Day One - Teacher's Guide

Introduction

Begin the class by posing the questions: "Have you ever wondered how apps are created? How games are made?" Some students may know that it has something to do with code. Some may have already created games using game creation software. Next you should introduce them to a few iPhone apps (see the list in the "Model-View-Controller: An Overview" section), try and get a bit of a "wow" factor in, or at the very least some amusement. Make sure you engage everyone during this early time, getting students excited is crucial to success.

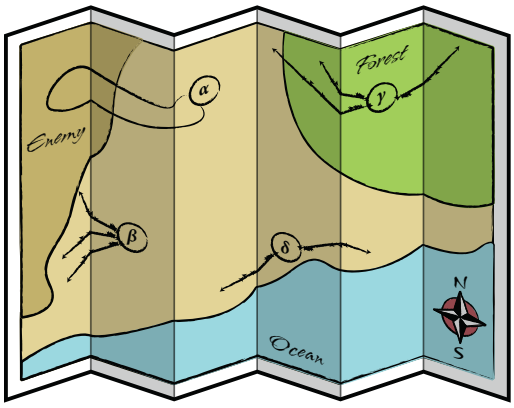
Object-Oriented Programming: An Overview

You will need to cover the following points:

- An object is an abstract concept. The basic idea is that a program is laid out in a modular fashion, with different objects controlling different aspects of the program's behavior.
- Objects can talk to each other through methods or functions (terminology differs based on language).
- Objects have properties, which describe a characteristic of an object. These properties can be set by other objects.
- An object should always have a clear purpose: keeping the modular design is crucial.

An excellent way of demonstrating the concept of an object is to use the example of an army in a campaign. There are several different troops, and each troop has a specific function. For instance, one troop is tasked with gathering intelligence and reporting it back. Another troop is assigned to attacking to the West of their current location, and yet another troop is responsible for protecting the Northern flank. A final troop is stationed on the South coastline of the area, prepared to defend if necessary.

In this example, each troop represents an object in an application, which is represented by the army as a whole. In this case, the application's purpose would be to win a war.



An illustration of the army analogy, also found on the student reference page.

Model-View-Controller: An Overview

You will want to introduce MVC as a “paradigm”, or a “design pattern”. It is a way of laying out the code for a program so that it is organized, efficient, and modular. It is the standard design pattern for most object-oriented languages. You will need to cover the following points:

Every object must either be a Model, a View, or a Controller. Merging any of them defeats the purpose. The Model represents data, the View is an interpretation of that data. The Controller handles and interprets user input and updates the Model accordingly.

The View polls the Model for changes, and updates itself accordingly. To demonstrate MVC, you can use the following iPhone apps:

- Bubblewrap (Model: List of bubbles, their positions, and whether or not they are popped; View: The image of bubbles on the screen; Controller: The code that takes the position of a tap, figures out which bubble was hit, and then sets that bubble to popped in the model)
- Mover (Model: List of files on disk and available to user; View: Flat view on screen with files displayed as small items; Controller: The interpreter of taps and drags that tells the filesystem what to send where, and the receiver of files sent by other devices which are then stored in the model)
- Notes (Model: Database of notes made by the user; View: A list of the titles of notes which can be tapped and expanded to show their full content; Controller: The code that interprets taps on either a note or the plus sign button and updates the model accordingly)
- Photos (Model: Database of photos taken on the phone; View: The expandable list of photos and their albums; Controller: The code that tells the model when to provide a big version of a photo, a thumbnail, or other information)

Students may try to call back to the army analogy in order to understand MVC. This may not be beneficial, as it is somewhat difficult to connect the two. The only thing to remember here is that every object has a very specific purpose, and shouldn’t have functions that overlap others. This does not mean they are entirely independent: each object should absolutely depend on other objects for extended functionality. The key is that it is a modular design, and can easily be broken down into pieces based on the different functions of the application.

Object-Oriented Programming

Basics

Have you ever wondered how the ever-popular "apps" of the App Store are made? How your favorite video games are created? Here's a hint: computers don't speak English. The processing core of a computer simple reads and writes 1's and 0's to and from a disk. The rest of what you see on the screen is based on the interpretation of those numbers by a programmer.

