JavaScript とは

JavaScript は、ウェブページを動的に操作し、対話的な体験を提供するためのプログラミング言語です。ウェブ開発において、HTML（構造）、CSS（スタイル）、JavaScript（動作）の 3 つが主要なコンポーネントとして使用されます。JavaScript は、ウェブブラウザ内で動作するクライアントサイド言語として広く使われています。

JavaScript の特徴

対話的なウェブページ: JavaScript を使用することで、ユーザーとウェブページが対話的にやり取りできるようになります。ユーザーの操作に応じて要素を変更したり、新しいコンテンツを表示したりすることができます。

動的なコンテンツ: ウェブページをリアルタイムで更新することができます。例えば、フォームの入力に基づいて結果を計算し表示するなどが可能です。

イベント駆動: JavaScript はイベント駆動のプログラミングモデルを採用しています。ユーザーのアクション（クリック、マウスオーバーなど）や、時間の経過に応じたイベントに反応してコードを実行できます。

クロスプラットフォーム: JavaScript は主にブラウザ内で実行されますが、Node.js と呼ばれる環境を使用することで、サーバーサイドでも動作することができます。

JavaScript 入門: 変数

変数とは何か？

変数は、プログラム中でデータを保存・操作するための入れ物です。名前を付けてデータを保持することができ、その名前を使ってデータを呼び出すことができます。

変数の宣言方法

JavaScript では、3 つのキーワードを使用して変数を宣言することができます: **var**, **let**, および **const**。それぞれの違いについて見てみましょう。

①**var**: これは古い変数宣言の方法ですが、古いブラウザの互換性のために残っています。ただし、スコープの扱いがやや特殊です。

②**let**: これはブロックスコープ内で有効な変数を宣言します。同じ名前の変数を同じブロック内で再宣言することはできません。

③**const**: **const** は変更不可能な定数を宣言します。一度値を代入すると、後から変更することはできません。

④

変数の例

```
// letを使った変数宣言
let age = 25;
console.log(age); // 25

// constを使った定数宣言
const name = "太郎";
console.log(name); // 太郎

// varを使った変数宣言（あまりおすすめされません）
var score = 90;
console.log(score); // 90
```

変数名のルール

・変数名はアルファベット、数字、アンダースコア（_）、またはドル記号（$）を使用できますが、数字で始めることはできません。
・変数名は大文字と小文字が区別されます（name と Name は異なる変数です）。
・変数名はわかりやすく、意味のある名前を付けることが重要です。

JavaScript 入門: 変数、定数、let、const、var の違い

変数と定数の違い

変数: データを保存・操作するための入れ物で、値が変更可能です。
定数: データを保存・操作するための入れ物で、値が一度設定されたら変更できません。

let と const、var の違い

・let: ブロックスコープ内で有効な変数を宣言します。同じ名前の変数を同じブロック内で・再宣言できません。
・const: 変更不可能な定数を宣言します。一度値を代入すると、後から変更することはできません。
・var: 古い変数宣言の方法で、スコープの特殊な扱いがあります。ブロックスコープを無視して関数スコープとして扱われます。

データ型とは

データ型は、プログラム内で使用されるデータの種類を定義するための仕組みです。各データ型は、メモリ内でのデータの保存方法や操作方法に影響を与えます。プログラムがデータを効果的に扱うためには、適切なデータ型を選択することが重要です。

JavaScript にはいくつかの基本的なデータ型があります。

①

数値型（Number）: 数値を表すためのデータ型です。整数や浮動小数点数を扱えます。

```
let age = 25;        // 整数
let height = 175.5; // 浮動小数点数
```

②

文字列型（String）: 文字の連なりを表すためのデータ型です。

```
let name = "Alice";
```

③

ブール型（Boolean）: 真（true）または偽（false）の値を表すためのデータ型です。

```
let isStudent = true;
```

④

未定義型（Undefined）: 値が定義されていないことを示すためのデータ型です。

```
let variable;
```

⑤

ヌル型（Null）: 値が存在しないことを示すためのデータ型です。

```
let empty = null;
```

⑥

シンボル型（Symbol）: シンボルを表すためのデータ型で、一意な識別子として使用されます（ES6 以降）。

```
let person = {
    firstName: "John",
    lastName: "Doe",
    age: 30,
    isStudent: false
};
```

```
let id = Symbol("uniqueID");
```

