Introduction to Programming Concepts
Instructor: Frank Stepanski

Learning How to Program

A common pitfall for beginners is getting stuck figuring out which programming language is best to learn first. There are a lot of opinions out there, but there's no one "best" language.

Here's the thing: In the end, language doesn't matter THAT much. Understanding data and control structures and design patterns does matter very much.

Every language—even common scripting language such as JavaScript—will have elements that you'll use in other languages as well and will help you learn

JavaScript History and Background

Use of the Web exploded in the mid-1990s after the first graphical browsers appeared. The need for computation associated with HTML documents, which by themselves are completely static, quickly became critical.

Computation on the server side was made possible with the Common Gateway Interface (CGI), which allowed HTML documents to request the execution of programs on the server, with the results of such computations returned to the browser in the form of HTML documents.

Computation on the browser end became available with the advent of Java applets. Both of these approaches have now been replaced for the most part by newer technologies, primarily scripting languages.

JavaScript was originally developed by Brendan Eich at Netscape. Its original name was Mocha. It was later renamed LiveScript. In late 1995, LiveScript became a joint venture of Netscape and Sun Microsystems and its name was changed to JavaScript.

A language standard for JavaScript was developed in the late 1990s by the European Computer Manufacturers Association (ECMA) as ECMA-262. This standard has also been approved by the International Standards Organization (ISO) as ISO-16262.

JavaScript code is embedded in HTML documents and interpreted by the browser when the documents are displayed. The primary uses of JavaScript in Web programming are to validate form input data and create dynamic HTML documents. JavaScript also is with the Rails Web development framework.

In spite of its name, JavaScript is related to Java only through the use of similar syntax.

One of the most important uses of JavaScript is for dynamically creating and modifying HTML documents. JavaScript defines an object hierarchy that matches a hierarchical model of an HTML document, which is defined by the Document Object Model.

Elements of an HTML document are accessed through these objects, providing the basis for dynamic control of the elements of documents.

Reference:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/JavaScript_Overview

Note: After learning the core concepts of programming with JavaScript, and you want to continue to learn more JavaScript concepts, you can register for the Introduction to JavaScript class at eClasses.org.

The Program Development Life Cycle

The general purpose for creating a program is involving problem solving strategies which involved understanding the problem, devise a plan, carry out the plan, and review the results.

1. **Analyze the problem:** Determine what information you are given, what results you need to get, what information you may need to get those results, and in general terms, how to proceed from the known data to the desired results.

2. **Design a program to solve the problem:** This is the heart of the program development process. Depending on how hard or complex the problem is, it might take one person a few hours or it might take a large team of programmers months to carry out this step.

3. **Code the program:** Write statements (program code) in a particular computer language (JavaScript for our class) that implement the design created in Step 2. The result of this step is the program.

4. **Test the program:** Run the program to see if it actually solves the given problem.

This process of analysis, design, coding and testing forms the core of what is known as the program development life cycle. The word cycle is used here because as with the general problem solving process, we often have to return to previous steps as we discover flaws in subsequent ones.

In order to develop a program to solve a specific problem, you must know and understand programming concepts in general and a specific language in particular.