Very few programmers deal directly with raw binary (1's and 0's), so don't be too worried about understanding it. Because of the complexity and difficulty that comes with dealing with binary, developers (another term for programmers) created **layers of abstraction**, known as **programming languages**. The idea is that someone can write a program without ever touching any binary by using one of these languages, built "above" another language. Languages are built on top of each other until they reach a level of simplicity and usability that satisfies the developers.

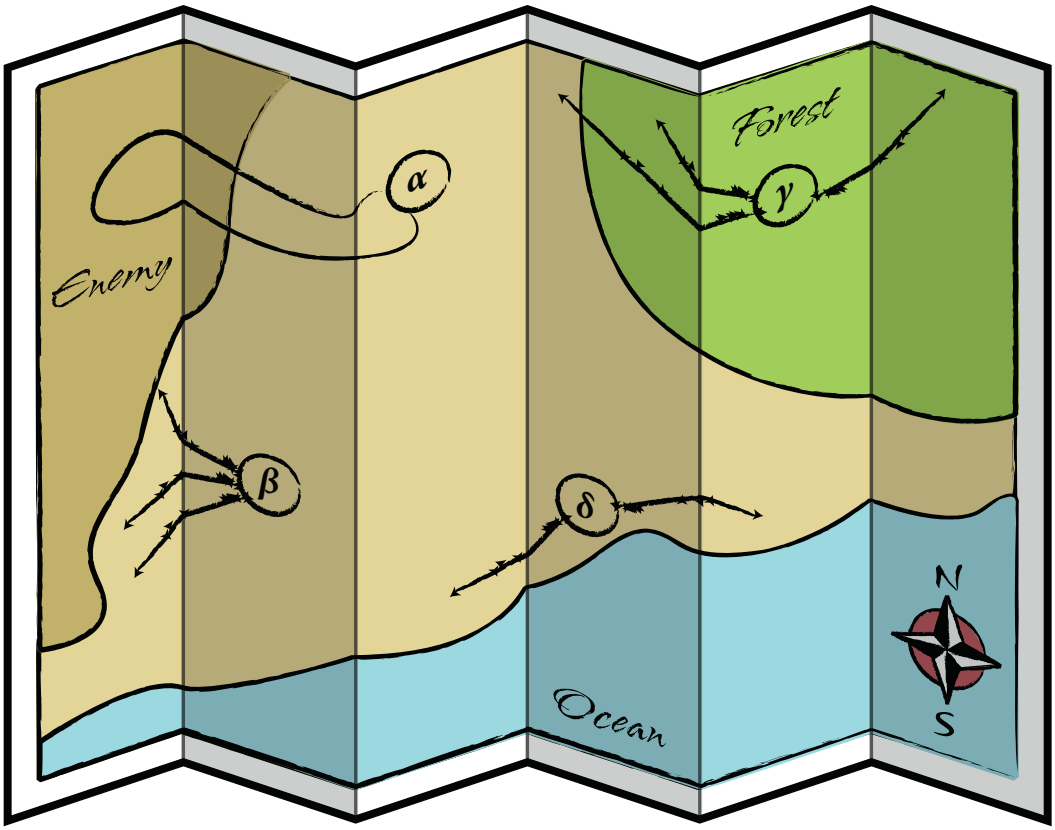
Once a developer has written their application using one of these languages, the entire program is then **compiled** into binary, which can then be used by the computer. There are many different types of programming languages, but one has emerged as the favorite of many developers.

Object-Oriented Programming

Traditional programming languages such as ANSI C are written as a chain of commands for the computer to execute, known as **functions**. Values can be stored in **variables**, but one cannot actually use the variable unless you pass it to a function. The layout of a program is similar to a very long math operation, and becomes increasingly complex as more features are added. For example, a very popular tool for dealing with media files has a whopping **3998** lines of code in a single file...and that isn't even the entire program. The code is unorganized and unruly, making it very difficult to work with and add to.

