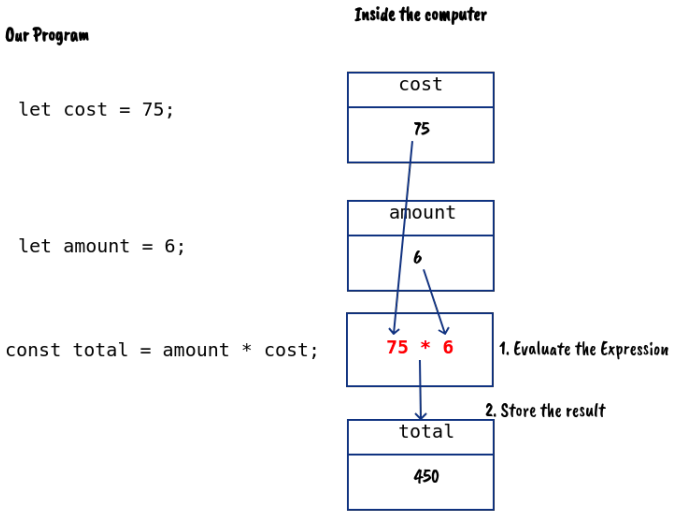
# A minimal toolkit for JavaScript programs

This toolkit for writing JavaScript programs might seem limited but we can write pretty much everything we need to with it. Of course later we will add to the toolkit later but you should become expert with these tools first.

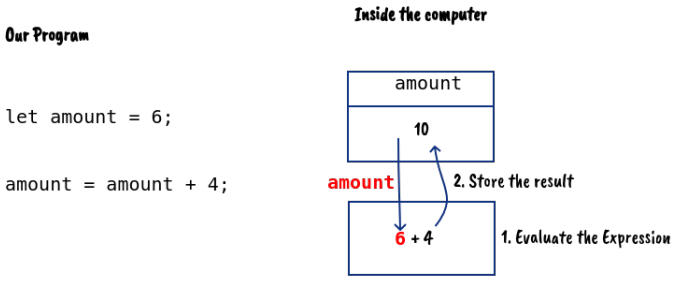
## Contents of our tool box

This is what we will use to write programs:

**Variables:** A variables are essential as places to store out data. We **declare** them with the keyword const or the keyword let. We assign values to variables, those declared with const may only have ONE value and that value never changes, variables declared with let
my have their value updated. Here is a picture of what happens when we use variables:




When we assign a new value to an existing variable, the computer overwrites the previous value:




**Expressions:** Expressions are just things to be **evaluated** and replaced with a **result**. Examples are 6 * 5 which evaluates to the result 30, or amount * cost, which

evaluates to a result depending on the values of the variables amount and cost. We have seen expressions already when we looked at variables. We almost always put expressions on the right hand side of an **assignment statement** in order to store the result of the expression. For example, each of the following statements has an expression on the right hand side of the =:

```
const hourlyRate = 15.5; // we earn 15.50 an hour
const numberOfHours = 40; // we worked 40 hours
const pay = hourlyRate * numberOfHours;
const tax = (pay * 0.2); // 20% of our pay is tax
const earnings = pay - tax; // subtract the tax from our pay
```

**Boolean Expressions:** These are just expressions that evaluate to either true or false. Examples are (amount > 10) or (cost !== 0). The **operator** used in a Boolean expression is a comparison operator such as greater than >, or less than <, or equal to ===, or greater than or equal to >=, etc. No other result is possible for a Boolean expression other than true or false.

**Conditional statement:** Conditional statements use keyword if and require a Boolean expression. They optionally have an else part. Here is an example:

```
let age;
// assign some value to age
if (age >= 18) {
  // do something based on age being 18 or more
} else {
  // do something based on age being less than 18
}
```

**Repetition:** Sometimes we want to repeat some code over and over, for example if we wanted to count from 1 to 10 and print out each number. We need the following tools to do this:

1. A variable, to keep track of where we have got to in the count. We assign a starting value to the variable (in this case the value 1). The variable should have a descriptive name, so we will use count for this example.
2. A while statement and its Boolean expression which evaluates initially to true. In this case while (count <= 10). The code inside the while statement is surrounded by curly brackets.
3. Some code to repeat. In our case we want to repeat logging the current value of count, console.log(count).
4. Finally we need a way to update the count and affect the Boolean expression so it eventually evaluates to false. In this case the update code is count = count + 1 and this update is the last statement inside the while.

