Web Front-End Development
Week 3: Program Design and Debugging

Although humans can perform somewhat complex tasks without thinking about them too much, computers need to follow a precise set of instructions to reach a specific outcome.
As our programs get more complicated we need to plan out what we want our programs to do and the steps required before jumping in and starting to create the script just like would for any complex task we don't yet know the steps for.

When you're going somewhere you've been before you don't need directions. If you're going somewhere new you'll need some form of directions to get there. Same for cooking something you're familiar with vs. something new where you'll want a recipe to follow.

It's the same with programming. If you're writing a simple program that's the same or similar to ones you've written before, you might be able to do it with minimal planning.
But for anything that's not simple you'll save yourself time in the long run by taking the time to think through and design your program before creating it.

Program Design Steps

1. Identify the Goal
Think abstractly about the problem you're trying to solve without worrying about the details of a solution.

2. Understand the problem
Make sure you understand the problem in its entirety

3. Identify ways to solve the problem
Identify different ways to approach the problem
- Usually more than one correct model
- There are incorrect models

4. Select the best way to solve the problem
Pick the model that you decide is the "best" way to solve the problem
- This could be based on many factors including which approach you're most comfortable with based on previous experience, resources, limitations, etc.

5. Develop the algorithm
What do you think of when you hear the word algorithm?
- A procedure for accomplishing a task
- A systematic method for producing a specified result
- A series of instructions that produce a solution to a given problem
- Algorithms we deal with every day
  - Recipes
  - Directions
  - Nominating procedures
  - Others?
- An algorithm for a computer to follow must be precise
  - Inputs specified

    - Outputs specified
    - Definiteness
    - Effectiveness
    - Finiteness

Think about a task or activity that you perform in your lives. How would you create an algorithm that describes this activity?

- Think about the activity abstractly
- Generalize the high-level tasks needed to reach your goal
- Think through the steps that describes this activity
- Gradually add detail to the solution until you have a step by step solution
    - drill down from a flowchart, to a list of steps, until you're ready to start creating your program.
- Make sure your algorithm covers all possibilities
- List any assumptions

<u>Pseudocode</u>
Pseudocode is English-like statements of an algorithm
Requires less precision than a formal programming language
An algorithm written in pseudocode should be independent of, but easily translated into, *any* formal programming language.
This is not a waste of time – it lets you focus on the logic of the algorithm without being distracted by details of language syntax.
Pseudocode is also used in job code interviews as employers want to see how you approach a problem, so this is a useful skill to build.

<u>6. Iteratively create the program</u>
Only when you have figured out your program's algorithm are you ready to turn it into code

Convert your pseudocode to code incrementally. Write and test small sections at a time.
Get each section working before you move on.
DON'T write your whole program at once. This makes it harder to find your errors.
Save new versions of your file so you always have the previous version that worked.
Testing helps locate problems to make sure the program works correctly.
Debugging is the process of finding bugs in a program and fixing them.

<u>7. Evaluate/test your solution</u>
You test a program to make sure it runs properly.
The goal of testing is to find, diagnose, and fix situations where the program doesn't work as intended.
To test a program you need to try all possible combinations of input data and user actions.
User Interface Testing

- Check the design and controls

You will need different types of test data, or test cases, to check that your program works in different scenarios.

- Test the program with valid data
    - The data you would expect a user to enter
    - Test the boundaries
- Test the program with invalid data
    - Test unexpected user actions
    - Test that the program handles various data, including "incorrect" data.

- Test at the boundary values: make sure your program works at the extremes.

Test all branches in conditional statements, and variables before and after loop execution.

There are also test frameworks available for JavaScript to help create test suites when you're working on larger scale JavaScript programs.

JavaScript being a loosely typed language reduces its ability to catch some errors early on. Some errors might be caught later or not at all.
JavaScript can be made a little stricter by using strict mode by adding at the top of a file or function "use strict".
There are also many JavaScript dialects, such as TypeScript, that build on JavaScript and add types so that they are checked.

Bugs
https://thenextweb.com/shareables/2013/09/18/the-very-first-computer-bug/#.tnw_QrhYBAvZ
Once you find problems during testing you need to figure out what's causing those problems, referred to as bugs.
On September 9, 1945, engineers working on the Harvard Mark II computer found a moth between the relays. In those days computers filled (large) rooms and the warmth of the internal components attracted moths, flies and other flying creatures. Those creatures then shortened circuits and caused the computer to malfunction. They taped the insect in their log book and labeled it "first actual case of bug being found."
The term 'bugs in a computer' had been used before, but after Grace Hopper wrote in her diary "first actual case of bug being found" the term became really popular, and that's why we are still using the terms "bug" and "debug" today.

Debugging
A well-designed program will avoid many errors, but not all.

- To debug the problem you must take a logical approach to figure out what's causing the problem.
- The process of debugging consists of the following steps:
  1. Determine that a problem exists
  2. Identify the type of problem
  3. Locate where the problem is in the code
  4. Identify what the exact problem is
  5. Correct the incorrect code

Errors
https://replit.com/@aileenjp/JavaScript-exception-handling
Some errors occur when your program communicates with the outside world such as another system or a user. Some conditions to consider:

- No data returned
- Unexpected data
- Unexpected user actions

You should try to handle as many of these conditions as possible.
Example:

- If my function returns NaN the program will go on to use that and the error will continue
- I can test for that and return something else like null but then the calling program should check for null

Exception Handling

When a program runs into a problem it can raise an exception. When an exception is thrown you can then catch the exception and handle it.

