4-9. Flow Control Statements

When you write code it very often comes to a state when you want a different outcome depending on different conditions. 4-9-1

That is when **conditional statements** comes in. 4-9-2

They are used to perform different actions based on different conditions.

If statement 4-9-3

We can check **if a statement is true** and in that case perform a specific action.

Imagine we have a variable with a value. Depending on that value we might want write a message to the console. 4-9-4

```
var age = 18;
var text = "";

if(age === 18) {
    console.log("The statement is true");
    text = "The age is equal to eighteen."
}
```

In real life this variable might not be hard coded as in the example.

We have a `variable` called `age`. We check if the value of `age` is equal to **18**, which it is. Because the **statement is true** we will write the message *The statement is true* to the console.

In this example, if the statement would equal false nothing would happen.

With conditional statements the statement wrapped within the `if()` is checked first. 4-9-5

If the statement is true it will enter the following block of code.

If else statement 4-9-6

In the previous example nothing would happen if the statement would turn out to be false.

To have one action for each case, **true and false**, we can use an *if else statement*. 4-9-7

If the statement is true, do this. 4-9-8

Else, if the statement is false, do this.

Think of an example where we want different outcomes depending on the value of a variable: 4-9-9

```
var age = 19;

if(age >= 18) {
    console.log("Adult")
}
else {
    console.log("Minor")
}
```

First we check the statement wrapped within `if()`. 4-9-10

If the value of the variable `age` is **equal to or greater than 18** we should write *Adult* to the console.

Else, meaning the statement is false (`age` is less than **18**), we should write *Minor* to the console. 4-9-11

If the statement within the `if()` is false we move to `else`.

No curly brackets required 4-9-12

If we only have one line in our block of code to be executed we can skip the curly brackets.

```
var age = 18;

if(age >= 18)
    console.log("Adult")
else
    console.log("Minor")
```

We can even simplify this one step further using one line conditional statements, also know as **ternary operator** 4-9-13

```
var age = 18;

age >= 18 ? console.log("Adult") : console.log("Minor");
```

When using **one line conditional statements** we skip the keywords and curly brackets. 4-9-14

Instead we replace these by a **question mark and a colon**.

4-9-15

```
var age = 18;

if(age >= 18) {
    console.log("Adult")
}
else {
    console.log("Minor")
}
```

... same as:

```
var age = 18;

age >= 18 ? console.log("Adult") : console.log("Minor");
```

The part of before the question mark is the actual statement. 4-9-16

The statement that would be wrapped within the parentheses of `if()`

The part after the question mark but before the colon is the block of code to be executed if the statement is true. 4-9-17

The code that would be wrapped within the curly brackets of `if(...) {...}`

The part after the colon is the block of code to be executed if the statement is false. 4-9-18

The code that would be wrapped within the curly brackets of the `else {...}`

But as soon as we have more than one action within our code to be executed this would not work. 4-9-19

```
var age = 18;
var text = "";

age >= 18 ? console.log("Adult"); text = "Adult" : console.log("Minor");
```

Then we have to wrap it within a block of curly brackets. 4-9-20

```
var age = 18;
var text = "";

if(age >= 18) {
    console.log("Adult");
    text = "Adult";
} else {
    console.log("Minor");
}
```

Several if statements 4-9-21

If we want more than one statement we can chain our if statements with `else if()`.

By doing this we can define several statements to check before we know what action that will happen. 4-9-22

```
var age = 23;

if(age >= 20) {
    console.log("Ready for SystemBolaget")
} else if(age >= 18 && age < 20) {
    console.log("The pub, here we come!")
} else {
    console.log("Got to wait a few more years... Bummer!")
}
```

Switch statement 4-9-23

Another way to chain statements would be to use a **switch statement**.

```
switch(fruits) {
    case "Banana":
        console.log("Banana is good!");
        break;
    case "Orange":
        console.log("I am not a fan of orange.");
        break;
    case "Apple":
        console.log("How you like them apples?");
        break;
    default:
        console.log("I have never heard of that fruit...");
}
```

We have different cases depending on the value we are operating on. 4-9-24

```
case "Banana":
    console.log("Banana is good!");
    break;
```

In every case we need to use the `break` keyword to say that we have found the right one. We do not want to continue.

If the value does not match any case the `default` part will apply. 4-9-25

```
default:
    console.log("I have never heard of that fruit...");
```

In switch statements it is mandatory to define default. 4-9-26

Conditional loops 4-9-27

In JavaScript we have different kinds of loops, **conditional loops** for example.

We can choose to do something once or several times depending on a condition. 4-9-28

Loops like these are **while** and **do...while** loops.

while loop 4-9-29

A **while loop** will preform a task *while* some given condition is true.

```
while (!finished) {
    // keep working
}
```

It will check the condition, if the condition is true it will enter the following block of code. 4-9-30

When the execution of the block of code is done, it will check the condition again. 4-9-31

This continues until the condition is false.

do...while loop 4-9-32

A **do...while loop** will perform a given task at least once.

```
do {
    // do work
} while (!finished);
```

Unlike the **while**, the **do...while** will perform a given task and after that check a condition. 4-9-33

If the condition returns true it will do the task one more time.

This continues until the condition returns false. 4-9-34

Q So what is the actual difference between a **while** and **do...while** loop? *4-9-35*

A The **do...while** performs the given task at least once, then checks the condition. *4-9-36*

The **while** loop will only do the task if the condition is true.

(As usual, you can `continue` and `break` in loops.) *4-9-37*

Exceptions *4-9-38*

We can try to perform a specific task, and catch the exception if the task fails.