When we put these things together we get:

```
let count = 1; // initialise the count
while (count <= 10) { // repeat while Boolean expression is true
  console.log(count);
  count = count + 1; // updates the count and affects the Boolean expression
}
```

You can do exactly the same repetition using a for statement.

```
for (let count = 1; count <= 10; count++) {
  console.log(count);
}
```

This puts all the control of the repetition at the top of the for statement.

**Arrays:** Arrays are like collections of variables. They hold multiple values at locations we can refer to with an index. Arrays use square brackets. Here is an example of how we might use an array:

```
const shoppingList = ['apples', 'milk', 'bread', 'coffee'];
```

Each item in the array has its own **index** with the first index being **zero**. In our array above, shoppingList[0] holds the value 'apples', and shoppingList[2] holds the value 'bread'. The code shoppingList[3] is an expression and can be used anywhere we can put expressions. For example:

```
let index = 3;
const item = shoppingList[index]
```

All arrays have a **length property**, shoppingList.length which tells us how many items are in the array. It is often useful to use repetition to visit each entry in the array. For example, to print our shopping list we would write:

```
let index = 0; // initialise the count
while (index <= shoppingList.length) {  // repeat while Boolean expression is true
  console.log(shoppingList[index]);
  index = index + 1; // updates the count and affects the Boolean expression
}
```

We can add an element to an array using push, so shoppingList.push('soap') would add 'soap' to the end of the array. We can replace existing values in an array by assigning a value to that index. For example:

```
shoppingList[1] = 'oranges';
```

would replace 'milk' with 'oranges'

# Solving problems

Writing a program involves some problem solving. Here is a strategy to help you solve problems.

1. Break the big problem down into a list of smaller, and hopefully simpler problems. In fact to get started, you only need one small problem that very roughly approximates the original problem, and is something you either know how to do, or you believe it can be done fairly easily.

2. Design a solution for your current, simpler problem.

3. Test your solution, make sure it works before continuing.

4. If you haven't yet solved the original problem, think about how to evolve your current solution so that it is one small step closer to the original problem. Go back to step 2

Here is an example. Say we wanted to write a program to think of a number and then to print rows of stars, starting with one star and ending with the number of stars being equal to the number we first thought of. If the number is 5, the output would look like this:

```
*
**
***
****
*****
```

1. First small problem, how do we log a * to the console:

```
1 console.log('*');
```

   Run this, check it works, we are on our way to a solution

2. Second small problem, on any given row, how many stars should we log? We notice that the first row has 1 star, the second row has 2 stars, the third has 3 stars and so on. Conclusion: We need to use a variable from our toolbox and store the number of stars in the row in a variable.

```
1 let starsThisRow = 1;
2 console.log('*');
```

   We run this code to check it works, it still prints only one star, even if we set starsThisrow to a value larger than 1. Note that we did not go on to try and use that variable to actually print a row because we want each step to be as simple as possible.

3. Next small problem. How can we print a single row of stars based on the value of starsThisRow. We decide that we need to repeat logging stars.

```
1 let starsThisRow = 4;
2 let count = 0;
3 while (count < starsThisRow) {
4   console.log('*');
5   count = count + 1;
6 }
```

   When we run it, we find that the output is not quite what we want:

```
*
*
*
*
```

4. We add a new problem, how do we get all the stars on one line. We might check stack overflow for "Is it possible to call console.log() without a newline?", and if we do, we would find the answer: "No, it's not possible. You'll have to keep a string and concatenate if you want it all in one line" (`https://stackoverflow.com/questions/9627646/chrome-javascript-developer-console-is-it-possible-to-call-console-log-withou/32049995`).

5. So, our next step is to create a string, put as many stars in it as we need and then log that string.

```
let starsThisRow = 4;
let count = 0;
let row = "";
while (count < starsThisRow) {
  row = row + '*';
  count = count + 1;
}
console.log(row);
```

We run it and it works, we get the output:

****

6. Next problem, how do we repeat this to get the number of lines that we want. We can think of the code above as **printOneRow** and if we repeated it, each time adding one to starsThisRow, we would print one more star each time we repeated that code. Here is our solution:

```
let numberOfRows = 10
let starsThisRow = 1;
while (starsThisRow <= numberOfRows) {
  // print one row
  let count = 0;
  // build our string of stars
  let row = "";
  while (count < starsThisRow) {
    row = row + '*';
    count = count + 1;
  }
  // log the stars to the console
  console.log(row);
  // update for the next row
  starsThisRow = starsThisRow + 1;
}
```

Notice that we solved this problem in several steps, each step required a bit of thought rather than immediately writing code.
Notice also that some steps don't actually result in a program that does anything different to the previous step, for example steps 2 and 4 didn't change our code, but did contribute to our understanding of how to solve the problem.
Notice also that the final version has some comments that explain what is going on in the code. The variable names are readable and they tell us what the variable is used for.
It is also possible that we get to a "wrong" solution that we need to fix, as in step 3. Step 3 wasn't a disaster, it did print the right number of stars. It just wasn't what we intended. That sometimes happens when we write some code, especially if we haven't properly thought about it before we write it.

