# ICA06 – Loan Repayment Calculator

Earlier in the semester, we spent some time talking about troubleshooting and debugging code. We talked about the differences between Syntax Errors and Logic Errors.

But since then, we have introduced functions, conditional statements, and loops. That means that our programs are getting more complicated. So, our debugging challenges are getting more complicated too.

That brings us to today … we are going to spend time in this activity doing some recap of the debugging tools and techniques we have seen so far, and then apply them to a simple JavaScript application: a loan repayment calculator.

After you have found all the bugs, you will be expected to add two features to the calculator on your own, without instruction or guidance.

**SCENARIO:**

This program should take a loan amount, an interest rate, and a loan duration expressed as a number of years as input.

The code should then generate a text message with the following format:

"Your estimated monthly payment is: $XXX.XX"

(Here, obviously, $XXX.XX is a calculated value!)

The code should not allow bad / nonsensical input.  For example:

* the loan amount must be a positive whole number.
* the interest rate must me in bounds (greater that zero and less than 1 (that is, 100%).
* the number of years must be a positive whole number.

The start file contains a first attempt at creating such a calculator.  However, there are 10 errors in the code, and you must find them all and fix them.

When you find an error, make a comment next to it. Identify if you think the error was a "logic error" or a "syntax error" and then fix the error.

For example:

let x = 100 / 0;   //logic error - you may not divide by zero.

// if you do, the result is "undefined"

In the “On your own” section you will be instructed to add new features to this application.

**TROUBLE SHOOTING / DEBUGGING TIPS:**

It won't do you much good to attempt to memorize the table below. It is provided here as list of things to try and think about as you work. When you encounter a JavaScript coding problem you don't know how to solve, start at the top of the list and work your way down.

|  |  |
| --- | --- |
| Declare Strict Mode | This has been done for you. You will see this statement inside your <script> tag, at the beginning of your script:"use strict";Using strict mode is a best practice in industry. Declaring strict mode forces you to declare your variables properly. Explicit variable declaration is important, and it is required in many other programing languages.You can read more about strict mode here:<https://www.w3schools.com/js/js_strict.asp>Bottom line: the solution to a coding problem should not be to turn off / remove the "use strict" statement. That's like "fixing" a smoke detector by taking the battery out.  |
| Look for “Live Sematic Errors” in VS Code | By now you know that VS Code will highlight live semantic errors with a red, wavey underscore: like this. This won't catch all your coding errors, but it is a good place to start! |
| Look in the console log for error messages | Many errors will only occur when you attempt to run your code in a browser. You must look at the console log, found in the Web Developer tools, if you want to find / fix them. Read each error message in its entirety and Google any words and phrases you don't understand. Pay attention to the line numbers as well.If you don't do that much, you really are "flying blind". |
| Comment your code | Taking a little time to comment your code can help you slow down and think through the problem. It is something you should be doing anyway! Take a moment and add some comments that describe in English what you are attempting to do with code. Add comments to each line of code, or at least each bock of code, indicating what you are doing. |
| Check opening / closing delimiters | Earlier in the semester, you were taught that VS Code will automatically highlight a closing delimiter when you click on the opening delimiter. This is true for parenthesis, square brackets, curly brackets, angle brackets, and HTML tags. If you suspect that an incorrect closure might be the problem (for example: you got an "Unexpected end of input" error message in the console log) then you need to find the opening / closing combination that is incorrect.There's no substitute for being systematic here. Start at high / outer level in your code, click on an opening delimiter, check if its corresponding partner is where you expected it to be, and work your way inward.Many seemingly “impossible” programming problems can be solved in this way. |
| Check for spelling inconsistencies | Here's a hot tip - VS Code will highlight matching pieces of text for you. So, if you click on a variable named noun1 then all occurrences of the text noun1 will be highlighted. VS Code's comparison of text here is case ***insensitive*** (so, noun1 and Noun1 would be considered the same by VS Code, but different when the code is interpreted by the browser!)But, even with this limitation, seeing just a little bit of highlighting can help you find spelling inconsistencies and other similar errors. |
| Use breakpoints and step through code | You have already seen how using the Web Developer tools in Chrome can be used to step through code as it runs. When you do this, you have the option of hovering your mouse over variable names to inspect their current values.This is a great technique, and it should be attempted before the final two items on this list. By now you know that you can step through code, line by line, using the button illustrated below. Note that you can also step out of the current function with the "up" button shown here.Complete this functionNext line of code  |
| Add console.log() statements | If you are having trouble getting a sense of how the code is being processed, or when certain events are taking place, strategically add some console.log() statements in to your code.Then open your code in the browser, watch the console log, and test your code.The issue with this approach is that you need to edit your code to add in the console.log statements. When you do this you risk creating new error in addition to the original one. Also, you should remove or "comment out" these statements when you are done. |
| Add alert() statements | ***LAST RESORT:*** In addition to console.log() statements, you can also add JavaScript alert commands to your script. Like this: alert("Hello World");When an alert statement it will broadcast a Window alert box to the user, and the script will hang until the user responds to it by clicking "OK".This is a very "old school" technique for determining how the program is flowing and the values of variables. The command alert(x); will display the value of x. It is unacceptable to leave alert statements in finished assignments, or (worse still!) production code. So, if you do employ the sketchy technique, you must clean up after yourself when you are done. |

