Introduction

Java, C++, and C# are known as Object Oriented Programming languages. This refers to how we design our programs in order to:

- Make the source code easier to follow
- Make the source code easier to maintain
- Make the program easier to extend with additional functionality
- Keep portions of the program "modular" - so that code relating to similar functionality are grouped together.

Intro Vocabulary List

Object Oriented Programming

Cohesion

Coupling

High Cohesion and Low Coupling

When dealing with software design, one term that pops up is "High Cohesion and Low Coupling" as a goal for how our programs should be structured.

Cohesion

> *"Cohesion refers to the degree to which the elements of a module belong together."*
>
> https://en.wikipedia.org/wiki/Cohesion_(computer_science)

In other words, the goal is to group code together that *logically belongs* together.

Good Example: In a video game, storing each gameObject's *velocity* and *x,y coordinates* in the *gameObject* class, so that each object gets its own version of these variables. The gameObject can then modify its own velocity and coords.

Bad Example: In a video game, storing each player's *score* in the *world class*, instead of in each player's object. This means each player object has to access the world class in order to get its score, which can make the code uglier and also doesn't logically make sense from a design standpoint.

Written by Rachel J Morris
2016-09-23

Coupling

> *"In software engineering, **coupling** is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules."*
>
> https://en.wikipedia.org/wiki/Coupling_(computer_programming)

Think of coupling like the ability to take out some chunk of code, and either replace it, or put that chunk of code in another program. If it is pretty difficult to do, then it has **high coupling**. If it isn't too hard to pull off, then it has **low coupling**.

It is good design to have low coupling, because then the code is more self-contained; this means that it can be easier to test it because all its functionality is in one area, but also that functionality can be added to with minimal effect to the overall program.

Is OOP the only way to program?

Nope! There are other programming paradigms out there, such as functional programming, imperative programming, structured programming, and others. But with languages like C++, Java, and C#, it is pretty standard to write in an object oriented style (perhaps borrowing some functional elements – but you don't need to worry about that right now!)

```
                "Don't worry if it doesn't work right.
              If everything did, you'd be out of a job."
                                  - Mosher's Law of Software Engineering


        "When someone says: 'I want a programming language in which
          I need only say what I wish done', give him a lollipop."
                                                        - Alan J. Perlis


            "Good design adds value faster than it adds cost."
                                                        - Thomas C. Gale
```

How do we build OOP programs?

How do we build programs that are object oriented? That is where classes and functions come in handy!