# Exercises

These exercises can be solved using only the tools in the toolkit described above. For the moment, stick to just that set of tools.

1. Solve the same problem above but in much less code. **Hint:** Look at how the solution treats the variable called row. It keeps setting it to an empty string and then adding all the stars for the row. An alternative might be to initialise it to the empty string and add one star to it before we log the value of row. Starting again with this idea might give us the following Step 1. Can you take the necessary steps to turn this into a different solution to the same problem?

```
let row = "";
// print one row
row = row + '*'
console.log(row);
```

2. Can you generate the following pattern, where the number of rows is determined by the value of a variable. Here is an example where the numberOfRows is equal to 10. Look for a pattern in each row. Hint: Two different characters are used when building each row.

```
         *
        **
       ***
      ****
     *****
    ******
   *******
  ********
 *********
**********
```

3. Write a JavaScript for loop that will iterate from 0 to 15. For each iteration, it will check if the current number is odd or even, and display a message to the screen. Sample Output :

```
0 is even
1 is odd
2 is even
```

   Before you start, think about how you decide whether a number is odd or even.

4. There are two arrays with individual values, write a JavaScript program to compute the sum of each individual index value from the given arrays. Sample array :

```
const array1 = [1,0,2,3,4];
const array2 = [3,5,6,7,8,13];
```

   Expected Output :

```
[4, 5, 8, 10, 12, 13]
```

5. Starting with this array:

```
const scores = [73, 35, 40, 68, 67, 91, 42, 48, 39, 55, 21, 95];
```

   Write JavaScript code to log the index of each score that is greater than or equal to 40.

6. Starting with this array:

```
const scores = [73, 35, 68, 67, 91, 42, 48, 55, 21, 95];
```

   Write JavaScript code to log the largest score in the array

7. Starting with this array:

```
const scores = [73, 35, 68, 67, 91, 42, 48, 55, 21, 95];
```

   Write JavaScript code to log the smallest score in the array

8. Starting with this array:

```
const scores = [73, 35, 68, 67, 91, 42, 48, 55, 21, 95];
```

   Write JavaScript code to log the average score. Compute the average by first adding up each individual score to calculate the total and then divide the total by the length of the array.

9. Use the following array with your solution to Exercise 8. The average printed out should be 40.

```
const scores = [40,40,40];
```

10. Use the following array with your solution to Exercise 8. How can you avoid an error when you divide by the length of the array.

```
const scores = [];
```

11. Starting with this array:

```
const scores = [73, 35, 68, 67, 91, 42, 48, 55, 21, 95];
```

    Write JavaScript code to log each score that is within the 10 marks each side of the average (ie less than (average + 10) and greater than (average − 10))

12. Starting with this array:

```
const scores = [73, 35, 68, 67, 20, 91, 42, 48, 55, 21, 95];
```

    Write JavaScript code to move the smallest score to the end of the array. Do this by starting with the first element and compare it with the second element. If the first element is smaller than the second, swap the two values. Next compare the second element with the third element and again swap if the second element is smaller than the third. Keep comparing and swapping until the smallest value has been moved all the way to the end of the array. When you have finished, the original array should look like this with the smallest value, 20, being the last element in the array:

```
[73, 68, 67, 35, 91, 42, 48, 55, 21, 95, 20];
```

13. How would you extend your solution to 13 to repeat this swapping of elements so that the entire array is sorted with the largest element at index 0 and the smallest element at the last index.
14. How would you change your solution to exercise 13 so that the array was sorted in the opposite order with the smallest element in index 0 and the largest element in the last index?
15. Try solving the Javascript-Core-1-Coursework-Week3 extra exercises. You will find them here: `https://github.com/CodeYourFuture/JavaScript-Core-1-Coursework-Week3/tree/main/3-extra`

## Adding functions to our toolkit

Read the lesson that covered functions. You will find it here:
`https://syllabus.codeyourfuture.io/js-core-1/week-1/lesson#functions`

Can you explain what a **parameter** is and why one or more parameters are sometimes required for a function?
Think about a function to multiply a number by itself (a process known as squaring the number). We could write it like this:

```
function square() {
    return 5 * 5;
}

let result = square();
```

It works, but it ONLY works if we want to multiply 5 by itself. The result is ALWAYS 25.

