## Function Properties and Methods:

Functions are first-class objects; therefore, they can have properties and methods themselves. For example, functions have a length property that returns the number of parameters they require.

### Call() and Apply() methods:

The *call()* method is used to set the value of ***this*** inside a function to an **object that is provided as the first argument.**

For example:  
 *function sayHello() {*

*return `Hello, my name id ${ this.name }.`;*

*}*

refers to an unspecific object called *this* that has a property called  *name.*

By creating some objects containing the *name* property, then using the *call()* function to invoke the *sayHello()* function, we would provide the object as an argument for *this*:  
 *const* ***clark*** *= { name: {{ Clark }} };  
 const* ***bruce*** *= { name: {{ Bruce }} };*

*sayHello.call(****clark****);  
 << {{ Hello, my name is Clark. }}*

*sayHello.call(****bruce****);  
 << {{ Hello, my name is Bruce. }}*

If the function that is called requires any parameters, these need to be provided as arguments after the first argument (the value of *this*). If *this* is not referred to within the function, thefirst argument of *call()* would then need to be ***null***.

The *apply()* method works in the same way as *call()*, except the arguments of the function are provided as an Array, even if there is only one argument.

*square.apply(null, [4])  
 << 16*

This can be useful if the data for the argument is already an Array, although unnecessary due to ES6 spread operators.

### Custom Properties:

Functions can have custom properties. For example, you can add a *description* property to describe what a function does:

*square.description = {{ Squares a number that is provided as an argument. }}  
 << {{ Squares a number that is provided as an argument. }}*

#### Memoization (or Result Caching):

If a function takes some time to compute a value, save it using the *cache* property. If the same argument is used again later, return the value from the cache, rather than recomputing the value.

## IIFEs (Immediately-Invoked Function Expressions) - Iffies:

IIFEs are anonymous functions that are invoked as soon as defined. This is accomplished by wrapping the function definition within parentheses, then adding another pair directly after, like so:

*(function() {*

*const* ***temp*** *= {{ World }};  
 console.log(`Hello ${temp}!`);*

*})****();***

*<< {{ Hello World! }}*

Doing this will help not to *overcrowd* the global namespace/scope with too many variable names.

## Generators:

ES6 introduced support for generators that maintain the state of a value. To define them, place an asterisk (\*) after the *function* declaration, like so:

*function****\**** *exampleGenerator() {  
 // code for the generator here.  
 }*

This will not run the code in the function, but return a ***Generator*** object to create an iterator that implements a *next()* method that returns a value every time the *next()* method is called.

Generator functions sue a special *yield* keyword to return a value, rather the normal *return* keyword. The difference is *yield* stores the state of the value for the next time *yield* is called. This also pauses the execution of a loop post-*yield*, until the *next()* method is called again.

You can also iterate over the generator using loops, like *for* and *while*.

## Functional Programming:

### Pure functions:

A key aspect of functional programming is the use of {{ pure functions }} that abide by the following rules:

1. The return value of a pure function should only depend on the values provided as arguments. It doesn't rely on values from somewhere else in the program.
2. There are no side-effects. A pure function doesn't change any values or data elsewhere in the program. It only makes non-destructive data transformations and returns new values, rather than altering any of the underlying data.
3. Referential transparency. Given the same arguments, a pure function will always return the same result.

In order to follow these rules, any pure function must have:

* At least one argument; otherwise, the return value must depend on something other than the arguments of the function, breaking the first rule
* A return value; otherwise, there is no point in the function (unless it has changed something else in the program - in which case, it is broken the 'no side-effects' rule).

Pure functions help to make functional programming code more concise and predictable than in other programming styles

## Clients and Servers:

### Request Interface:

Request objects are created using the *Request()* constructor, and include the following properties:

* *Url*: The URL of the requested resource (the only property that is required).
* *Method*: a string that specifies which HTTP method should be used for the request. By default, this is 'GET'.
* *Headers*: This is a Headers object (see later section) that provides details of the request's headers.
* *Mode*: Allows you to specify if CORS is used or not. CORS is enabled by default.
* *Cache*: Allows you to specify how the request will use the browser's cache. For example, you can force it to request a resource and update the cache with the result, or you can force it to only look in the cache for the resource.
* *Credentials*: Lets you specify if cookies should be allowed with the request.
* *Redirect*: Specifies what to do if the response returns a redirect. There is a choice of three values: 'follow' (the redirect is followed), 'error' (an error is thrown) or 'manual' (the user has to click on a link to follow the redirect).

#### Headers:

HTTP headers are used to pass on any additional information about a request or response. Typical information contained in headers includes the file-type of the resource, cookie information, authentication information and when the resource was last modified.

The Fetch API introduced a Headers interface, which can be used to create a Headers object, which can then be added as a property of Request and Response objects.

A new Headers instance is created using a constructor function, as seen in the example below:

*const* ***headers*** *= new Headers();*

and can be given optional arguments with initial header values:

*const* ***headers*** *= new Headers({ {{ Content-type }}: {{ text/plain }}, {{ Accept-Charset }}: {{ utf-8 }}, {{ Accept-Encoding }}: {{ gzip,deflate }} });*

Headers objects also have the following properties and methods to access information about the headers, as well as edit the header information itself:

* *has():* Can be used to check if the headers object contains the header provided as an argument; returns Boolean.
* *get():* Returns the value of the header provided as an argument (like {{ text/plain }}).
* *set():* Can be used to set a value of an already-existing header, or create a new header with the value provided as an argument if it does not already exist:
  + *headers.set({{ Content-Type }}, {{ application/json }});*
* *append():* Adds a new header to the headers object.
* *delete():* Removes the header provided as an argument.
* *keys(), values(),* and *entries():* Iterators that can be used to iterate over the headers key, values, or entries (like key-value pairs in JS or dictionaries in Python).