SD400: Problem Solving

Lesson 6: Functions

Wholeness

- Today we are going to look at functions, which let us take a part of our program and cleanly and clearly make it “it’s own thing”
- By encapsulating code in a function its variables become separate and can no longer be interfered with.
- Unity in Diversity – the same steps, from any location.

Lesson Objectives

- What are functions
- How to create functions
- Local and Outer variables
- Parameters
- Return value of a function
- Calling functions

Functions

- Quite often we need to perform a similar action in many places of the script.
- For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else.
- Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.
- We’ve already seen examples of built-in functions,
  like alert(message), prompt(message, default) and confirm(question).
  But we can create functions of our own as well.

Function Declaration

- To create a function we can use a *function declaration*.

```
function showMessage() {
alert( 'Hello everyone!' );
}
```

- The function keyword goes first, then goes the *name of the function*, then a list of *parameters* between the parentheses (comma-separated, empty in the example above, we'll see examples later) and finally the code of the function, also named "the function body", between curly braces.

```
function name(parameter1, parameter2, ... parameterN) {
// body }
```

Calling a function

- Our new function can be called by its name: showMessage().

```
function showMessage() {
alert( 'Hello everyone!' ); }
showMessage();
showMessage();
```

- The call showMessage() executes the code of the function. Here we will see the message two times.
- This example clearly demonstrates one of the main purposes of functions: to avoid code duplication.
- If we ever need to change the message or the way it is shown, it’s enough to modify the code in one place: the function which outputs it.

Local variables

- A variable declared inside a function is only visible inside that function.
- For example:

```
1  function showMessage() {
2    let message = "Hello, I'm JavaScript!"; // local variable
3
4    alert( message );
5  }
6
7  showMessage(); // Hello, I'm JavaScript!
8
9  alert( message ); // <-- Error! The variable is local to the function
```

Outer variables

- A function can access an outer variable as well, for example:

```
let userName = 'John';

function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}

showMessage(); // Hello, John
```

Outer variables

- The function has full access to the outer variable. It can modify it as well.
- The outer variable is only used if there's no local one.

```
let userName = 'John';

function showMessage() {
  userName = "Bob"; // (1) changed the outer variable

  let message = 'Hello, ' + userName;
  alert(message);
}

alert( userName ); // John before the function call

showMessage();

alert( userName ); // Bob, the value was modified by the function
```

Outer variables

- If a same-named variable is declared inside the function then it *shadows* the outer one. For instance, in the code below the function uses the local userName. The outer one is ignored:

```
let userName = 'John';

function showMessage() {
  let userName = "Bob"; // declare a local variable

  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

// the function will create and use its own userName
showMessage();

alert( userName ); // John, unchanged, the function did not access the outer
```

Global variables

- Variables declared outside of any function, such as the outer userName in the previous slide, are called *global*.
- Global variables are visible from any function (unless shadowed by locals).
- It’s a good practice to minimize the use of global variables. Modern code has few or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

Main Point 1

- Functions are a set of instructions (a flowchart) which can be called upon from your flowchart. By using functions we can organize our code so that each part of it (each function) is short and clear.
- Purification leads to progress

Parameters

- We can pass arbitrary data to functions using parameters.
- In the example below, the function has two parameters: from and text.

```
1  function showMessage(from, text) { // parameters: from, text
2    alert(from + ': ' + text);
3  }
4
5  showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
6  showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

- When the function is called in lines (*) and (**), the given values are copied to local variables from and text. Then the function uses them.

Parameters

- Here's one more example: we have a variable from and pass it to the function. Please note: the function changes from, but the change is not seen outside, because a function always gets a copy of the value:

```
1  function showMessage(from, text) {
2
3    from = '*' + from + '*'; // make "from" look nicer
4
5    alert( from + ': ' + text );
6  }
7
8  let from = "Ann";
9
10 showMessage(from, "Hello"); // *Ann*: Hello
11
12 // the value of "from" is the same, the function modified a local copy
13 alert( from ); // Ann
```

Parameters

- When a value is passed as a function parameter, it’s also called an *argument*.
- In other words, to put these terms straight:
- A parameter is the variable listed inside the parentheses in the function declaration (it’s a declaration time term).
- An argument is the value that is passed to the function when it is called (it’s a call time term).
- We declare functions listing their parameters, then call them passing arguments.
- In the example in the previous slide, one might say: "the function showMessage is declared with two parameters, then called with two arguments: from and "Hello" ".

Default values

- If a function is called, but an argument is not provided, then the corresponding value becomes undefined.
- For instance, the aforementioned function showMessage(from, text) can be called with a single argument: `showMessage("Ann");`

- That's not an error. Such a call would output "*Ann*: undefined". As the value for text isn't passed, it becomes undefined.

Default values

- We can specify the so-called “default” (to use if omitted) value for a parameter in the function declaration, using =:

