FUNCTION EXPRESSIONS AND ARROW FUNCTIONS

FUNCTION EXPRESSIONS

This syntax is called a Function Declaration:

```
function sayHi() {
  alert( "Hello" );
}
```

This another syntax for creating a function is called a Function Expression. In JavaScript, a function is a special kind of value.

```
let sayHi = function() {
  alert( "Hello" );
};
```

FUNCTION EXPRESSIONS

Here we can see a variable sayHi getting a value, the new function, created as function() { alert("Hello"); }.

As the function creation happens in the context of the assignment expression (to the right side of =), this is a Function Expression.

There's no name after the function keyword. Omitting the name is allowed for Function Expressions.

FUNCTION EXPRESSIONS

Here we immediately assign it to the variable.

In more advanced situations, a function may be created and immediately called or scheduled for a later execution, not stored anywhere, thus remaining anonymous.

FUNCTION IS A VALUE

In JavaScript, a function is a special value, in sense that we can call it like sayHi().

We can work with it like with other kinds of values so we can copy a function to another variable:

```
function sayHi() {   // (1) create
  alert( "Hello" );
}

let func = sayHi;    // (2) copy
func(); // Hello     // (3) run the copy (it works)!
sayHi(); // Hello    // this still works too
```

FUNCTION IS A VALUE

Here's what happens in this code in detail:

1. The Function Declaration (1) creates the function and puts it into the variable named sayHi.
2. If there were parentheses (2) after sayHi, then func = sayHi() would write the result of call sayHi() into func, not function sayHi itself.
3. Now the function can be called as both sayHi() and func().

FUNCTION IS A VALUE

We could also have used a Function Expression to declare sayHi:

```
let sayHi = function() { // (1) create
  alert( "Hello" );
};

let func = sayHi;
// ...
```

FUNCTION EXPRESSION VS FUNCTION DECLARATION

Function Declaration: a function, declared in main code flow:

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

Function Expression: a function, created inside an expression or inside another syntax construct. Here, this function is created on right side of “assignment expression” (=):

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

FUNCTION EXPRESSION VS FUNCTION DECLARATION

A Function Expression is created when execution reaches it and is usable only from that moment.

Once the execution flow passes to right side of assignment let sum = function(), function is created and can be used (assigned, called, etc.) from now on.

FUNCTION EXPRESSION VS FUNCTION DECLARATION

A Function Declaration can be called earlier than it is defined.

For example, a global Function Declaration is visible in whole script, no matter where it is.

When JavaScript prepares to run a script, it first looks for global Function Declarations in it and creates the functions.

And after all Function Declarations are processed, code is executed. So it has access to these functions.

Function Declaration sayHi is created when JavaScript is preparing to start script and is visible in it.

```
sayHi("John"); // Hello, John

function sayHi(name) {
  alert( `Hello, ${name}` );
}
```

Function Expressions are created when the execution reaches them. That would happen only in the line (*).

```
sayHi("John"); // error!

let sayHi = function(name) {  // (*)
  alert( `Hello, ${name}` );
};
```

FUNCTION EXPRESSION VS FUNCTION DECLARATION

Another special feature of Function Declarations is their block scope.

In strict mode, when a Function Declaration is within a code block, it's visible everywhere inside that block. But not outside of it.

Let's declare a function welcome() depending on the age variable that we get during runtime and then we plan to use it some time later.

FUNCTION EXPRESSION VS FUNCTION DECLARATION

If we use Function Declaration, it won't work as intended.

That's because a Function Declaration is only visible inside the code block in which it resides.

The correct approach would be to use a Function Expression and assign welcome to the variable that is declared outside of if and has the proper visibility.

FUNCTION EXPRESSION VS FUNCTION DECLARATION

```
let age = prompt("What is your age?", 18);
let welcome;

if (age < 18) {
  welcome = function() {
    alert("Hello!");
  };
} else {
  welcome = function() {
    alert("Greetings!");
  };
}

welcome(); // ok now
```

FUNCTION EXPRESSION VS FUNCTION DECLARATION

Or we could simplify it using a question mark operator ?:

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  function() { alert("Hello!"); } :
  function() { alert("Greetings!"); };

welcome(); // ok now
```

FUNCTION EXPRESSION VS FUNCTION DECLARATION

As a rule, when we need to declare a function, first we need to consider Function Declaration syntax.

It gives freedom in how to organize our code, because we can call such functions before they are declared.

That's also better for readability, as it's easier to look up function f(…) {…} in code than let f = function(…) {…};.

But if a Function Declaration does not suit us for some reason, or we need a conditional declaration, then Function Expression should be used.

SUMMARY

- Functions are values and they can be assigned, copied or declared in any place of code.
- If function is declared in main code flow, that's called a "Function Declaration".
- If function is created as a part of an expression, it's called a "Function Expression".
- Function Declarations are processed before the code block is executed and they are visible everywhere in the block.
- Function Expressions are created when execution flow reaches them.

SUMMARY

In most cases when we need to declare a function, a Function Declaration is preferable.

It's visible prior to the declaration itself. That gives us more flexibility in code organization, and is usually more readable.

We should use a Function Expression only when a Function Declaration is not fit for the task.

ARROW FUNCTIONS

There’s another simple and concise syntax for creating functions

That’s often better than Function Expressions and it’s called “arrow functions”:

```
let func = (arg1, arg2, ..., argN) => expression;
```

ARROW FUNCTIONS

This creates a function func that accepts arguments arg1..argN.

Then evaluates the expression on right side with their use and returns its result.

In other words, it is the shorter version of:

```
let func = function(arg1, arg2, ..., argN) {
  return expression;
};
```

ARROW FUNCTIONS

Let's see a concrete example: (a, b) => a + b means a function that accepts arguments named a and b. Upon execution, it evaluates expression a + b and returns the result.

```
let sum = (a, b) => a + b;

/*
let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

ARROW FUNCTIONS

If we have only one argument, then parentheses around parameters can be omitted, making that even shorter.

```
let double = n => n * 2;
// let double = function(n) { return n * 2 }
alert( double(3) ); // 6
```

If there are no arguments, parentheses are empty, but they must be present:

```
let sayHi = () => alert("Hello!");
sayHi();
```

ARROW FUNCTIONS

Arrow functions can be used in the same way as Function Expressions. For instance, to dynamically create a function:

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  () => alert('Hello!') :
  () => alert("Greetings!");

welcome();
```

They are also convenient for simple one-line actions.

ARROW FUNCTIONS

Sometimes we need more complex function, with multiple expressions and statements.

We can enclose them in curly braces but it require a return within them to return a value (like a regular function does).

```
let sum = (a, b) => {  // curly brace opens a multiline function
  let result = a + b;
  return result; // we need an explicit "return"
};

alert( sum(1, 2) ); // 3
```

SUMMARY

Arrow functions are handy for simple actions, especially for one-liners:

1. Without curly braces `(...args) => expression`, right side is an expression so function evaluates it and returns the result. Parentheses can be omitted, if there's only a single argument.
2. With curly braces `(...args) => { body }`, brackets allow us to write multiple statements inside the function, but we need an explicit `return` to return something.