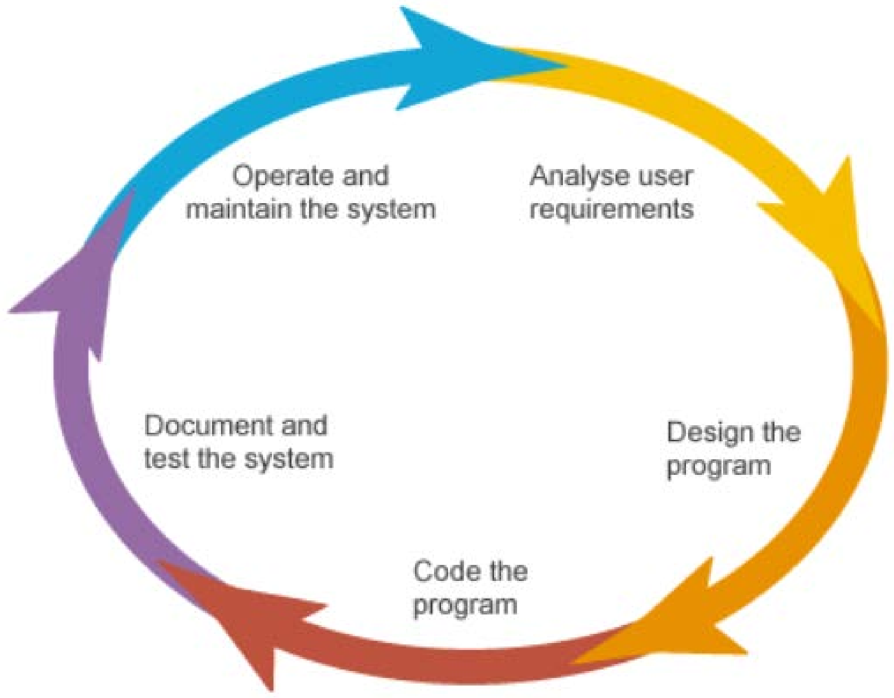

Operate and maintain the system
Analyse user requirements
Design the program
Code the program
Document and test the system

Figure 1 - Cyclical Process of the Program Development Life Cycle

Computer programs are written in a programming language and follow exact **syntax**. The syntax of a computer language is its rules of usage. If you don't use correct syntax, your program won't work.

Program Development as a Cyclical Process

Referring to Figure 1, it may seem to imply that once you have completed a step, you never return to a previous one. In fact, in real-world program development the following is true:

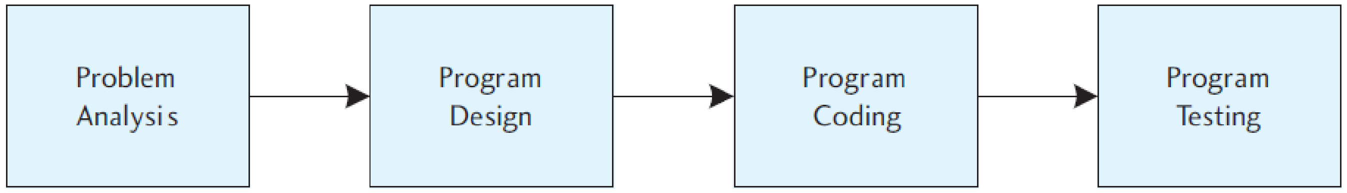
- The design process often uncovers flaws or inadequacies in the analysis of the problem and may necessitate a return to this phase.

- In coding the program certain problems may be encountered that lead to the need for modifications of additions to the program design.

- The final testing phase, which can last for months for very complex programs, inevitable uncovers problems with the coding, design and analysis phases.

The need to return to previous steps in the process of completing a program is why we refer to it as the program development life cycle. The word cycle implies that you may have to repeat previous steps, perhaps over and over, until the program works correctly.

In fact, in commercial programs (written for profit), the cycle is rarely ever complete.



Note: Depending on the complexity of the program, there may be other aspects to the design phase such as creating the user interface (UI) which would be a collection of displays, menus and other features to allow easy user input of data.

Data Input

The **input** operation in a program transmits data from an outside source to the program. Often this data is typed at the computer keyboard by the person using the program.

The program **prompts** the user to enter a value.

Input from the keyboard is very common, but data may be input into a program by other means as well such as by clicking or moving the mouse.

Program Variables and Constants

When we write a program, most of the time we don't know the actual number or other data that the user will enter while running (executing) the program, therefore, we assign the input data into a program **variable**.

A variable is called a variable because it can vary. It is a quantity that can change value during the execution of a program. Anytime we need to refer to that data in a subsequent program statement, we simply refer to its variable name. At this point the value of the variable, the number or other data value it represents will be used in this statement.

Let's say you are creating a program to calculate total cost of purchases made by customers:

First, your program might use a tax rate value in several places. If you enter the value of the tax rate as a number (a constant) and the tax rate changes, you would have to go through your program code and change it each time the constant is typed into the code.

