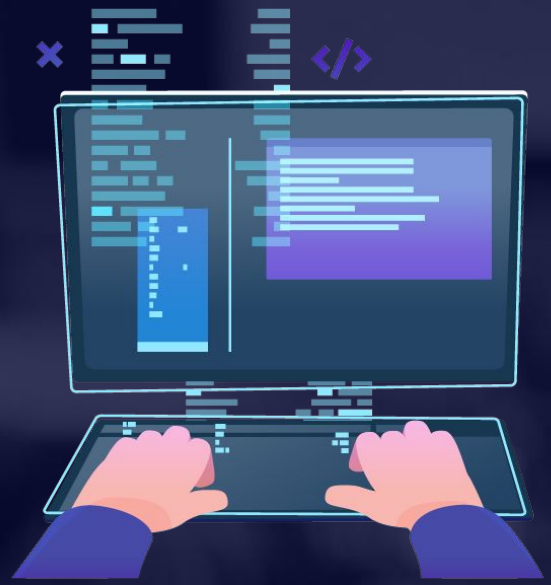
Syntax, Functions and Statements



Values, Operators, Parameters, Return Value, Arrow Functions

Table of Contents

Introduction to JavaScript

Dynamic Programming Language

- JavaScript is a **dynamic programming language**
  - Operations otherwise done at **compile-time** can be done at **run-time**
- It is **possible** to change the **type** of a variable or add new properties or methods to an object **while** the program is **running**
- In **static programming languages**, such changes are normally **not possible**

Data Types

- Seven data types that are **primitives**
  - **String -** used to represent textual data
  - **Number -** a numeric data type
  - **Boolean -** a logical data type
  - **Undefined** - automatically assigned to variables
  - **Null** - represents the **intentional absence** of any object value
  - **BigInt** - represent integers with **arbitrary precision**
  - **Symbol** - **unique** and **immutable** primitive value
- **Object**

Identifiers

- An **identifier** is a sequence of characters in the code that identifies a **variable**, **function**, or **property**
- An identifier **differs** from a string
  - in that a string is **data**, while an identifier is **part of** the **code**
- In JavaScript, identifiers are case-sensitive and can contain Unicode **letters**, **$**, **_**, and **digits** (0-9), but may **not** start with a digit

Variable Values

- Used to **store** data values
- Variables that are assigned a **non-primitive** value are given a **reference** to that value
- **Undefined** - a variable that has been declared with a keyword, but not given a value

```
let a;
console.log(a) //undefined
```

- **Undeclared** - a **variable** that hasn't been declared at all

```
console.log(undeclaredVariable);
//ReferenceError: undeclaredVariable is not defined
```

Variable Values

- **`let`**, **`const`** and **`var`** are used to declare variables
  - **`let`** - for **reassigning** a variable

```
let name = "George";
name = "Maria";
```

  - **`const`** - once assigned it **cannot** be modified

```
const name = "George";
name = "Maria";  // TypeError
```

  - **`var`** - defines a variable in the lexical scope **regardless** of block scope

```
var name = "George";
name = "Maria";
```

Dynamic Typing

- Variables in JavaScript are **not** directly **associated** with any particular **value type**
- Any variable **can** be assigned (and re-assigned) values of all types

```
let foo = 42;      // foo is now a number
foo = 'bar';       // foo is now a string
foo = true;        // foo is now a boolean
```

Strict Mode

- Strict mode - helps you to write **cleaner** code
- Strict mode is declared by adding "use strict";
  - Declared at the beginning of a **script**, it has **global scope**
  - Declared inside a **function**, it has **local scope**
- The "use strict" directive is only **recognized** at the **beginning** of a script or a function

Strict Mode Examples

```
"use strict";
x = 3.14; // This will cause an error because x is not declared
```

```
x = 3.14;       // This will NOT cause an error.

myFunction();

function myFunction() {
    "use strict";
    y = 3.14;   // This will cause an error
}
```

Fixed Values

- Fixed values - literals
  - **Array Literals**: list of zero or more **array elements**,
    enclosed in square brackets (**[ ]**)

```
let cars = ["Ford", "BMW", "Peugeot"];
let arrayLength = cars.length;  // 3
let secondCar = cars[1];        // "BMW"
```

Fixed Values

- **Object Literals**:
  - List of zero or more **pairs** of property names
  - Associated values of an object, enclosed in curly braces **{ }**

```
let car = { type: "Infinity", model: "QX80", color: "blue" };
let carType = car.type;
let carType = car["type"];  // Access property
car.year = 2018;
car["year"] = 2018; // Add new property
car.color = "black";
car["color"] = "black"; // Correct existing property
```

Arithmetic, Assignment, Comparison, Logical Operators

Arithmetic Operators

- **Arithmetic operators** - take numerical values (either literals or variables) as their operands
  - Return a single numerical value
    - Addition (**+**)
    - Subtraction (**-**)
    - Multiplication (*)
    - Division (**/**)
    - Remainder (**%**)
    - Exponentiation (**)

```
let a = 15;
let b = 5;
let c;
c = a + b; // 20
c = a - b; // 10
c = a * b; // 75
c = a / b; // 3
c = a % b; // 0
c = a ** b;//15^5 = 759375
```

