JavaScript Switch Statements

A *switch* statement allows a program to evaluate an expression by attempting to match the expression's value to a *case label*. If a match is found, the program jumps to the statement(s) associated with the matched label and continues executing at that point. Note that execution will continue sequentially through all the statements starting at the jump point unless there is a call to `break;`, which exits the switch statement. A switch statement looks like this:

```
switch (expression) {
    case label1:
        statement1;
        break;
    case label2:
        statement2;
        break;
    case label3:
        statement3;
        statement4;
        break;
    default:
        statement;
}
```

JavaScript Strings:

These are chains of zero or more Unicode characters (i.e., letters, digits, and punctuation marks) used to represent text.

We denote string literals by enclosing them in single (`'`) or double (`"`) quotation marks. Double quotation marks can be contained in strings surrounded by single quotation marks (e.g., `'"'` evaluates to `"`), and single quotation marks can be contained in strings surrounded by double quotation marks (e.g., `"'"` evaluates to `'`). In addition, you can also enclose a single or double quotation within another quotation of its same type by preceding the quotation you wish to have interpreted literally with the escape character (\).

Each string has a property called `String.length` denoting the length of, or number of characters in, the string.

String Constructor

To create a new string, we use the syntax `String(value)` where value denotes the data we want to turn into a string.

Methods :

- `String.charAt()`
- `String.concat()`
- `String.includes()`
- `String.endsWith()`
- `String.indexOf()`
- `String.lastIndexOf()`
- `String.match()`
- `String.normalize()`
- `String.repeat()`
- `String.replace()`
- `String.search()`
- `String.slice()`
- `String.split()`
- `String.startsWith()`
- `String.substr()`
- `String.substring()`
- `String.toLowerCase()`
- `String.toUpperCase()`
- `String.trim()`
- `String.trimLeft()`
- `String.trimRight()`

JavaScript Loops

Loops are a quick and easy way to repeatedly perform a series of instructions, and they are typically run a finite number of times. JavaScript has the following types of loops:

- *for*
- *while*
- *do-while*
- *for-in*
- *for-of*

(i) for

The *for* statement creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by one or more statements that will be executed in the loop.

Basic Syntax

```
for (initialization; condition; finalExpression) {
    statement(s);
}
```

(ii) while

The *while* statement creates a loop that executes its internal statement(s) as long as the specified condition

evaluates to *true*. The condition is evaluated before executing the statement.

Basic Syntax

```
while (condition) {
    statement(s);
}
```

(iii) do-while

The *do-while* statement creates a loop that executes its internal statement(s) until the specified condition

evaluates to false. The condition is evaluated after executing the internal statement(s), so the contents of the loop always execute *at least* once.

Basic Syntax

```
do {
    statement(s);
} while (condition);
```

(iv) for-in

This loop iterates (in an arbitrary order) over the *name* of each enumerable property in an object, allowing statements to be executed for each distinct property.

Basic Syntax

```
for (var variable in object) {
    // insert code that uses variable here
}
```

(v) for-of

This loop iterates over iterable objects such as an *Array*, *Map*, *Set*, *String*, *TypedArray*, *arguments object*, etc. It essentially iterates over the *value* of each distinct property in the structure, such as each letter in a word or each element in an array.

Basic Syntax

```
for (let variable of iterable) {
    statement(s);
}
```

Arrays in JS:

(i) Create an array:

```
var a = ['first', 'second'];

console.log('a\'s contents:', a);
console.log('a\'s length:', a.length);
```

(ii) Access (Index into) an Array item:

```
let a = ['first', 'second'];

// first = 'first'
let first = a[0];

// last = 'second'
let last = a[a.length - 1];

console.log('a[0]:', first);
console.log('a[a.length - 1]:', last);
```

(iii) Loop over an array:

```
var a = ['first', 'second'];

a.forEach(function(e, i, array) {
   // 'i' is the index & 'e' is the element
   console.log(i + ' ' + e);
});
```

(iv) Append at the end of array:

```
var a = ['first', 'second'];

// Append 'third' to array 'a'
a.push('third');

console.log('a:', a);
```

(v) Remove from end of array:

```
var a = ['first', 'second', 'third'];
console.log('Original Array:', a);
// Remove the last element from the array
let removed = a.pop();
console.log('Modified Array:', a);
console.log('Removed Element:', removed);
```

(vi) Remove from front of array:

```
var a = ['first', 'second', 'third'];
console.log('Original Array:', a);

// Remove the first element from the array
let removed = a.shift();

console.log('Modified Array:', a);
console.log('Removed Element:', removed);
```

(vii) Add to front of array:

```
var a = ['first', 'second', 'third'];
console.log('Original Array:', a);

// Insert element at the beginning of the array
a.unshift('fourth');

console.log('Modified Array:', a);
```

(viii) Find index of an item of an Array:

```
var a = ['first', 'second', 'third', 'fourth'];

let position = a.indexOf('second');

console.log('a:', a);
console.log('position:', position);
```