- The *throw* keyword is used to raise an exception
- The *try* keyword is used to call a function that might throw an exception
- The *catch* keyword is used to wrap a block of code to handle the exception
  - The error passed to the catch block has the scope of the catch block

When an error is thrown the function exits and program control returns to the calling program.
If any statement in the try block throws an exception, control goes to the catch block.
If the statements in the try block are successful the catch block is skipped.
JavaScript has an Error type that provides information about the error thrown. It has two properties:

- name – there are 6 different types of errors or you can create a custom error type https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#error_types
- message – a string you can pass when you throw the error

This is the standard way of handling errors in JavaScript and many other languages.

Syntax

- Syntax errors are a violation of the grammatical rules in a programming language.
- Common syntax errors
  - Spelling, spelling, and spelling. Check keywords and identifiers you've created
  - Forgetting or misplacing an opening or closing parenthesis, bracket, brace, or semicolon.
  - Forgetting or misplacing an opening or closing quote mark or not using the same type of quote in a pair.
  - Using a reserved word.
  - Make sure you're testing for equality using == or === and not assignment =
  - Make sure your test condition in in parens () and evaluates to true or false
  - Make sure you have { } around the body of if, else, and loop blocks
  - Remember that functions have their own scope — you can't access a variable value set inside a function from outside the function.
- If your editor is showing an error fix it before you even try to run it
- Often just 1 typo can cause the entire page to not load and all you get is a blank white page.
- The JavaScript engine won't run the application until all of the syntax errors have been corrected.

Logic

- Logic errors produce the wrong results.
  - These are the hardest to fix because the problem lies either in the original algorithm or in the way the algorithm has been translated into JavaScript.
- Good algorithm development practices is the best defensive against logic errors.
- A walk-through of each section is a good way to check your algorithm and logic.
- Desk-checking by hand: tracing the steps that the program goes through and tracking all variable values by hand.
- Trace the program execution using alert/write statements
  - Track the values of variables
  - Track what part of the code is being run
- Print the values before and after the variable is changed such as in an assignment.

- Examine the incorrect assignment -- the error is either in the expression on the right side or in one of its variables.

Conditional statements

- Trace the execution path at each conditional statement.
- Test both the body of an if statement and the else statement.
- Display critical variables before and after if and else statements.

Loops

- Common loop errors that should be tested for include:
    - Off-by-one errors
    - Infinite loops
    - Never-executed loops
- Test loops with different data
    - force the test expression to be false initially
    - force the test expression to be true initially
- Always check the *loop boundaries* to prevent *off-by-one* errors. The loop boundaries occur at the initial and final value of the loop control variable.
- Display critical variables before and after ***while*** and ***for*** blocks. Within the loop, display the variables before or after the loop counter has been changed.

Functions

- Make sure you're calling the function you wrote.
- Remember that functions have their own scope — you can't access a variable value set inside a function from outside the function.
- Trace the beginning and end of every function call.
- Trace the variables at the beginning of each function.
- Display all parameters before and after each function entry.
- Display all parameters and return values before and after each function exit.
- Make sure you don't have any code after a return statement as it will never run

Correcting Errors
Once you've identified the location of the error, you need to figure out what the problem is.
Don't randomly guess and change a lot of code - you could just be introducing new errors.
Save your file as different versions so you always have the previous version that worked.
Each time you fix a bug you must test again to make sure you didn't break something else.

There are some tools that can help you find your errors.

Linters
JavaScript linters point out errors and warnings before you run your code.
ESLint https://eslint.org/ and JSHint https://jshint.com/ are two popular ones and JSHint can even be installed as a plugin for most editors https://jshint.com/install/

Browser Developer Tools
The browser developer tools make it much easier to debug and find errors. It's really worth taking the time to get comfortable with them as they'll save you hours in the future.

- JavaScript/Web/Error console

    - Tells you when there is a problem
    - Displays the line where the problem was caught
        - Look at that line number and go back up
    - Gives an error message
    - Can also type in the console to quickly test code
    - console.log() debug statements are written here
    - The web console shows the DOM node sheet for the page.
        - displays the html and css properties for each element type
        - you can unselect css rules or change values to quickly test
- Debugger
    - Use breakpoints to pause your script and check variable values at that point
    - Step through the code line by line
    - Lets you monitor what your program is doing
    - The Watch List lets you see what's going on in your script, such as the values of variables
    - To add a breakpoint click to the left of the line number and a red bullet will appear indicating a breakpoint.
    - Reload the page to run your program.
    - The debugger will stop the program before your breakpoint.

<u>Chrome Developer Tools</u>

- View | Developer | Developer tools (or right click on page)
- View | Developer | Inspect Elements
    - shows the DOM for the current page
    - or right-click Inspect
- View | Developer | JavaScript Console
    - Console tab shows all errors or warnings

<u>Firefox Developer Tools</u>

- Tools | Web Developer | Inspector (menu might be different on a pc) or right-click Inspect Element
    - shows the DOM for the current page
- Tools | Web Developer | Web Console
    - Console records all errors or warnings in a browser session
        - JS and Logging will show JavaScript errors
- Tools | Web Developer | Debugger
- May be kept open during a browser session
- Firefox often gives the most understandable error messages

<u>Safari Developer Tools</u>

- Safari | Preferences | Advanced
    - Check the "Show Develop menu in menu bar" box
    - Restart Safari to see Develop menu
- Develop | Show Web Inspector
    - Elements tab shows the DOM
- Develop | Show JavaScript Console
    - Console shows JavaScript errors