Function Vocabulary List

Function

Parameter

Argument

Return Type

Void

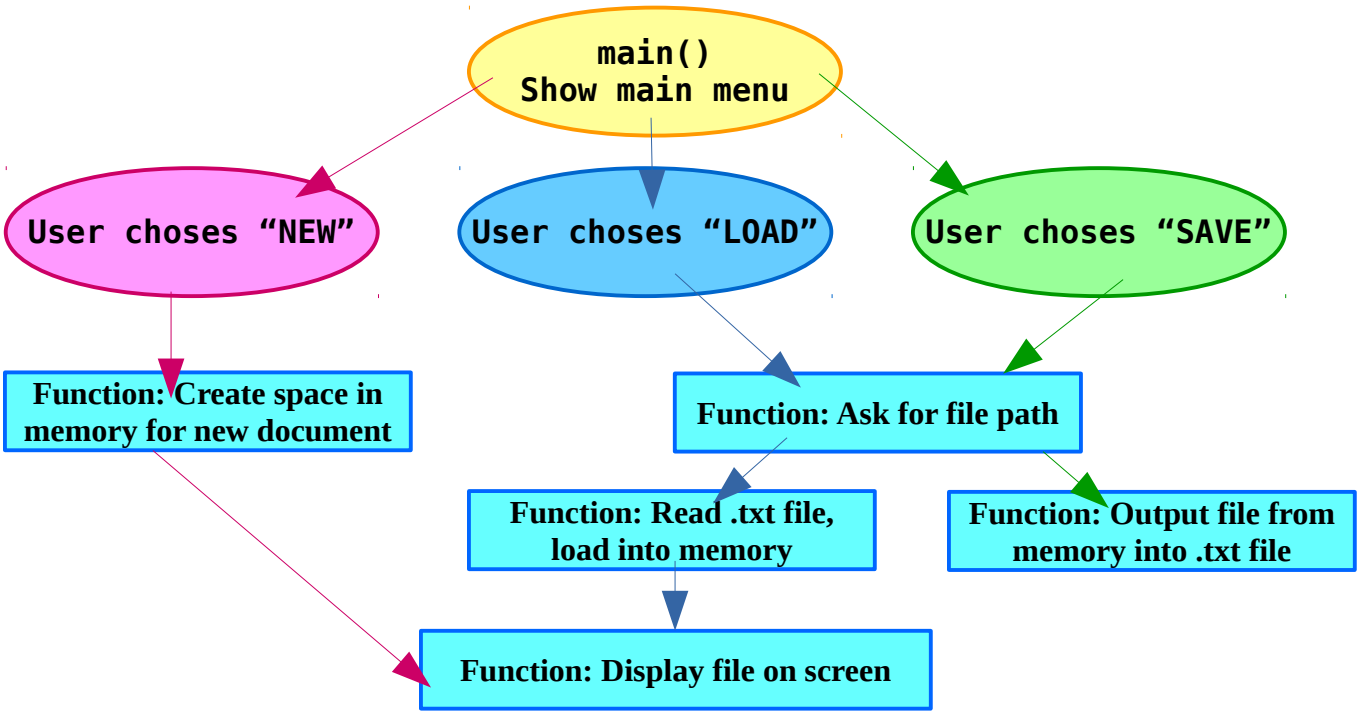
Functions

A function contains code to execute, and a function has a name. Functions are the simplest way to modularize a program, putting different functions of a program in their own *functions*.

Example:

We are writing a text editing program and it has features like load existing document, create new document, and save document.

Each of these features might have their own functions, or set of functions, so that we're not storing all of the program's functionality in **main()** (or the starter function of the program). Look at the following diagram and study the paths each feature takes.



From the diagram, notice…

- Certain functions were called by different features – **new** and **load** use “Display file on screen”, **load** and **save** use “Ask for file path”.
- Multiple functions are used, each handling their own little task. Because of this, each function might only be a few lines long – and that’s good. It increases readability and maintainability.
- Functions can call other functions

If you were sketching out the “flow” of your program on paper, each step could possibly be implemented as its own function. We can turn this list for the flow of a simple Bank program into a set of functions for the program:

- Display main menu (Main Menu 1)
- Get user’s input (Main Menu 2, Deposit 1)
- Check to see if input is valid (Deposit 2)
- Deposit amount into account (Deposit 3)

```
My Bank Program

Main Menu:

1. Display the balance

2. Ask the user if they want
to deposit, withdraw, or quit.

Deposit:

1. Get the deposit amount

2. Check to see if the amount
      is valid (not < 0?)

3. Deposit into account if
      valid.

Etc...
```

Then, functions like “Check to see if input is valid” can be used in multiple areas, like deposit and withdraw.

By having a special function that deals with the validity of the $ input, we can then go back and change *what is valid user input* if we need to. Perhaps we missed a case (“is the withdraw amount greater than the balance?”), or we want to add more specific cases to handle (“is the withdraw amount over $200?”). If this is all contained within one function, we only need to update it in one place!

Review

Think about each of these types of programs, and try to come up with a simple list of *functions* that might be needed to implement them.

1. A recipe program that allows the user to store a list of recipes or choose a recipe either from a list of all recipes or by doing a search for keywords.	
2. A “go fish” card game where each player gets a turn to draw a card and ask another player if they have a specific card.	
3. A simulation that helps the user learn to drive. Includes being able to accelerate and break the car, check to see if the car has hit any obstacles, and check to see if the car is going the speed limit.	

Input and Output of Functions

Beyond just being ways to organize our code by nice names, functions can also have input and output (or they can have one or the other, or neither!)

For example, it could be useful to have one function to calculate the price of an item after tax has been applied to it. For a function like this, the input could be the *itemPrice* and the *cityTaxRate*. Then, after the calculation, the output would be *pricePlusTax*.

But the input and output doesn’t have to only be numbers – any type of information we can create a variable as, we can also use as both the input parameters (you can have more than one!) and the output return type.