Q What is an exception? *4-9-39*

A An exception is an error, same thing but different name basically. *4-9-40*

Errors will occur when executing code, such can be failures made by the creator of the code, wrong input by user, etc. *4-9-41*

When an error occurs, JavaScript will **throw an exception**, *an error*. *4-9-42*

We want to handle this so that our code can continue and not end up in a crash.

`try`/`catch` *4-9-43*

```
try {
    // something that might fail
}
catch (e) {
    // handle a possible thrown exception
}
```

To catch an exception it has to be thrown. *4-9-44*

Example of exceptions we have encountered is `ReferenceError` and `TypeError`.

When we've seen these it have been something like *4-9-45*

```
Uncaught TypeError: Assignment to constant variable.
```

Notice the **uncaught** word.

We can catch such errors within our `catch(e)`, the `e` parameter holds information about the error. *4-9-46*

```
try {
    console.log(x); // This line will throw an exception
}
catch (e) {
    // We catch the exception
    console.log(e); // ReferenceError: x is not defined
}
```

We can throw exceptions ourselves, with the `throw` statement. *4-9-47*

```
throw ReferenceError("The reference wasn't found");
// Uncaught ReferenceError: The reference wasn't found
```

With help from the exception coming into the catch we can check what type of error we got. *4-9-48*

```
try {
    console.log(x); // This line will throw an exception
}
catch (e) {
    if(e instanceof ReferenceError)
        console.log("ReferenceError") // ReferenceError
}
```

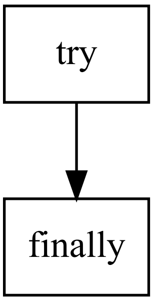
`try/catch/finally` 4-9-49

We can apply one more step to our `try/catch`; **finally**

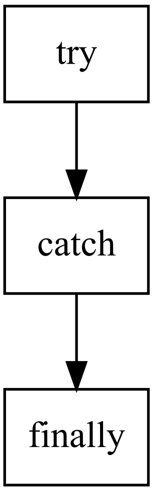
```
try {
    console.log(x); // Try performing the task
}
catch (e) {
    console.log(e); // Error occured
}
finally {
    console.log("This will always be done");
}
```

The `finally` code block will always be executed, no matter if an error occurs or not. 4-9-50

If the task succeeds we will enter `finally` after `try` is done. 4-9-51



If an error occurs, `finally` will be entered after the exception has been handled in the catch. 4-9-52



Iteration through values

4-9-53

If we have a collection of values it usually comes to a point where we want to iterate through these values.

for loop

4-9-54

For iteration through collections of values we have `for loops`.

```
let myCollection = [1, 3, 5, 8];

for (let element of myCollection) {
    // handle each element
}
```

In JavaScript, there are different forms of `for` loops. 4-9-55

One is a counter-like loop, that starts on a specific number and then continues depending on some given condition. 4-9-56

```
for (var i = 0; i < 5; i++) {
    console.log(i)
    // 0
    // 1
    // 2
    // 3
    // 4
}
```

This loop will start on the value of 0 (`i = 0`).

It will continue as long as `i` is less than `10`.

After the block of code has been executed, `i` will be increased by **1**.

The output would be:

The syntax: 4-9-57

```
for (statement 1; statement 2; statement 3) {
    block of code to be executed
}
```

Statement 1) Executed before the loop starts 4-9-58

Statement 2) Condition which determines if the loop should continue

Statement 3) Executed each time after the block of code has been executed

We have for loops used for iteration through arrays of values. 4-9-59

The following gives us the current **value** of the current item in the collection. 4-9-60

```
var collection = [33,44,55];
for (var i of collection) {
    console.log(i);
    // 33
    // 44
    // 55
}
```

Note the `of` keyword

The following gives us the **index** of where in the collection the current item is located. 4-9-61

```
var collection = [33,44,55];
for (var i in collection) {
    console.log(i);
    // 0
    // 1
    // 2
}
```

Remember indices of an array in JavaScript start on 0.

If we want to access the actual value of the item in the collection but we are still using the `in for` loop, we use index of the item. *4-9-62*

```
var collection = [33,44,55];
for (var i in collection) {
    console.log(i);
    console.log(collection[i]);
    // 0
    // 33
    // 1
    // 44
    // 2
    // 55
}
```

What to remember with these is that: *4-9-63*

- **in** gives the index of the item in the collection
- **of** gives the value of the item in the collection

forEach *4-9-64*

We have a method called `.forEach` on our array object.

This works almost the same way as with the for loops. *4-9-65*

```
var collection = [33,44,55];
collection.forEach(e => {
    console.log(e);
    // 33
    // 44
    // 55
});
```

The first parameter is the value.

With the `forEach` we can access both the value and the index at the same time. 4-9-66

```
var collection = [33,44,55];
collection.forEach((e, i) => {
    console.log(e, i);
    // 33, 0
    // 44, 1
    // 55, 2
});
```

By including the second parameter to our callback function we get the index.

break and continue 4-9-67

When working with loops we can decide when the loop should stop och continue.

Break terminates the execution of the loop entirely. 4-9-68

```
var i = 0;

while (i < 10) {
  if (i == 5) {
    break;
  }
  i += 1;
  console.log(i);
  // 1
  // 2
  // 3
  // 4
  // 5
}
```

When the loop hits the `break` it will not continue. 4-9-69

continue 4-9-70

The `continue` statement behaves differently depending in what loop it is present.

It does not terminate the loop but rather jumps to next step. 4-9-71

In `while` loops, it jumps back to the condition 4-9-72

In a `for` loop it jumps to the update expression (`i++`);

4-9-73

```
var i = 0;

while (i < 10) {
  i++;

  if (i > 2 && i < 8 ) {
    continue;
  }

  console.log(i);
  // 1
  // 2
  // 8
  // 9
  // 10
}
```