On the other hand, if you give the tax rate a variable name and use this variable name wherever you need to use the tax rate in a calculation, when the tax rate changes you would only have to change the value in one place. You would simply change the one line of code where you assigned the constant value to the variable name and everywhere in the code where that variable was used, the new value would be in effect.

This would be a **named constant**, a variable that is solely used in your program to store an internal value used for calculations and not user input.

Numeric Data

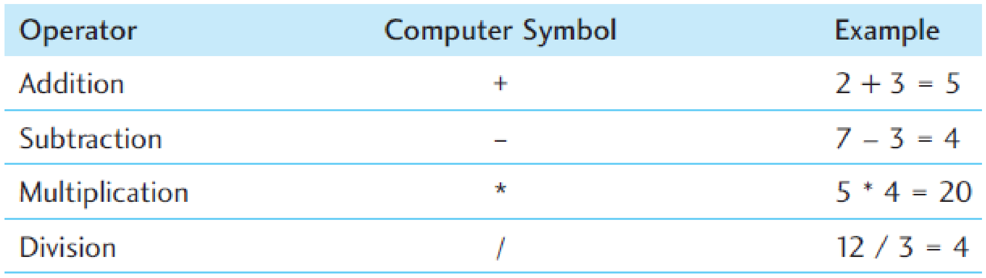
Most programming languages allow for at least two types of numeric data to be used in programs: **integers** and **real** (or **floating point**) numbers. These types of data are stored differently and they take up different amounts of space in the computer's memory.

Integers are positive or negative or zero whole number. For example, the numbers 430, -17 and 0 are integers.

Real numbers are any number that is not of Integer type which includes all numbers with fractional parts, such as 4.6, -34.786 and 7.0.

Operations on Data

Almost all programming languages have basic arithmetic operators such as addition, subtraction, multiplication and division.



Operator	Computer Symbol	Example
Addition	+	2 + 3 = 5
Subtraction	–	7 – 3 = 4
Multiplication	*	5 * 4 = 20
Division	/	12 / 3 = 4

There is also a hierarchy of operations which define the rules of arithmetic that tell us the order in which arithmetic operations are performed:

1. **First -** Perform the operations in parenthesis.
2. **Second** - Perform exponentiations.
3. **Third -** Do multiplication, division, addition then subtraction.

So this arithmetic expression would be calculated in order of:

$$
\begin{aligned}
6+ 8/(2 * 4) &= 6 + 8/8 \\
&= 6 + 1 \\
&= 7
\end{aligned}
$$

Character String Data

Loosely speaking, a character is any symbol that can be typed on the keyboard. This includes letters, numbers, punctuation, spaces, etc.

A **character string** (or more simply, a **string**) is a sequence of characters. In most programming languages, a string is enclosed in quotation marks.

A single character is also considered a string, so "B" and "C" are strings. But a string may be void of any characters; if this is the case, it's called the **null string** (or **empty string**) and is represented by two consecutive quotation marks ("").

The length of a string is simply the number of characters in it.

Operating on Strings

Like numbers, strings can be input, processed and output.

Any data stored in a string variable is treated as text. Therefore, if you declare a variable named ItemNumber as a String and then assign ItemNumber = "12" you cannot do any mathematical operations on that variable since it is not a Number.

Many programming languages include at least one string operator, **concatenation**, which takes two strings and joins them to produce a string result. The symbol used to concatenate two strings is the plus sign, +.

For example, if String1 = "Part1" and String2 = "Time", then the statement:

```
NewString = String1 + String2
```

assigns the string "PartTime" to the string variable NewString. In other words, the value stored in the variable String1 is concatenated with the value stored in String2 to form the new string.

