SUNSTONE

JavaScript

LECTURE 5 NOTES

Statements

- **What are Control Statements?**

  - In JavaScript, control statements are used to control the flow of execution in a program. They allow you to make decisions or repeat code based on certain conditions.
  - Control Statements also known as Control Structures or Flow Control statements, these are the statements that decide the execution flow of the program. Typically the program execution begins from the first line and then proceeds in a sequential manner to the last line of that JS code. However, in between this, the flow of the execution can be branched (based on some conditions) or re-iterated (loops) based on some criteria. This is the functionality that is provided by Control Statements in JavaScript.
  - There are three categories of control statements:
    1) Selection statements
    2) Iterative statements
    3) Jump statements

1. Selection Statements

- In JavaScript, selection statements are used to make decisions and execute different blocks of code based on conditions. They are also known as conditional control statements.
- Whenever a condition is to be tested depending on which particular tasks are to be performed, Conditional control statements are used.
- It is subdivided into two categories:
  A) If statements
  B) Switch statements

A. If Statements

The JavaScript if-else statement is used to execute the code whether a condition is true or false. There are four forms of if statements in JavaScript.

a) If Statement
b) If else statement
c) if else if statement
d) Nested if statement

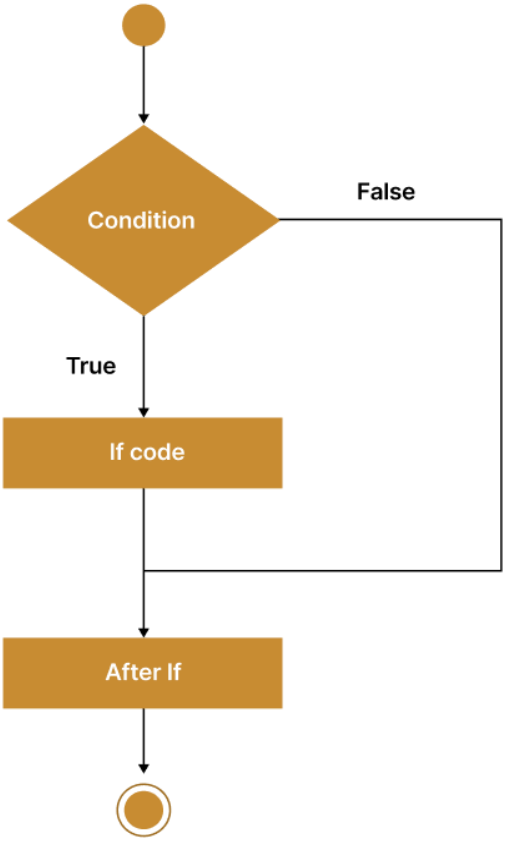
a. If statement

It evaluates the content only if the expression is true.

SYNTAX:

```
if(expression){
//content to be evaluated
}
```

Flowchart of If statement:



Let's see the simple example of if statement in javascript.

```
var a=20;
if(a>10){
document.write("value of a is greater than 10");
}
```

Output: `value of a is greater than 10`

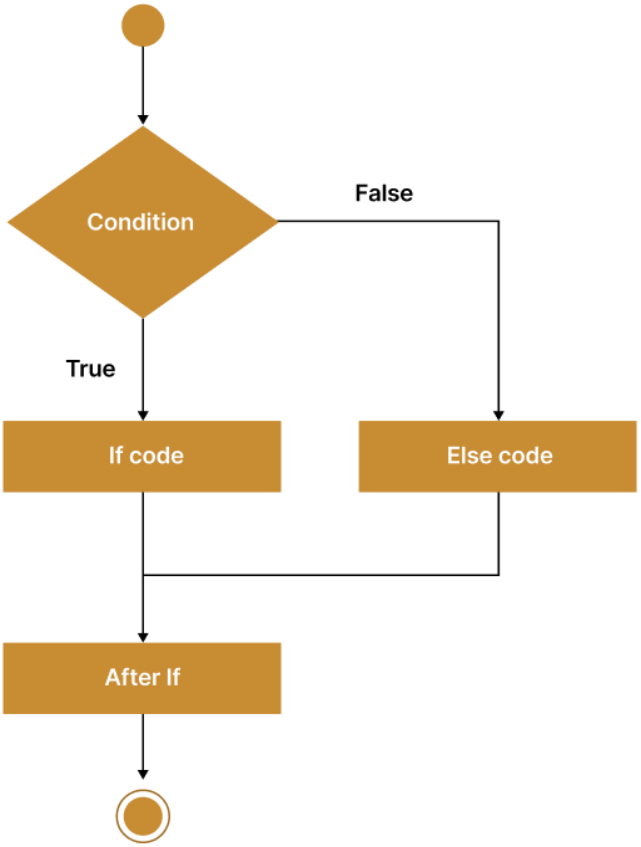
b. If...else Statement

It evaluates the content whose condition is true or false.

