## Basics Syntax

### Statements, expressions, and comments:

| // This is a single-line comment  /\*  This is a  multi-line comment \*/  // Statement let x = 5;  // Expression let y = x + 3; |
| --- |

### Variables (var, let, const):

| // var (function-scoped) var a = 10; a = 20; // can be reassigned  // let (block-scoped, can be reassigned) let b = 30; b = 40;  // const (block-scoped, cannot be reassigned) const c = 50; // c = 60; // Error: Assignment to constant variable |
| --- |

### Data types (number, string, boolean, object, undefined, null):

| // Number let num = 10;  // String let str = "Hello, world!";  // Boolean let bool = true;  // Object let obj = {  name: "John",  age: 30,  isStudent: false };  // Undefined let und;  // Null let nul = null; |
| --- |

## Operators

### Arithmetic operators (+, -, \*, /, %)

| let a = 10; let b = 5;  console.log(a + b); // Addition: 10 + 5 = 15 console.log(a - b); // Subtraction: 10 - 5 = 5 console.log(a \* b); // Multiplication: 10 \* 5 = 50 console.log(a / b); // Division: 10 / 5 = 2 console.log(a % b); // Modulus (Remainder): 10 % 5 = 0 |
| --- |

### Comparison operators (==, ===, !=, !==, <, >, <=, >=)

| let x = 5; let y = 10;  console.log(x == y); // Equality: 5 == 10 => false  console.log(1 == ‘1’); // Equality: 1 == 1 (type coercion) => true  console.log(x === y); // Strict equality: 5 === 10 => false  console.log(1 == ‘1’); // Equality: 1 == ‘1’ (data type differences) => false  console.log(x != y); // Inequality: 5 != 10 => true console.log(x !== y); // Strict inequality: 5 !== 10 => true console.log(x < y); // Less than: 5 < 10 => true console.log(x > y); // Greater than: 5 > 10 => false console.log(x <= y); // Less than or equal to: 5 <= 10 => true console.log(x >= y); // Greater than or equal to: 5 >= 10 => false |
| --- |

### Logical operators (&&, ||, !)

| let p = true; let q = false;  console.log(p && q); // Logical AND: true && false => false console.log(p || q); // Logical OR: true || false => true console.log(!p); // Logical NOT: !true => false |
| --- |

### Assignment operators (=, +=, -=, etc.)

| let num = 10;  num += 5; // num = num + 5 => 10 + 5 console.log(num); // Output: 15  num -= 3; // num = num - 3 => 15 - 3 console.log(num); // Output: 12  num \*= 2; // num = num \* 2 => 12 \* 2 console.log(num); // Output: 24  num /= 4; // num = num / 4 => 24 / 4 console.log(num); // Output: 6  num %= 5; // num = num % 5 => 6 % 5 console.log(num); // Output: 1 |
| --- |

## Function-scoped vs Block-scoped variables

### Function-scoped variables:

var keyword were function-scoped. This means that the variable's scope is limited to the function in which it is declared. Consider the following example:

| function exampleFunction() {  var x = 10;  console.log(x); // Output: 10 } exampleFunction(); |
| --- |

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

In this example, the variable x is declared within the exampleFunction(). It cannot be accessed outside of the function.

### Block-scoped variables:

Starting from ES6, JavaScript introduced the let and const keywords for declaring variables. Variables declared with let and const are block-scoped, meaning their scope is limited to the block (enclosed within curly braces {}) in which they are declared. Consider the following example:

| if (true) {  let y = 20;  console.log(y); // Output: 20 } //console.log(y); // ReferenceError: y is not defined |
| --- |

In this example, the variable y is declared within the if block. It cannot be accessed outside of the block.

### Differences:

1. Scope: Function-scoped variables are accessible within the function in which they are declared, while block-scoped variables are accessible only within the block in which they are declared.
2. Hoisting: Variables declared with var are hoisted to the top of their function scope, meaning you can access them before they are declared (but they will have an initial value of undefined). Variables declared with let and const are also hoisted, but they are not initialized until their declaration is evaluated.
3. Re-declaration: Function-scoped variables declared with var can be re-declared within the same function scope without throwing an error. Block-scoped variables declared with let and const cannot be re-declared within the same block scope.
4. Access outside of block/function: Attempting to access a block-scoped variable outside of its block scope will result in a ReferenceError. Similarly, attempting to access a function-scoped variable outside of its function scope will also result in a ReferenceError.

## Common file extensions used in JavaScript development

1. .js: This is the most common extension for JavaScript files. Files with a .js extension contain JavaScript code that can be executed by a JavaScript engine, either in a web browser or in a server-side environment like Node.js.
2. .mjs: Introduced in ECMAScript 6 (ES6), the .mjs extension is used for ES module files. ES modules allow developers to organize their code into reusable modules, making it easier to manage large codebases. Files with a .mjs extension can use ES module syntax (import and export) and are treated as ES modules by JavaScript engines.

Tips: running mjs file requires --experimental-modules flag

| node --experimental-modules example.mjs |
| --- |

## Naming Conventions

1. Variable and Function Names:
   * Use camelCase for variable and function names.
   * Use descriptive names that convey the purpose or meaning of the variable or function.
   * Start function names with a verb when they perform an action (e.g., getUserData, calculateTotal).
   * Use nouns for variables that represent data (e.g., userName, productList).
2. Module Names:
   * Use lowercase letters for module names.
   * Use hyphens or underscores to separate words in module names if needed (e.g., my-module, my\_module).
   * Choose module names that are descriptive and reflect the functionality of the module.
3. Constants:
   * Use uppercase letters for constant names.
   * Separate words in constant names with underscores (e.g., MAX\_SIZE, API\_URL).
   * Constants are often used for values that are not expected to change during the execution of the program.
4. File Names:
   * Use lowercase letters for file names.
   * Separate words in file names with hyphens or underscores.
   * Use descriptive names that indicate the content or purpose of the file (e.g., user-controller.js, data-service.js).
5. Class Names:
   * Use PascalCase for class names (also known as upper camel case).
   * Class names should be descriptive and start with a capital letter (e.g., UserService, ProductController).
6. Callbacks and Promises:
   * Use meaningful names for callback functions and promise variables.
   * Use err as the first parameter in callback functions to represent errors, if applicable.
7. Event Names:
   * Use lowercase letters for event names.
   * Separate words in event names with hyphens or underscores.
8. Middleware Functions:
   * Use descriptive names for middleware functions that indicate their purpose or functionality (e.g., authenticate, validateInput).
9. Route Names:
   * Use descriptive names for routes that reflect the resource or functionality they represent (e.g., /users, /products).
10. Environment Variables:
    * Use uppercase letters for environment variable names.
    * Separate words in environment variable names with underscores (e.g., DATABASE\_URL, API\_KEY).

### File Naming considerations

1. Common Convention: It's a common convention, especially in languages like Java or C#, to name a file after the class it contains. For example, a class named UserService would typically be defined in a file named UserService.js or user-service.js. This convention helps developers quickly identify the content of a file.
2. Multiple Classes in a File: In JavaScript, you can define multiple classes or modules in a single file. In such cases, naming the file after one of the classes may not be sufficient or accurate. You might choose a name that reflects the primary purpose of the module or includes all the classes it contains.

For example:

| // File: UserService.js  class User {  // Class definition } class UserService {  // Class definition }  module.exports = {  User,  UserService }; |
| --- |

In this case, the file name (UserService.js) doesn't necessarily match any specific class name. Instead, it reflects the purpose of the module.