name)
  - Instantiate the object. (creates a new object with an initial state)
WHAT ARE CLASSES.

• A class is a special type of code file that contains the implementation of the object’s data and behavior.
  – The data is defined by instance variables.
  – The behaviors are described by methods.

• The class also provides the steps for creating instances of the class.

AN ANALOGY.

• Classes are blueprints for building instances.

• Classes are a cookie cutter, to make many instances of the objects.
DIE OBJECTS

• Think about a real world die:
  – What are attributes do they possess?
  – What actions can I take with them?

DIE OBJECTS – IDENTIFYING ATTRIBUTES AND BEHAVIORS

– Attributes:
  • a specific Number of Sides
  • colors (surface and pips)
  • Material

– I can take actions with the die (Behaviors):
  • Count the sides
  • Pick it up
  • Roll it
  • Read the value on top
DIE OBJECTS – DETERMINING AN INTERFACE

– It has physical attributes:
  • a specific Number of Sides
  • colors (surface and pips) \textit{Not relevant}
  • Material \textit{Not relevant}

– I can complete certain actions with the die:
  • Count the sides
  • \textit{Pick it up Not relevant}
  • Roll it
  • Read the value on top

– Sometimes there are attributes or behaviors that are \textit{irrelevant} to my program, so I leave them out.
  – In this example, I don’t care about color, so I don’t use it.
  – Is there ever a case that I do want to have die objects of different color in my program?

– I can also create virtual objects that don’t exist in the real world.
  – For example, a \textit{three-sided} Die.
DIE OBJECTS – UML CLASS DIAGRAM

Die

- numberOfSides: int
- currentValue: int

+ getNumberOfSides(): int
+ roll(): int
+ getCurrentValue(): int

DEFINITION TIME

– What is an interface?

– What is an implementation?
**PUBLIC INTERFACE**

- The **public interface** is the **outside** view visible to other objects.
  - The public interface defines what the object **does**, and how to **interact** with an object.
  - For example, what **messages** can be sent to the object.
  - This refers to the **application programming interface (API)** that other programmers have available to them from your classes.
  - The interface can be viewed as a **contract** with other programmers.
- What we mean by this use of interface is **similar** too, but **distinct** from the user’s **graphical user interface (GUI)**.

**PRIVATE IMPLEMENTATION**

- The **implementation** is the **internal** view of the object that is hidden from the user’s view.
  - The implementation are the details of **how** the object completes its behaviors.
  - The implementation includes what **attributes** the object **maintains** and the **methods** that are **invoked** when the object receives a message.
AN EXAMPLE – A WRISTWATCH

**Interface**
- Push to start/stop stopwatch
- Pull out and turn to set time
- Push to reset stopwatch

Image Source: http://www.thewatchspot.co.uk/images/BlogImages/Large/Jaquet-Droz-149-5.jpg

AN EXAMPLE – A WRISTWATCH

**Implementation**

The interior design details to accomplish the tasks.

Image Source: http://www.thewatchspot.co.uk/images/BlogImages/Large/Jaquet-Droz-149-5.jpg
DESIGNING A MINIMAL INTERFACE

• Provide *only* what is needed in the *interface*.
  • Design based the prospective of the *user* of the class/object.
  • The interface can be *adapted* later, if necessary.
  • Hide any implementation details that are not *necessary* to *use* the object inside the class.
  • The goal is to create an object that is *simple* to use.

• For Example, what is the interface/implementation of a automatic vs. manual transmission in a car?

YOUR TURN

• Suppose I have a collection of Lego.
• Some are flat panels, some are 2x4 blocks, and others are 2x1 blocks.
• Write a procedure using pseudo code or descriptive steps of how you would sort the pile of Lego into the three different types.
A SOLUTION WITH OBJECTS

- I have an object, LegoSorter that is composed of other objects, LegoFilter.

- Each LegoFilter has a specific implementation to filter out Lego with certain attribute values.

- This works in the real world for size/shape. Could you create an object that sorts on color?

REPRESENTING A CLASS

- We will use Unified Modeling Language (UML), to define our object models.

- This example from the book shows a view of a class with its instance variables (data) and methods (behaviors).

- More on this later.
AN EXAMPLE

• Let's look at pages 23 and 24 in the book at the IntSquare class.