SYNTAX:

```
if(expression){
//content to be evaluated if the condition is true
}
else{
//content to be evaluated if the condition is false
}
```

Flowchart of If...else statement



Let's see the example of an if-else statement in JavaScript to find out the even or odd number.

```
var a=20;
if(a%2==0){
document.write("a is even number");
}
else{
document.write("a is odd number");
}
```

Output: `a is even number`

c. If...else if Statement

It evaluates the content only if the expression is true from several expressions.

SYNTAX:

```
if(expression1){
//content to be evaluated if expression1 is true
}
else if(expression2){
//content to be evaluated if expression2 is true
}
else if(expression3){
//content to be evaluated if expression3 is true
}
else{
//content to be evaluated if no expression is true
}
```

Let's see the simple example of if else if statement in JavaScript.

```
var a=20;
if(a==10){
document.write("a is equal to 10");
}
else if(a==15){
document.write("a is equal to 15");
}
else if(a==20){
```

```
document.write("a is equal to 20");
}
else{
document.write("a is not equal to 10, 15 or 20");
}
```

Output: `a is equal to 20`

d. Nested if statement

A nested if statement in JavaScript refers to an if statement that is placed within another if statement. This allows for multiple levels of conditions to be evaluated and different blocks of code to be executed based on the outcome of those conditions.

SYNTAX:

```
if condition1 {
   // code to be executed if condition1 is true
  if condition2 {
       // code to be executed if both condition1 and condition2
are true
  }
}
```

Let's see the simple example of a nested if statement in JavaScript.

```
let x = 10;
let y = 5;
if (x > 0) {
  console.log("x is positive");
  if (y > 0) {
    console.log("y is also positive");
  } else {
    console.log("y is not positive");
  }
} else {
  console.log("x is not positive");
}
```

Output:

```
x is positive
y is also positive
```

B. Switch Statements

The switch statement allows you to perform different actions based on different values. It evaluates an expression and compares it with multiple cases to execute the corresponding block of code. It can also include a default case to specify code to execute when none of the cases match.

SYNTAX:

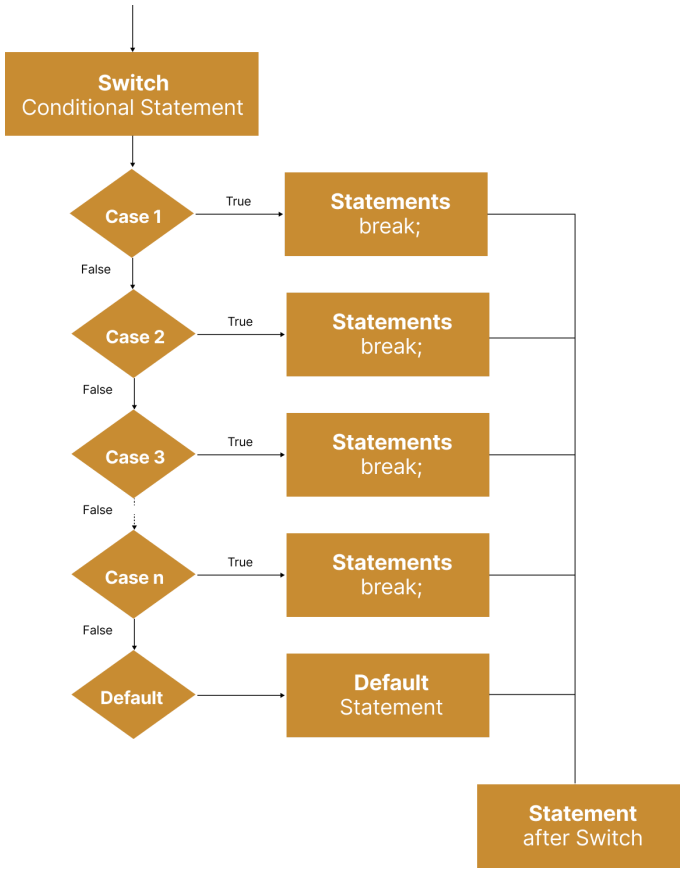
```
switch (expression) {
  case value1:
    // code to execute if the expression matches value1
    break;
  case value2:
    // code to execute if the expression matches value2
    break;
  default:
     // code to execute if the expression matches none of the
cases
}
```