Here is a more powerful version of the same function. Compare it with the version above:

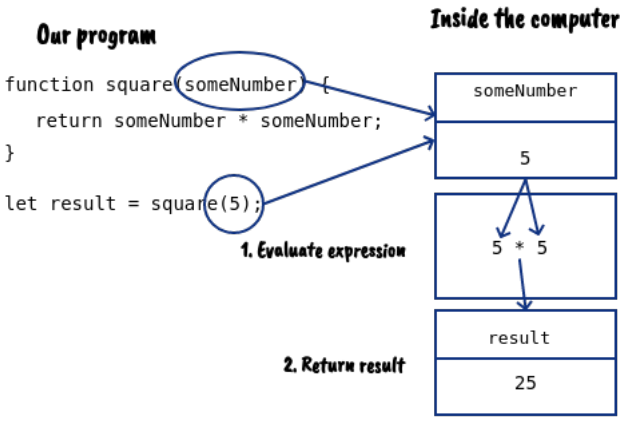
```
function square(someNumber) {
    return someNumber * someNumber;
}

let result = square(5);
```

We have added a **parameter** called someNumber. This is like a variable that is only used inside the function. It doesn't exist outside the function. Like all variables it needs to be assigned a value before we can use it. That comes from the place where we **call** the function. In the example above, we write

```
let result = square(5);
```

The effect is to call the function called square, passing it the value 5. The computer will immediately assign the value 5 to someNumber as the first step of executing the function. This parameter/variable named someNumber is then used in the multiplication.

This is what happens when we use the function with its parameter and pass a value to it:




The reason that the version with the parameter is more powerful is because we can **reuse** the same function over and over and pass different values to it. That really means that it is more powerful because it will square any value that we ask it to:

```
let square5 = square(5);
let square9 = square(9);
let square238 = square(238);
```

**Why do we need a parameter?** We need a parameter whenever the function requires some additional data in order to perform its task. In the example above where the function multiplies some number by itself, it needs to know the value of that number. When we call the function we provide the value.
**Returning a value** In the example above, the function **returned** the result of the calculation using the return statement. Whenever a function has a return statement, there is always a corresponding assignment statement when we call the function:

```
let square238 = square(238);
```

The function call becomes an expression to be calculated and the result of the function needs to be stored into a variable. If we don't do that, the returned value is lost.

# More Exercises

1. Write a function called findLargest that will find and return the largest value in an array of numbers (see your solution to Exercise 6 above). Before you start, think about the parameter you will provide to this function. Here are two different ways your function can be called:

   ```
   let large1 = findLargest([5, 8, 3, 1, 0]);
   let large2 = findlargest([10, 10, 9, 8, 11]);
   ```

   The first version should return 8 and the second one should return 11. test your function to check the correct value is returned.

2. Write a function called drawTriangle that will produce the triangle output in Exercise 2 from the set of exercises above. Use a parameter that will determine how many rows of stars to draw. If you call drawTriangle(5), it should draw the following output:

```
    *
   **
  ***
 ****
*****
```

Does this function need to return anything?

3. Write a JavaScript function called reverseIt that reverses an integer number and returns the reversed value. You need to use a parameter (the function needs to know the number to be reversed). You also need a return statement. Here is a test to see if your function works:

```
let reversed = reverseIt(36478);
console.log(reversed); // expected output 87463
console.log(typeof reversed); // expected output: "number"
```

This sounds easy but it's not as easy as it looks. Before you write any code, look at the hints below and work out a plan for reversing the number. When you have a plan, write the function and test it using the test shown above.

**Hints**:

- You can turn a number into a string (see `https://www.w3schools.com/jsref/jsref_tostring_number.asp`).
- All strings have a length (see `https://www.w3schools.com/jsref/jsref_length_string.asp`)
- You can use the charAt string function to get each character of a string in any order you want. (See `https://www.w3schools.com/jsref/jsref_charat.asp`).
- You must return a number. You can convert a string into an integer, see `https://www.w3schools.com/jsref/jsref_parseint.asp`