```
function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
}

showMessage("Ann"); // Ann: no text given
```

- Now if the text parameter is not passed, it will get the value "no text given".
- The default value also jumps in if the parameter exists, but strictly equals undefined, like this:

```
showMessage("Ann", undefined); // Ann: no text given
```

Calling a function as default

- Here "no text given" is a string, but it can be a more complex expression, which is only evaluated and assigned if the parameter is missing. So, this is also possible:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() only executed if no text given
  // its result becomes the value of text
}
```

Alternative default parameters

- Sometimes it makes sense to assign default values for parameters at a later stage after the function declaration.
- We can check if the parameter is passed during the function execution, by comparing it with undefined:

```
1  function showMessage(text) {
2    // ...
3
4    if (text === undefined) { // if the parameter is missing
5      text = 'empty message';
6    }
7
8    alert(text);
9  }
10
11 showMessage(); // empty message
```

Operator ??

Modern JavaScript engines support the nullish coalescing operator ??, it's better when most falsy values, such as 0, should be considered "normal":

```
function showCount(count) {
  // if count is undefined or null, show "unknown"
  alert(count ?? "unknown");
}

showCount(0); // 0
showCount(null); // unknown
showCount(); // unknown
```

Returning a value

- A function can return a value back into the calling code as the result.
- The simplest example would be a function that sums two values:

```
function sum(a, b) {
  return a + b;
}

let result = sum(1, 2);
alert( result ); // 3
```

- The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).

Returning a value

- There may be many occurrences of return in a single function. For instance:

```
1  function checkAge(age) {
2    if (age >= 18) {
3      return true;
4    } else {
5      return confirm('Do you have permission from your parents?');
6    }
7  }
8
9  let age = prompt('How old are you?', 18);
10
11 if ( checkAge(age) ) {
12   alert( 'Access granted' );
13 } else {
14   alert( 'Access denied' );
15 }
```

Calling an empty return

- It is possible to use return without a value. That causes the function to exit immediately.

In the code below, if checkAge(age) returns false, then showMovie won't proceed to the alert.

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Showing you the movie" ); // (*)
  // ...
}
```

Without a return value

- **A function with an empty return or without it returns undefined**
- If a function does not return a value, it is the same as if it returns undefined:
- An empty return is also the same as return undefined:

```
1  function doNothing() { /* empty */ }
2
3  alert( doNothing() === undefined ); // true
```

```
1  function doNothing() {
2    return;
3  }
4
5  alert( doNothing() === undefined ); // true
```

Naming a function

- Functions are actions. So their name is usually a verb. It should be brief, as accurate as possible and describe what the function does, so that someone reading the code gets an indication of what the function does.
- It is a widespread practice to start a function with a verbal prefix which vaguely describes the action. There must be an agreement within the team on the meaning of the prefixes.
- For instance, functions that start with "show" usually show something.
- Function starting with…
    - "get…" – return a value,
    - "calc…" – calculate something,
    - "create…" – create something,
    - "check…" – check something and return a boolean, etc.

Examples:

```
showMessage(..)     // shows a message
getAge(..)          // returns the age (gets it somehow)
calcSum(..)         // calculates a sum and returns the result
createForm(..)      // creates a form (and usually returns it)
checkPermission(..) // checks a permission, returns true/false
```

One function – one action

- A function should do exactly what is suggested by its name, no more.
- Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two).
- A few examples of breaking this rule:

  `getAge` – would be bad if it shows an alert with the age (should only get).

  `createForm` – would be bad if it modifies the document, adding a form to it (should only create it and return).

  `checkPermission` – would be bad if it displays the access granted/denied message (should only perform the check and return the result).

Splitting functions

- Functions should be short and do exactly one thing. If that thing is big, maybe it's worth it to split the function into a few smaller functions. Sometimes following this rule may not be that easy, but it's definitely a good thing.
- A separate function is not only easier to test and debug – its very existence is a great comment! For instance, compare the two functions showPrimes(n) below. Each one outputs prime numbers up to n.
- The first variant uses a label:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert( i ); // a prime
  }
}
```

The second variant uses an additional function isPrime(n) to test for primality:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

    alert(i);  // a prime
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

Main Point 2

- Functions should generally not use input and output. Instead they receive parameters (incoming values) and use their return to give the result of their computation to the caller (output).
- Every action has a reaction. The correct reaction to incoming parameters is to return a result – not output it!.