⑦

オブジェクト型

JavaScript では、オブジェクトも重要なデータ型です。オブジェクトは複数の値や関数をグループ化して扱うための構造です。

⑧

配列型（Array）

配列は、複数の値を順序付けて保存するためのデータ型です。

```
let colors = ["red", "green", "blue"];
```

⑨

関数型（Function）

関数は、特定のタスクを実行するための再利用可能なコードブロックです。

```
function greet(name) {
    console.log("こんにちは、" + name + "さん！");
}
```

型変換

JavaScript では、異なるデータ型間で変換を行うこともできます。

```
let num = 10;
let str = "5";

let result = num + Number(str); // 数値型としてstrを変換
console.log(result); // 15
```

型変換には明示的な方法（例: Number()、String()、Boolean() 関数）と暗黙的な方法（例: 数値と文字列の結合）があります。

算術演算子

算術演算子は、数値の計算を行うための記号です。

算術演算子の詳細

加算（+）

加算演算子は、数値を足し合わせるために使用されます。

```
let x = 10;
let y = 5;

let result = x + y; // 15
```

減算（-）

減算演算子は、数値を引き算するために使用されます。

```
let x = 10;
let y = 5;

let result = x - y; // 5
```

乗算（*）

乗算演算子は、数値を掛け合わせるために使用されます。

```
let x = 10;
let y = 5;

let result = x * y; // 50
```

除算（/）

除算演算子は、数値を割り算するために使用されます。

```
let x = 10;
let y = 5;

let result = x / y; // 2
```

剰余（%）

剰余演算子は、2 つの数値の剰余（余り）を計算するために使用されます。

```
let x = 10;
let y = 3;

let result = x % y; // 1
```

インクリメント（++）

インクリメント演算子は、変数の値を 1 増やすために使用されます。

```
let x = 5;
x++; // xは6になる
```

デクリメント（--）

デクリメント演算子は、変数の値を 1 減らすために使用されます。

```
let x = 5;
x--; // xは4になる
```

数値の場合

数値に対して算術演算子を使用すると、期待通りの数学的な計算が行われます。

```
let x = 10;
let y = 5;

let addition = x + y; // 15
let subtraction = x - y; // 5
let multiplication = x * y; // 50
let division = x / y; // 2
let remainder = x % y; // 0
```

文字列の場合

文字列に対して算術演算子を使用すると、主に「+」演算子が文字列の連結に使われます。

```
let firstName = "John";
let lastName = "Doe";

let fullName = firstName + " " + lastName; // "John Doe"
```

このように、文字列と文字列を「+」演算子で連結することができます。ただし、他の算術演算子（加算、減算など）は、文字列に対して使用すると、自動的に数値への変換が試みられます。

この場合は？

```
let strNum = "10";
let num = 5;

let result = strNum + num; // "105"
```

上記の例では、**strNum** は文字列ですが、**num** は数値です。文字列と数値を「+」演算子で結合しようとすると、数値が文字列に変換されてから連結されます。

JavaScript 入門: if 文

比較演算子

比較演算子は、2 つの値を比較して、その関係が真（true）か偽（false）かを評価するために使用される演算子です。

等しい（==）

この演算子は、2 つの値が等しいかどうかを比較します。

```
let x = 5;
let y = "5";

console.log(x == y); // true
```

厳密に等しい（===）

この演算子は、値と型が両方とも等しいかどうかを比較します。

```
let x = 5;
let y = "5";

console.log(x === y); // false
```

等しくない（!=）

この演算子は、2 つの値が等しくないかどうかを比較します。

```
let x = 5;
let y = 10;

console.log(x != y); // true
```

厳密に等しくない（!==）

この演算子は、値または型が異なる場合に真を返します。

```
let x = 5;
let y = "5";

console.log(x !== y); // true
```