Ready? Let's get started…

**INSTRUCTIONS:**

## Getting started

1. Retrieve the ica06\_loanrepayment.zip provided by your instructor.
2. Extract the code into your mis2402workspace and open the index.html file in Visual Studio Code.
3. Inspect the file. Read the comments in the file. Note the following …

## About the calculation

1. As mentioned previously, this calculator estimates the monthly payment on a loan. It assumes that you are making payments on an amortized loan. Here's the formula it uses:

$$a ÷\left(\left(1+r\right)^{m}-1\right) ÷ r\left(1+r\right)^{m}=p$$

Where:
*a* is the loan amount

*r* is the monthly contribution to the annual interest rate (the annual interest divided by 12)

*m* is the number of monthly payments

*p* is the monthly payment amount

This formula came from this source: <https://www.thebalance.com/loan-payment-calculations-315564>

Please note, the above formula is not JavaScript code, it's just math.

## Testing

1. It's always a good idea to have some test values in mind before you start debugging your code. You should test for a combination of ***both*** good and bad inputs.

Here are some good values to test with:

|  |  |  |
| --- | --- | --- |
| Loan Amount: 10000Interest Rate: 0.05Loan Duration : 5Expected Output:Your estimated monthly payment is: $188.71 | Loan Amount: 100000Interest Rate: 0.06Loan Duration : 30Expected Output:Your estimated monthly payment is: $599.55 | Loan Amount: 1Interest Rate: 1Loan Duration : 1Expected Output:Your estimated monthly payment is: $0.13 |

1. Find a classmate to work with. Working in pairs, find the bugs and fix them. Make comments in your code as you go. There are 10 bugs in the start file. Each student must edit his / her own code.
2. Once you have everything working, make it better! Replace the text input tags for interest rate and loan duration with HTML select tags. This limits the values that a user can plug into your calculator, and that can be a good thing! (Fewer options usually means fewer potential errors!)

If you don't know / don't remember the HTML select tag, you'll need to look it up on W3 schools.

When you are all done, your solution should run without error, and it should look like this:



The Interest Rate dropdown list should contain the values .010, .015, .02 … up until 0.100.

(That's 1% up to 10% interest!)

The Loan Duration dropdown list should contain the values 5, 10, 15, 20 … up to 50.

1. Upload your work to ICA06 on canvas.

How will this activity be graded?

Your instructor will look at the comments in your code. You should have identified and corrected all 10 bugs, worth 5 points each.

Your code should use the select tags, and run without error (even if the user enters wrong / incorrect values!) This is worth 50 points.

|  |  |
| --- | --- |
| Task | Points |
| All 10 bugs identified and corrected and commented on. (5 points each) | 50 |
| HTML select tags applied as directed. | 25 |
| Final upload runs without error. | 25 |
| **Total:** | 100 |