The concept of object-oriented programming is not new: it was first introduced in the early 1970's. The idea is that instead of having a massive list of functions that simply exchange numbers, there is an abstract concept of "objects" present. Objects can store data, interact with users, crunch numbers, and do anything else a program is capable of doing. Each object has a very clear definition of its purpose, and interacts with other objects in an organized manner.

This may seem confusing, but an easy way of looking at Object-Oriented Programming is an army analogy. In the following image, you can see a wartime map, displaying different troops in an army. Troop α is responsible for infiltrating enemy lines, gathering intelligence, and coming back to safety. Troop β is in charge of attacking on the western front, and troop γ's task is to defend to the North. Finally, troop δ is set to defend the coast, in case of an amphibious attack.



Now, imagine that each one of the described troops is an object in a war computer game. And there, you have object-oriented programming.

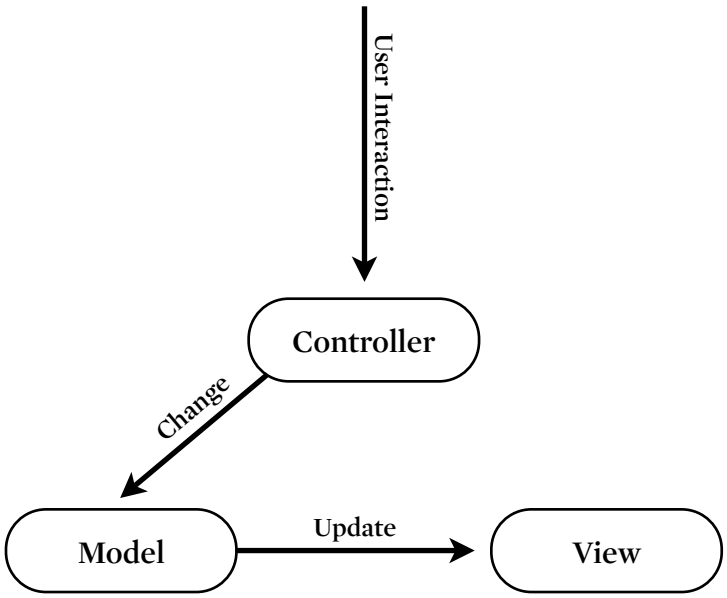
Some key points to remember are:

- Objects are modular by design, meaning that each object is fairly self-contained and focused in purpose.
- Just because objects are self-contained doesn't mean they can't talk to each other: they can, and should when necessary.
- An object can have many properties, which other objects can change, or simply read. These properties describe traits about the object and its behavior.

Model-View-Controller

Basics

Model-View-Controller, or MVC is a programming "paradigm". In other words, it is a pattern or template for the way a program's code is written and organized. It is specific to Object-Oriented Programming, the type you learned about earlier.



The above diagram shows the basic concept of MVC. This probably seems very complicated and a bit daunting, so let's go through it one step at a time.

1. **User Interaction:** Let's say a user presses a button in a calculator program. The button they press is '=', which should execute any math operations the user already entered into the calculator. For the sake of example, let's say that the user already pressed the '2' button, the '+' button, and finally the '2' button again. When the user presses the '=' button, the Controller intercepts that and talks to the Model.
2. **Change:** The Controller tells the Model to calculate the answer to the problem that's already been entered. The Model does so, and publicizes its calculated answer.
3. **Update:** The View receives a notification when the Model changes, and then sees that an answer has been calculated. It updates itself so that it is displaying the answer to the user.

Even when you break it down into pieces, it is still a complicated pattern to understand. It might help you to think of it in terms of an example application.

The drawing above depicts a computer recording input from a microphone. This is an example of the MVC paradigm:

- The Model is the file in the computer where the sound is being stored, as well as the code that is used to translate from binary to data that can be used to create a waveform.
- The View is the waveform on the screen. It uses code in the Model to get information, and then displays that information in a way that you understand.
- The Controller is the code that is used, whether it is inside the microphone or inside the computer, to translate the vibrations recorded by the microphone into data useable by the Model.

As you can see, the MVC paradigm can be applied in many instances, even though it is designed for use with Object-Oriented Programming. When you join the two concepts together, the result is clean, organized, and modular code that is efficient to execute.