(ix) Remove an item by index of an array:

```
var a = ['first', 'second', 'third', 'fourth', 'fifth'];
console.log('Original Array:', a);
let position = 1;
let elementsToRemove = 2;
// Remove 'elementsToRemove' element(s) starting at 'position'
a.splice(position, elementsToRemove);
console.log('Modified Array:', a);
```

(x) Copy an Array :

```
var a = ['first', 'second', 'third', 'fourth'];
console.log('a:', a);

// Shallow copy array 'a' into a new object
let b = a.slice();

console.log('b:', b);
```

(xi) Sort an Array :

```
var a = ['c', 'a', 'd', 'b', 'aa'];
var b = [9, 2, 13, 7, 1, 12, 123];
// Sort in ascending lexicographical order using a built-in
a.sort();
b.sort();
console.log('a:', a);
console.log('b:', b);
var a = ['c', 'a', 'd', 'b', 'aa'];
var b = [9, 2, 13, 7, 1, 12, 123];

// Sort in descending lexicographical order using a compare function
a.sort(function(x, y) { return x < y; } );
b.sort(function(x, y) { return x < y; } );
console.log('a:', a);
console.log('b:', b);
```

```
var a = ['c', 'a', 'd', 'b', 'aa'];

// Sort in descending lexicographical order using a compare arrow function
a.sort((x, y) => x < y);

console.log('a:', a);
```

(xii) Iterate over an Array:

```
var a = ['first', 'second', 'third', 'fourth'];

for (let e of a) {
    console.log('e:', e);
}
```

Error Handling

JavaScript Errors

There are three types of errors in programming:

1. Syntax Error (Parsing Error)

In a traditional programming language, this type of error occurs at *compile time*; because JavaScript is an *interpreted* language, this type of error arises when the code is interpreted. When a syntax error occurs in JavaScript, only the code contained within the same *thread* as the syntax error is affected; independent code running in other threads will still be executed, as nothing in them depends on the code containing the error. For example, consider the following code containing a syntax error:

```
console.log("Hello"
```

This produces the following error: `SyntaxError: missing ) after argument list`. This is because we failed to add a closing parenthesis to our call to *console.log*.

2. Runtime Error (Exception)

Commonly referred to as *exceptions*, this type of error occurs during execution (i.e., after compilation or interpretation). Once a runtime error is encountered, an exception is *thrown* in the hope that it will be *caught* by a subsequent section of code containing instructions on how to recover from the error. Much like syntax errors, these affect the thread where they occured but allow other, independent threads to continue normal execution. For example, consider the following code containing a runtime error:

```
function sum(a, b) {}
add(2, 3)
```

This produces the following error: `ReferenceError: add is not defined`. This is because we attempted to call the *add* function without ever declaring and defining it.

3. Logical Error

These are some of the most difficult errors to isolate because they cause the program to operate without terminating or crashing, but the operations the code performs are not correct. Unlike syntax and runtime errors which arise due to some violation of the rules of the language, these errors occur when there is a mistake in your the code's logic.

Try, Catch, and Finally

The *try* block is the first step in error handling and is used for any block of code that is likely to raise an exception. It should contain one or more statements to be executed and is typically followed by at least one *catch clause* and/or the optional *finally clause*. In other words, the *try* statement has three forms:

- *try-catch*
- *try-finally*
- *try-catch-finally*

The *catch* block immediately follows the *try* block and is executed only if an exception is thrown when executing the code within the *try* block. It contains statements specifying how to proceed and recover from the thrown exception; if no exception is thrown when executing the *try* block, the *catch* block is skipped. If any statement within the *try* block (including a function call to code outside of the block) throws an exception, control immediately shifts to the catch clause.

It's important to note that we always want to avoid throwing an exception. It's best if the contents of the *try* block execute without issue but, if an exception is unavoidable, control passes to the *catch* block which should contain instructions that report and/or recover from the exception.

The *finally* block is optional. It executes after the *try* and *catch* blocks, but before any subsequent statements following these blocks. The *finally* block always executes, regardless of whether or not an exception was thrown or caught.

In the code below, the call to `getElement(arr, 4)` inside the *try* block will throw an exception because the code declaring was commented out. It's immediately followed by a *catch* block that catches the exception and prints the *message* associated with it (`arr is not defined`). Because the exception was caught, the program continues executing, printing the next line after the *catch* block (`The program continued executing!`).

```
"use strict"


function getElement(arr, pos) {
    return arr[pos];
}
//let arr = [1, 2, 3, 4, 5];
try {
    console.log(getElement(arr, 4));
}
catch (e) {
    console.log(e.message);
}
console.log("The program continued executing!");
```

Throw

We use the *throw* statement, denoted by the `throw` keyword, to throw an exception. There are two ways to do this, shown below.

1. `throw value`

We can throw an exception by following the keyword `throw` with some

that we wish to use for the exception being thrown.

2. `throw new Error(customError)`

We can throw an exception by following the keyword `throw` with `new Error(customError)`, where customError is the value we want for the property of the exception being thrown.