Pseudocode

Once we have identified the various tasks a program needs to accomplish, we must fill in the details of the program design. For each module, we can provide specific instructions to perform that task. We supply this detail using **pseudocode.**

Pseudocode uses short, English-like phrases to describe the outline of a program. It's not actual code from any specific programming language, but sometimes it strongly resembles actual code. It is common to start with a rough pseudocode outline for each module and then refine the pseudocode to provide more and more detail.

```
Input Data module

    Input ItemName, OriginalPrice, DiscountPrice

Perform Calculations module

    Compute SalePrice
    Compute TotalPrice

Output Results module

    Output the input data and TotalPrice
```

Note: Although a good practice, in real-world program development, this step is sometimes skipped for time constraints, but is an excellent and important technique in helping write out complex program operations.

Documenting a Program

All programs should include annotation that explains the purpose of portions of code with the program itself. This kind of annotation is known as **internal documentation** and is made up of comments. A **comment** is text inserted into the program for explanatory purposes, but ignored by the computer when the program is run.

Types of Errors

When a test run of your program turns up problems, we must debug it, which means we must locate and eliminate the errors. This may be relatively easy or very difficult, depending on the type of error and the debugging skill of the programmer.

The two fundamental types of errors that can arise in coding a program are **syntax errors** and **logic errors**.

Types of Errors: Syntax Errors

A syntax error is a violation of the programming language's rules for creating valid statements. It can be caused by misspelling a keyword or by omitting a required punctuation mark. Syntax errors are normally detected by the language software, either when the invalid statement is typed or when the program is executed in the environment it will run in (the browser for our class).

Sometimes though, the syntax error is found in one place that actually is caused by another type of error somewhere else in the code. In this case, it takes programming skill to analyze an error message, find the place where the error originates, and correct it.

Types of Error: Logic Errors

A logic error results from failing to use the proper combination of statements to accomplish a certain task. It may occur due to faulty analysis, faulty design or failure to code the program properly.

Logic errors often cause the program to fail to proceed beyond a certain point (i.e. crash or hang or freeze) or to give incorrect results. Unlike syntax errors, logic errors are not detected by the programming language software. Usually they can be found only by running the program with a sufficient variety of test data. Extensive testing is the best way to ensure that a program's logic is sound.

Note: Although no special software is required (a text editor is all that is needed) to code and develop in JavaScript, I will be giving students some options on what they can use.

JavaScript Software

JavaScript, which is embedded in an HTML page, can be created with software tools such as HTML text editors (Notepad++ , TextWrangler, Komodo IDE, Codelobster) or more commercial IDE tools such as WebStorm, Adobe Dreamweaver or Microsoft Visual Studio.

There are sophisticated open source tools such as Aptana and Netbeans that can be used for multiple programming languages. Many of these tools can have a steep learning curve and are beyond the scope of this class.

If you wish to try out one of these tools, I have an educational license for WebStorm for students of eClasses.org (email me about it).

Note: I will be covering browser-based tools that you can use for syntax error and debugging in a later lesson.

JavaScript Syntax

Inserting JavaScript into a web page is much like inserting any other HTML content; you use tags to mark the start and end of your script code. The tag used to do this is the <script></script> tag.

This tells the browser that the following chunk of text, bounded by the closing </script> tag, is not HTML to be displayed, but rather script code to be processed.

The chunk of code surrounded by the <script> and </script> tags is called a **script block.**

Basically, when the browser spots <script> tags, instead of trying to display the contained text to the user, it uses the browser’s built-in JavaScript interpreter to run the code’s instruction.

You can put the <script> tags inside the header (between the <head> and </head> tags), or inside the body (between the <body> and </body> tags) of the HTML page.

The <script> tag has a number of attributes, but the most important one is the type attribute. You can tell the browser which scripting language to expect so that it knows how to process that language. Your opening script tag will look like this:

```
<script type=”text/javascript”>
```

Including the type attribute is good practice, but within a web page it can be left off. Browsers such as IE, Firefox, Safari, and Chrome JavaScript as their default script language.

This means that if the browser encounters a <script> tag with no type attribute set, it assumes that the script block is written in JavaScript.

However, use of the type attribute is specified as mandatory by W3C (the World Wide Web Consortium), which sets the standards for HTML and XHTML.

```
document.write(“I love JavaScript!”);
```

The preceding line of code is an example of a JavaScript **statement**. Every line of code between the <script> and </script> tags is called a statement, although some statements may run on to more than one line.