Assignment Operators

- **Assignment operators - assign** a value to its left operand based on the value of the right operand

Name	Shorthand operator	Basic usage
Assignment	x = y	x = y
Addition assignment	x += y	x = x + y
Subtraction assignment	x -= y	x = x - y
Multiplication assignment	x *= y	x = x * y
Division assignment	x /= y	x = x / y
Remainder assignment	x %= y	x = x % y
Exponentiation assignment	x **= y	x = x ** y

Comparison Operators

Operator	Notation in JS
EQUAL value	==
EQUAL value and type	===
NOT EQUAL value	!=
NOT EQUAL value/type	!==
Greater than	>
Greater than OR EQUAL	>=
LESS than	<
LESS than OR EQUAL	<=

Comparison Operators

```
console.log(1 == '1'); // true
console.log(1 === '1');// false
console.log(3 != '3'); // false
console.log(3 !== '3');// true
console.log(5 < 5.5);  // true
console.log(5 <= 4);   // false
console.log(2 > 1.5);  // true
console.log(2 >= 2);   // true

console.log(5 ? 4 : 10);// 4
```

The "? " is a ternary operator

Truthy and Falsy Values

- "**truthy**" - a value that **coerces** to **true** when **evaluated** in a boolean context
- There are only **six** "**falsy**" values - **false**, **null**, **undefined**, **NaN**, **0** and **""**

```
function logTruthiness (val) {
    if (val) {
        console.log("Truthy!");
    } else {
        console.log("Falsy.");
    }
}
```

```
logTruthiness (3.14);      //Truthy!
logTruthiness ({});        //Truthy!
logTruthiness (NaN);       //Falsy.
logTruthiness ("NaN");     //Truthy!
logTruthiness ([]);        //Truthy!
logTruthiness (null);      //Falsy.
logTruthiness ("");        //Falsy.
logTruthiness (undefined); //Falsy.
logTruthiness (0);         //Falsy.
```

Logical Operators

- && (**logical AND**) - returns the leftmost "**false**" value

```
let val = true && 'yes' && 5 && null && false;
console.log(val); // null
let val = true && 'no' && 5 && 25 && 'yes';
console.log(val); // 'yes'
```

- || (**logical OR**) - returns the leftmost "**true**" value

```
let val = false || 0 || '' || 5 || 'hi' || true;
console.log(val); // 5
let val = false || '' || null || NaN || undefined;
console.log(val1); // undefined
```

Logical Operators

- ! (**logical NOT**) - Returns **false** if its single operand can be converted to **true**; otherwise, returns **true**

```
let val = !true
console.log(val); // false
let val = !false;
console.log(val); // true
```

Typeof Operator

- The **typeof** operator returns a string indicating the type of an operand

```
let val = 5;
console.log(typeof val);     // number
```

```
let str = 'hello';
console.log(typeof str);     // string
```

```
let obj = {name: 'Maria', age:18};
console.log(typeof obj);    // object
```

Instanceof Operator

- The **instanceof** operator returns **true** if the current object is an instance of the specified object

```
let cars = ["Saab", "Volvo", "BMW"];
console.log(cars instanceof Array);  // Returns true
console.log(cars instanceof Object); // Returns true
console.log(cars instanceof String); // Returns false
console.log(cars instanceof Number); // Returns false
```

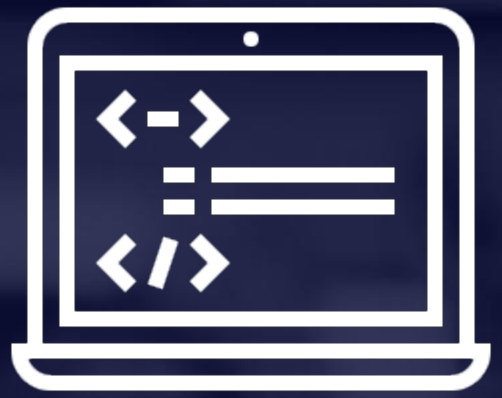
Some Interesting Examples

- Data Types

```
console.log(typeof NaN);                    //number
console.log(NaN === NaN);                   //false
console.log(typeof null);                   //object(legacy reasons)
console.log(null instanceof Object);        //false
console.log(new Array() == false);          //true
console.log(0.1 + 0.2);                     //0.30000000000000004
console.log((0.2 * 10 + 0.1 * 10) / 10); //0.3
```

- Truthy and Falsy values

```
let variable = [];                          //empty array
console.log(variable == false);             //evaluates true
if (variable) { console.log('True!') }; //variable evaluates
                                              to true
```



Declaring and Invoking

Functions

Functions

- **Function** - named list of instructions (statements and expressions)
- Can take **parameters** and return **result**
  - Function names and parameters use **camel case**
  - The **{** stays at the same line

```
function printStars(count) {
    console.log("*".repeat(count));
}
```