Let's see a simple example of a switch statement.

```
var grade='B';
var result;
switch(grade){
case 'A':
result="A Grade";
break;
case 'B':
result="B Grade";
break;
case 'C':
result="C Grade";
break;
default:
result="No Grade";
}
document.write(result);
```

Output: `B Grade`

Flow Chart of Switch Statement



2. Iterative Control Statements

- Whenever a particular task has to be iterated for a particular number of times or depending on some condition, Looping control statements are used.
- They are also known as looping control statements. Iterative control statements in JavaScript allow you to repeat a block of code multiple times. They are used to create loops and perform tasks such as iterating over arrays, executing code until a certain condition is met, or executing code a specific number of times.
- They are sub-divided into four categories:

    a. For loop
    b. While loop
    c. Do While loop

d. For in loop

a. For loop

The JavaScript for loop iterates the elements for a fixed number of times. It should be used if the number of iterations is known.

SYNTAX:

```
for (initialization; condition; increment)
{
    code to be executed
}
```

Let's see a simple example of for loop in javascript.

```
<script>
for (i=1; i<=5; i++)
{
document.write(i + "<br/>")
}
</script>
```

Output:

1
2
3
4
5

b. While loop

The JavaScript while loop iterates the elements for an infinite number of times. It should be used if the number of iterations is not known.

SYNTAX:

```
while (condition)
{
    code to be executed
}
```

Let's see a simple example of a while loop in JavaScript.

```
<script>
var i=11;
while (i<=15)
{
document.write(i + "<br/>");
i++;
}
</script>
```

Output:

```
11
12
13
14
15
```

c. Do while loop

The do while loop iterates the elements for an infinite number of times like the while loop. But, code is executed at least once whether the condition is true or false.

SYNTAX:

```
do{
    code to be executed
}while (condition);
```

Let's see a simple example of do-while loop in javascript.

```
<script>
var i=21;
do{
document.write(i + "<br/>");
i++;
}while (i<=25);
</script>
```

Output:

```
21
22
23
24
25
```

d. For-in loop

The for...in loop in JavaScript is used to iterate over the enumerable properties of an object. It allows you to loop through the keys (property names) of an object and perform actions on each property.

SYNTAX:

```
for (variable in object) {
  // code to execute for each property
}
```

The variable represents a new variable name that will be assigned the property key on each iteration. The object is the object you want to iterate over.

```
const person = {
  name: 'John',
  age: 30,
  gender: 'male'
};

for (let key in person) {
  console.log(key + ': ' + person[key]);
}
```

In the above code, the for...in loop iterates over the properties of the person object. On each iteration, the key variable is assigned the current property name, and person[key] accesses the corresponding property value. The loop body logs the key-value pairs to the console.

Output:

```
name: John
age: 30
```

```
gender: male
```

3. Jump Statement

- Jump statements in JavaScript allow you to control the flow of execution by 'jumping' to a different part of the code. They provide a way to break out of loops prematurely, skip iterations, or terminate the execution of a block of code.
- There are three main jump statements in JavaScript:

    a. break statement
    b. continue statement
    c. return statement

a. break statement

The break statement is used to exit or terminate a loop or switch statement. It allows you to immediately stop the execution of the loop or switch and continue with the next statement after the loop or switch.

Example of using the break in a loop:

```
for (let i = 0; i < 5; i++) {
  if (i === 3) {
    break; // exit the loop when i is 3
  }
  console.log(i);
}
```

Output:

```
0
1
2
```

Example of using the break in a switch statement:

```
let value = 2;
switch (value) {
  case 1:
    console.log("Case 1");
```

```
    break;
  case 2:
    console.log("Case 2");
    break; // exit the switch statement when value is 2
  case 3:
    console.log("Case 3");
    break;
  default:
    console.log("Default case");
}
```

Output:

```
Case 2
```

b. continue statement

The continue statement is used to skip the current iteration of a loop and move to the next iteration. It is commonly used when you want to skip specific iterations based on certain conditions.

Example:

```
for (let i = 0; i < 5; i++) {
  if (i === 2) {
    continue; // skip the iteration when i is 2
  }
  console.log(i);
}
```

Output:

```
0
1
3
4
```

c. return statement

The return statement is used to exit a function and optionally return a value. It terminates the function's execution and immediately returns the specified value (if

any) to the calling code.
Example:

```
function multiply(a, b) {
  if (b === 0) {
    return 0; // return 0 if b is 0
  }
  return a * b; // return the multiplication result
}
console.log(multiply(5, 3)); // Output: 15
console.log(multiply(4, 0)); // Output: 0
```

These jump statements provide control over loops, switch statements, and functions, allowing you to break out of loops, skip iterations, or exit functions when certain conditions are met.

