# Conditional Execution

Not all programs are straight roads. Conditional excutions are like what if in real life. We use if keyword in JavaScript.

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
console.log("Your number is the square root of " +
theNumber * theNumber);
}
```

what if we didn't type an number we can execute the code with conditional excution

```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
console.log("Your number is the square root of " +
theNumber * theNumber);
} else {
console.log("Hey. Why didn't you give me a number?");
}
```

if we have severl conditions we can write like this -

```
let num = Number(prompt("Pick a number"));
if (num < 10) {
console.log("Small");
} else if (num < 100) {
console.log("Medium");
} else {
console.log("Large");
}
```

# While and Do Loops

Imagine we wanna output all numbers starting from 0. One way to write as follows.

```
console.log(0);
console.log(1);
console.log(2);
console.log(3);
console.log(4);
console.log(5);
```

```
console.log(6);
console.log(7);
console.log(8);
console.log(9);
console.log(10);
```

That works but the main idea of writing a program is not to repeast something again and again. DRY(don't repeat yourselves) is the important idealogy that programmers need. That's why we have to change the above code like this -

```
let number = 0;
while (number < 10) {
console.log(number);
number = number + 1;
}
```

The code means while the number is still less than 10 the below code will work and at last the number will increase by one.

It is really effecive right now we can write 2 power ten.

```
let result = 1;
let counter = 0;
while (counter < 10) {
    result = result * 2;
    counter = counter + 1;
}
console.log(result);
```

## Do while loop

```
let yourName = "";
do {
    yourName = prompt("Who are you?");
} while (!yourName);
console.log(yourName);
```

The above code will ask your name until it gets something that is not an empty string.

## Switch break

```
switch (prompt("What is the weather like?")) {
case "rainy":
console.log("Remember to bring an umbrella.");
```

```
break;
case "sunny":
console.log("Dress lightly.");
case "cloudy":
console.log("Go outside.");
break;
default:
console.log("Unknown weather type!");
break;
}
```

## The quiz

```
#
##
###
####
#####
######
#######
```

Hint you can know your string lenght by using .length function. As an example -

```
let dog = "aung net"
dog.length;
```

There are 3-4 solutions to this quiz.

```
for (let hash = "#"; hash.length < 7; hash += "#") {
    console.log(hash)
}
```

# Functions

Functions are the bread and butter of JavaScript Programming also the main pillar of most of the programming languages.

### Defining function

```
function greeting() {
    console.log("hello world")
}
```

The above code is function declaration, we had declare the greeting function and we can use it by writing /// greeting() // . In JS, () means it is a function whatever we use the already declared function, we need to use ().

## Function Expressions

Variable can be said an expression. Whatever we assigned something to something, it means expression. In real life, we express our love, hate and greed by expressing with our behavior. Functions are always like that.

```
let hello = function greeting() {
    console.log("hello world")
}
```

you can write the above function in short like below -

```
let hello = function() {
    console.log("hello world")
}
```

These are the difference between function declaration and expression. Let's look at the code below to understand about functions -

```
function add(a,b) {
    return a + b
}

function circle(r) {
    const PI = 3.142
    return PI * r * r
}

function greet() {
    console.log("Hello World")
}

function sayHello(name) {
    console.log(`Hello ${name}`)
}
```

When we want to output the value from function instead of using console.log, we use return keywords. While declaring function inside the () are called parameters and when we call the function like add(1,2), (1,2) are called arguments.

In function expressions -

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

For some knowledge -

```
function circle(r) {
    const PI = 3.142
    return PI * r * r
}

function diameter(d) {
    return PI * d
}
```

If you use diameter function, there will be no output because PI is assigned in the circle function and it can't be used outside of its scope function.

OK! let's write some function combining with conditions.

```
const power = function(base, exponent) {
    let result = 1;
    for (let count = 0; count < exponent; count++) {
        result *= base;
    }
    return result;
};
power(2,10);
```

### Optional Arguments

We can write something like this -

```
const power = function(base, exponent = 10) {
    let result = 1;
    for (let count = 0; count < exponent; count++) {
        result *= base;
    }
    return result;
};
power(2);
```

This version of power makes its second argument optional .

## Nested Scope

There is an ingredient function inside cooking function. let's see -

```
const cooking = function(factor) {
const ingredient = function(amount, unit, name) {
let ingredientAmount = amount * factor;
if (ingredientAmount > 1) {
unit += "s";
}
console.log(`${ingredientAmount} ${unit} ${name}`);
};
ingredient(1, "can", "chickpeas");
ingredient(0.25, "cup", "tahini");
ingredient(0.25, "cup", "lemon juice");
ingredient(1, "clove", "garlic");
ingredient(2, "tablespoon", "olive oil");
ingredient(0.5, "teaspoon", "cumin");
};
```

The code inside ingredient function can see the cooking function. But its local bindings, such as ingredientAmount, unit and name can't be seen from cooking function. This approach to binding visibility is called lexical scoping.

## Declaration Notation

```
console.log("The future says:", future());

function future() {
    return "You'll never have girlfriend";
}
```

## Arrow Functions

Let's use the code from above example -

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

By arrow funtion you can write it below -

```
let add = (a,b) => a + b
```

=> remove all of the function{} and return keywords. How simple and clear. If functions have just one statement like above, we can remove return.

# Callback Function

Callback function is the most fundamental and important function in javasript.

```
function twice(number, add) {
    let result = add(number)
    return result * 2
}
```

Firstly, we have defined the twice function and there is a add function inside twice function. That's why we can call twice function as below -

```
twice (5, function(x) => {
    return x + 1
})
```

In your eye, it might be complicated but among the two parameter in twice, add is a function after we have assigned to result. The engine know add is not number or string, that is the reason we can called function(x) when we use twice function. Please look this code again and again you understand.

In arrow function we can write -

```
let twice = (n,f) => f(n) * 2

twice(2, a => a + 2)
```

It is really compact and simple, right?