You’ll also see that there’s a semicolon (;) at the end of the line. You use a semicolon in JavaScript to indicate the end of a statement.

Finally, to tell the browser to stop interpreting your text as JavaScript and start interpreting it as HTML, you use the script close tag:

```
</script>
```

You’ve now looked at how the code works, but you haven’t looked at the order in which it works.

When the browser loads in the web page, the browser goes through it, rendering it tag by tag from top to bottom of the page. This process is called **parsing.**

The web browser starts at the top of the page and works its way down to the bottom of the page. When it comes to the JavaScript code, it will write the string value: “I love JavaScript! “ to the page.

Reference: https://developer.mozilla.org/en-US/docs/Web/API/document.write

```
<html>
<body>

<script>

document.write("I love JavaScript!");

// document.write displays text on a Web Page <--- this is a comment

</script>

</body>
</html>
```

JavaScript Syntax: alert()

The alert() function function enables you to alert or inform the user about something by displaying a message box. The message to be given in the message box is specified inside the parentheses of the alert() function and is known as the function’s parameter.

The message box displayed by the alert() function is **modal** which means that the message box won’t go away until the user closes it by clicking the OK button. In fact, parsing of the page stops at the line where the alert() function is used and doesn’t restart until the user closes the message box.

Reference: https://developer.mozilla.org/en-US/docs/Web/API/window.alert

JavaScript Syntax: Case Sensitive

Everything is case-sensitive: variables, function names, and operators are all case-sensitive.

Note: A variable named *test* is different from a variable named *Test*.

JavaScript Syntax: Comments

A single - line comment begins with two forward - slash characters, such as this:

```
//single line comment
```

A block comment begins with a forward - slash and asterisk (/*), and ends with the opposite (*/), as in this example:

```
/*
* This is a multi-line
* Comment
*/
```

Note: Even though the second and third lines contain an asterisk, these are not necessary and are added purely for readability (format preferred in enterprise applications).

JavaScript Syntax: Variables

Every variable is simply a named placeholder for a value. To define a variable, use the var operator followed by the variable name, like this:

```
var message;
```

This code defines a variable named *message* that can be used to hold any value.

```
var message = “hi”;
```

Here, message is defined to hold a string value of “hi”. Doing this initialization doesn’t mark the variable as being a string type; it is simply the assignment of a value to the variable.

It is still possible to not only change the value stored in the variable, but also to change the type of value, such as this:

```
var message = “hi”;
message = 100; //legal, but not recommended
```

In this example, the variable message is first defined as having the string value “hi” and then overwritten with the numeric value 100. Though it’s not recommended to switch the data type that a variable works with, it is completely valid.

Also, you can’t use certain names and characters for your variable names. Names you can’t use are called reserved words.

Reserved words are words that JavaScript keeps for its own use, for example the word var or the word with. Certain characters are also forbidden in variable names; for example, the ampersand (&) and the percent sign (%).

You are allowed to use numbers in your variable names, but the names must not begin with numbers. So 101myVariable is not okay, but myVariable101 is.

Reference:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Values,_variables,_and_literals

JavaScript Syntax: Numerical Data

Numerical data come in two forms:

1. Whole numbers, such as 145, which are also known as *integers*. These numbers can be positive or negative and can span a very wide range in JavaScript: –253 to 253.

2. Fractional numbers, such as 1.234, which are also known as *floating-point* numbers. Like integers, they can be positive or negative, and they also have a massive range.

JavaScript Syntax: Text Data

Another term for one or more characters of text is a *string*. You tell JavaScript that text is to be treated as text and not as code simply by enclosing it inside quote marks (“). For example, “Hello World” and “A” are examples of strings that JavaScript will recognize.

You can also use the single quote marks (‘), so ‘Hello World’ and ‘A’ are also examples of strings that JavaScript will recognize.

However, you must end the string with the same quote mark that you started it with. Therefore, “A’ is not a valid JavaScript string, and neither is ‘Hello World”.