大なり（>）、小なり（<）、大なりイコール（>=）、小なりイコール（<=）

これらの演算子は、数値間の大小関係を比較します。

```
let a = 10;
let b = 15;

console.log(a > b); // false
console.log(a < b); // true
console.log(a >= b); // false
console.log(a <= b); // true
```

JavaScript 入門:if 文

if 文

if 文は、特定の条件が満たされた場合に、あるコードブロックを実行するための制御構造です。

if 文の基本構造

if 文は以下のような基本構造を持ちます。

```
if (条件式) {
    // 条件が真の場合に実行されるコード
}
```

if 文の例

以下は、if 文を使用した例です。

```
let age = 20;

if (age >= 18) {
    console.log("成人です");
}
```

上記の例では、age の値が 18 以上である場合、"成人です" というメッセージが表示されます。

if - else 文

単純な条件だけでなく、条件が満たされない場合の処理も実行したい場合、**if** 文と組み合わせて **else** 文を使用できます。

```
let age = 15;

if (age >= 18) {
    console.log("成人です");
} else {
    console.log("未成年です");
}
```

if - else if 文

複数の条件に対して異なる処理を行いたい場合、**else if** 文を追加できます。

```
let score = 75;

if (score >= 90) {
    console.log("優秀です");
} else if (score >= 70) {
    console.log("良いです");
} else {
    console.log("頑張りましょう");
}
```

ネストされた if 文

if 文は、他の **if** 文の中にネストして使用することもできます。これにより、より複雑な条件分岐を実現できます。

```
let isWeekend = true;
let isSunny = true;

if (isWeekend) {
    if (isSunny) {
        console.log("週末で晴れています");
    } else {
        console.log("週末ですが天気は悪いです");
    }
} else {
    console.log("平日です");
}
```

論理式

論理式（または論理演算式）は、論理演算子を使用して複数の条件を組み合わせることで、真（true）または偽（false）の結果を得る式です。

論理積（&&）

論理積演算子は、複数の条件がすべて真である場合に結果が真になります。

```
let age = 25;
let hasLicense = true;

if (age >= 18 && hasLicense) {
    console.log("運転できます");
} else {
    console.log("運転できません");
}
```

論理和（||）

論理和演算子は、いずれかの条件が真である場合に結果が真になります。

```
let isWeekend = true;
let hasHoliday = false;

if (isWeekend || hasHoliday) {
    console.log("休みです");
} else {
    console.log("仕事です");
}
```

論理否定（!）

論理否定演算子は、条件の真偽を反転させます。

```
let isSunny = true;

if (!isSunny) {
    console.log("曇りです");
} else {
    console.log("晴れです");
}
```

これらの論理演算子を組み合わせて、複雑な論理式を作成できます。

また、カッコ () を使用して論理式の優先順位を明示的に指定することもできます。

```
let x = 10;
let y = 5;
let z = 3;

if ((x > y) && (y > z)) {
    console.log("条件成立");
} else {
    console.log("条件不成立");
}
```

switch 文

switch 文は、特定の式の値に基づいて、複数の可能な実行パスの中から 1 つを選択して実行するための制御構造です。複数の **if** 文を使うよりもコードを簡潔に保ちたい場合や、特定の値に応じて処理を分岐する必要がある場合に使用されます。以下に **switch** 文の詳細な説明を提供します。

switch 文の基本構造

switch 文は以下のような基本構造を持ちます。

```
switch (式) {
    case 値1:
        // 値1の場合の処理
        break;
    case 値2:
        // 値2の場合の処理
        break;
    // 他のケースも同様に続く
    default:
        // どのケースにも該当しない場合の処理
}
```

式 の評価結果と、各 **case** の値を比較し、一致する場合にその **case** 内のコードブロックが実行されます。**break** 文は、一致した **case** ブロックの実行を終了し、**switch** 文全体から抜け出るために使用されます。

switch 文の例

以下は、**switch** 文の基本的な例です。

```
let day = 3;
let dayName;

switch (day) {
    case 1:
        dayName = "日曜日";
        break;
    case 2:
        dayName = "月曜日";
        break;
    case 3:
        dayName = "火曜日";
        break;
    case 4:
        dayName = "水曜日";
        break;
    case 5:
        dayName = "木曜日";
        break;
    case 6:
        dayName = "金曜日";
        break;
    case 7:
        dayName = "土曜日";
        break;
    default:
        dayName = "無効な日付";
}
```