  - **Invoke** the function

```
printStars(10);
```

Declaring Functions

- Function declaration

```
function walk() {
    console.log("walking");
}
```

- Function expression

```
let walk = function (){
    console.log("walking");
}
```

- Arrow functions

```
let walk = () => {
    console.log("walking");
}
```

Parameters

- You can instantialized parameters with no value

```
function foo(a,b,c){
  console.log(a);
  console.log(b);
  console.log(c); //undefined
}
foo(1,2)
```

- The unused parameters are ignored

```
function foo(a,b,c){
  console.log(a);
  console.log(b);
  console.log(c);
}
foo(1,2,3,6,7)
```

Default Function Parameter Values

- Functions can have **default parameter** values

```
function printStars(count = 5) {
    console.log("*".repeat(count));
}
```

```
printStars(); // *****
```

```
printStars(2); // **
```

```
printStars(3, 5, 8); // ***
```

Function Overloading

- In C# / Java / C++ functions can be overloaded
  - **Function overloading** == same name, different parameters
- JavaScript (like Python and PHP) **does not support** overloading

```
function printName(firstName, lastName) {
  let name = firstName;
  if (lastName != undefined) {
     name += ' ' + lastName;
  }
  console.log(name);
}
```

Simulate overloading by parameter checks

Arguments

- Arguments - **object** which looks like array
- Through arguments you can **access parameters** that are **not** passed in the function
- In **arrow functions** you **don't** have **access** to arguments
- Changing the arguments object is **not** a good practice

```
function foo(a,b,c){
  console.log(arguments[0]); // 1
  console.log(arguments[4]); // 7
  console.log(arguments[3] + arguments[4]); // 13
  console.log(arguments);
// [Arguments] { '0': 1, '1': 2, '2': 3, '3': 6, '4': 7 }
}
foo(1,2,3,6,7)
```

First-class Functions

- First-class functions- a function can be passed as an **argument** to other functions
- Can be **returned** by another function and can be **assigned** as a value to a variable

```
function running() {
    return "Running";
}
function category(run, type) {
    console.log(run() + " " + type);
}
category(running, "sprint");
```

Callback function

Hoisting

- Variable and function declarations are **put into memory** during the **compile** phase, but stay exactly where you **typed** them in your code
- **Only declarations are hoisted**

Hoisting Variables

```
console.log(num); // Returns undefined
var num;
num = 6;
```

```
num = 6;
console.log(num); // returns 6
var num;
```

```
num = 6;
console.log(num); // ReferenceError: num is not defined
let num;
```

```
console.log(num); // ReferenceError: num is not defined
num = 6;
```

Hoisting Functions

```
run(); // running
function run() {
    console.log("running");
};
```

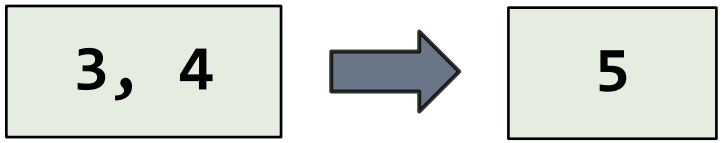
```
walk(); // ReferenceError: walk is not defined
let walk = function () {
    console.log("walking");
};
```

```
console.log(walk); //undefined
walk(); // TypeError: walk is not a function
var walk = function () {
    console.log("walking");
};
```

Nested Functions

- Functions can be **nested - hold other functions**
  - Inner functions have **access** to **variables** from **their parent**

```
function hypotenuse(m, n) { // outer function
    function square(num) {  // inner function
        return num * num;
    }
    return Math.sqrt(square(m) + square(n));
}
```



Problem: Sum / Inverse / Concatenate

- Using the aggregating function, calculate:
  - Sum of **elements**
    - e.g. `[1, 2, 4] = 1 + 2 + 4 = 7`
  - Sum of **inverse elements** ($1/a_i$)
    - e.g. `[1, 2, 4] = 1/1 + 1/2 +1/4 = 7/4 = 1.75`
  - **Concatenation** of elements
    - e.g. `['1', '2', '4'] = '1'+'2'+'4' = '124'`

Solution: Sum / Inverse / Concatenate

```
function aggregateElements(elements) {
    aggregate(elements, 0, (a, b) => a + b);
    aggregate(elements, 0, (a, b) => a + 1 / b);
    aggregate(elements, '', (a, b) => a + b);
    function aggregate(arr, initVal, func) {
        let val = initVal;
        for (let i = 0; i < arr.length; i++)
            val = func(val, arr[i]);
        console.log(val);
    }
}
```

Live Exercises

Summary

- Variables are used to store data references
  - `let`, `const` and `var` are used to declare variables
- Arithmetic operators take numerical values as their operands
- Functions can:
  - Take parameters and return result
  - Hold other functions inside them

Questions?

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © Kingsland University – https://kingslanduniversity.com


KINGSLAND
UNIVERSITY
THANK YOU