Function Input/Output Examples

Function Name	Input	Output
CalculateArea	Width, a float Length, a float	Area, a float
IsUserRegistered	Username, a string	IsRegistered, a boolean
GetCustomerName	ID, an integer	CustomerName, a string

Usually, a function will return only one output, but it could take in multiple inputs.

A function's **output** is not given a name, but a data-type must be specified: float, int, string, etc. - The data-type of our output value is known as the return type.

A function's **inputs** must have their data-type specified, and be given a name. The name will be used throughout the function in order to do calculations. The inputs of functions are known as parameters.

When a function is invoked from another part of the program, it is known as a function call. The code that does the *calling* is responsible for providing **input** to the called function (at this point, the input is called an argument, not a parameter), as well as handling the **output** of the function – such as by declaring a new variable and storing the function output there.

No output? No problem!

Some functions won't require returning any output, but a return-type must be defined when creating a function. In these cases, we mark the return-type as void, for any functions that have no data to return as output.

Review

Now think more in-depth about functions: What would be the function name, input, and output for each type of feature being implemented?

	Input	Output
1. A function that calculates the perimeter of a rectangular plot of land.	1. 2.	1.
2. A function that takes a breed of dog, searches a database, and returns the name of the first match.	1.	1.
3. A function that takes the time (Eastern timezone), what the local timezone is, and only *displays* the resulting local timezone; does not return anything.	1. 2.	NONE

Classes

Classes are the next step up from functions. Classes allow us to further modularize our programs by creating objects.

Objects can contain their own variables and functions (these are called member variables and member functions or methods), so in a way they are almost like their own mini-programs inside your main program.

Class Vocabulary List

Class

Member Variable

Member Function / Method

Object / Instance

Composition

Inheritance

With more sophisticated programs, there could potentially be a lot of action and *interaction* going on, and only tackling the design with functions would still eventually lead to messy code. By writing classes, we can make objects that can be used in multiple programs, as well as being used throughout the program in multiple places.

Class Examples

Class Name	Member Variables	Member Functions
TextDocument	• Filename • Contents	• Clear() • Save() • Save(NewFilename) • Load() • Load(NewFilename)
Player	• Name • Image • X, Y • Speed	• Move(Direction) • SetImage(Filename) • SetSpeed(NewSpeed) • SetCoordinates(NewX, NewY) • Draw(Screen)
Button	• BackgroundColor • TextColor • Text • ID • IsEnabled	• OnClick(Event) • SetBackgroundColor(NewColor) • SetTextColor(NewColor) • SetText(NewText) • GetText() • GetID() • DisableButton() • EnableButton()

Classes and Objects

When we declare a new class, we are effectively creating new data-types for our program. We can then declare new variables whose data-types are ***TextDocument***, ***Player*** and ***Button*** – and we can declare more than one of each, too!

```
Player plumber;
Player turtle;

Button pauseGame;
Button goBack;

TextDocument highScoreList;
TextDocument saveGame;
```

The class is how we define our new data-types. When we actually *declare a new variable* of that data-type, that variable is known as an object or an instance of the class.

Written by Rachel J Morris
2016-09-23

Review

For each class type, list out member variables and member functions it might have. You don't have to define the functions' inputs like in the example above – just come up with function names!

Class Name	**Variables**	**Functions**
1. ContactList		
2. Sandwich		
3. Radio		

Objeception

Because writing a class makes a new data-type, and because classes can contain member variables, when we write a new class, we can store variables of that class-type in other classes. Beyond that, part of Object Oriented Programming includes building ***families*** of objects.

When a class *contains* another object, it is known as a "has-a" relationship (or composition).

When a class *inherits* from another class, it is known as an "is-a" relationship (or inheritance).

Design is important!

There's more to programming than just writing text in an editor!

As you can probably guess, once we jump into classes and object oriented programming, program designs can become quite complex. This is why design is important – both planning the structure ahead of time, and also going back to existing code and cleaning it up so that it stays **readable**, **maintainable**, and **extendable**.

In your day-to-day life, try to practice your design skills by abstracting objects you use every day into *classes* and think about what variables and functions they might contain.

Written by Rachel J Morris
2016-09-23