- **Exception Handling with 'try/catch/finally'**
  - Exception handling with try/catch/finally is a common programming construct used to handle and manage exceptions in many programming languages. It allows you to catch and handle exceptions that occur during the execution of a block of code and provides a mechanism to perform cleanup actions regardless of whether an exception was thrown or not.
  - Let's break down the different parts of this construct:
    - **Try block:** The code that may potentially throw an exception is placed within a try block. It is enclosed by the try keyword followed by a set of curly braces. If an exception occurs within the try block, it is immediately caught and handled by the corresponding catch block.
    - **Catch block:** The catch block is where you handle the exceptions that occur within the try block. It is preceded by the catch keyword followed by an exception type and a parameter name within parentheses, and then a set of curly braces. The catch block is executed only if an exception of the specified type (or any of its derived types) is thrown within the try block. You can have multiple catch blocks to handle different types of exceptions.

- **Finally block:** The finally block is optional and follows the catch block(s) if present. It is executed regardless of whether an exception occurred or not. This block is useful for performing cleanup actions, such as closing resources or releasing memory, that should be executed regardless of the outcome.

  Here's an example in Java to illustrate the usage of try/catch/finally:

```
try {
    // Code that may throw an exception
} catch (ExceptionType1 e) {
    // ExceptionType1-specific exception handling
} catch (ExceptionType2 e) {
    // ExceptionType2-specific exception handling
} finally {
    // Cleanup actions
}
```

  In the example, if an exception of ExceptionType1 occurs, the first catch block will handle it. If an exception of ExceptionType2 occurs, the second catch block will handle it. If no exception is thrown, the finally block will still be executed to perform any necessary cleanup actions.

  For example,

```
try{
var a=2;
if(a==2)
document.write("ok");
}
catch(Error){
document.write("Error found"+e.message);
}
finally{
document.write("Value of a is 2 ");
}
```

  Output:

```
okValue of a is 2
```

- **The ‘with’ Statement and Strict Mode**

The ‘with’ statement and strict mode are two distinct features in JavaScript, and they have different purposes and implications. Let's discuss each of them separately:

1. The ‘with’ Statement

- The with statement in JavaScript allows you to create a temporary scope where the properties of an object can be accessed directly without explicit qualification. It provides a shorthand way to write code by reducing the need for repetitive object references. However, its usage is generally discouraged due to various issues and potential pitfalls it can introduce. It is not recommended to use the with statement in modern JavaScript code.
- Here's an example of how the with statement can be used:

```
const person = {
  name: 'John',
  age: 30
};
with (person) {
  console.log(name); // Accesses person.name directly
  console.log(age); // Accesses person.age directly
}
```

- In the example, the with statement creates a temporary scope where the properties name and age of the person object can be accessed directly, as if they were local variables. However, using the with statement can make the code more error-prone and harder to understand. It can lead to unintended variable shadowing and affect performance negatively. Therefore, it is generally recommended to avoid using the with statement in JavaScript.

2. Strict Mode

- Strict mode is a feature introduced in ECMAScript 5 (ES5) that enables a stricter set of rules for JavaScript code. When strict mode is enabled, the JavaScript runtime enforces stricter parsing and error handling, which helps in writing more reliable and maintainable code. It is enabled by adding the following directive at the beginning of a script or a function:

```
'use strict';
```

- When strict mode is active, it introduces the following changes:
    - Variables must be declared with var, let, or const before use. Assigning a value to an undeclared variable results in an error.
    - Deleting variables, functions, or function arguments is not allowed.
    - Assigning values to read-only properties or non-writable global variables is not allowed.
    - Octal literals (e.g., 0123) and certain octal escape sequences are not allowed.
    - The `this` value is undefined in functions that are not methods or constructors.
    - Duplicate parameter names in function declarations produce an error.
    - And more.

- By enabling strict mode, you can catch common programming mistakes, avoid potentially confusing behavior, and ensure better code quality.
- Here's an example that demonstrates the usage of strict mode:

```
'use strict';

// Some code here
```

- In the example, the `'use strict';` directive at the beginning of the script enables strict mode for the entire script. It's worth noting that strict mode is opt-in, meaning that it needs to be explicitly enabled in each script or function where you want it to apply.
- It's generally recommended to use strict mode in all JavaScript code as it promotes better coding practices and helps catch errors early.