この例では、変数 **day** の値に基づいて、対応する曜日の名前を **dayName** に代入しています。

switch 文の注意点

・各 **case** の末尾には **break** 文を置くことで、一致したケースの処理が完了した後に **switch** 文を終了させます。**break** を省略すると、一致したケース以降のケースも実行されます（フォールスルー）。

・**default** ケースは、どの **case** にも一致しない場合のデフォルトの処理を指定するために使用されます。

JavaScript 入門:for 文

for 文

for 文は、特定の条件が満たされる間、あるいは特定の回数だけコードブロックを繰り返し実行するための制御構造です。

for ループの基本構造

for ループは以下のような基本構造を持ちます。

```
for (初期化式; 条件式; 更新式) {
    // ループ内で実行されるコード
}
```

①初期化式

ループ変数を初期化するための式です。通常、ループ変数の初期値を設定します。

②条件式

ループが継続するかどうかを判断するための式です。条件式が真（true）の間、ループは続きます。

③更新式

各ループの終了時に実行される式です。通常、ループ変数の更新を行います。

for ループの例

以下は、for ループの基本的な例です。

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

この例では、i はループ変数です。初期値は 0 から始まり、条件式が i < 5 の間、ループが繰り返されます。各ループで i が更新されて 1 ずつ増加し、結果として 0 から 4 までの数字がコンソールに表示されます。

特定の回数だけ繰り返す

```
for (let i = 0; i < 3; i++) {
    console.log("Hello");
}
```

逆順で繰り返す

```
for (let i = 10; i >= 1; i--) {
    console.log(i);
}
```

JavaScript 入門:配列

配列

配列は、複数の値を一つの変数に格納するデータ構造で、それぞれの値にはゼロベースの番号（インデックス）が割り当てられます。

配列の作成

JavaScript の配列は、[] 内にカンマで区切られた要素を記述して作成します。

```
let fruits = ["りんご", "バナナ", "オレンジ"];
```

配列の要素へのアクセス

配列の要素には、インデックスを使用してアクセスします。インデックスは 0 から始まります。

```
console.log(fruits[0]); // "りんご"
console.log(fruits[1]); // "バナナ"
console.log(fruits[2]); // "オレンジ"
```

配列の長さ

配列の要素数は length プロパティで取得できます。

```
console.log(fruits.length); // 3
```

配列の要素の追加・変更・削除

配列の要素は、インデックスを指定して追加・変更・削除できます。

```
// 要素の追加
fruits.push("グレープ");

// 要素の変更
fruits[1] = "マンゴー";

// 要素の削除
fruits.splice(0, 1); // インデックス0から1つの要素を削除
```

配列の繰り返し処理

配列の要素を繰り返し処理するために、for ループや forEach メソッドなどが利用されます。

```
// forループを使用した繰り返し
for (let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}

// forEachメソッドを使用した繰り返し
fruits.forEach(function(fruit) {
    console.log(fruit);
});
```

for...of ループ

ES6 以降、**for...of** ループを使用して、配列などの反復可能なオブジェクトを簡潔に処理できます。

```
let numbers = [1, 2, 3];

for (let num of numbers) {
    console.log(num);
}
```

多次元配列

配列の中に別の配列を要素として持つことで、多次元配列を作成できます。

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

console.log(matrix[1][2]); // 6
```

配列のメソッド

JavaScript の配列は多くの組み込みメソッドを持っており、これらを使用して配列を操作できます。代表的なメソッドには push、pop、shift、unshift、splice、concat、join、slice などがあります。

```
let numbers = [1, 2, 3, 4, 5];

numbers.push(6); // 末尾に要素を追加
numbers.pop();   // 末尾の要素を削除
numbers.shift(); // 先頭の要素を削除
numbers.unshift(0); // 先頭に要素を追加
```

JavaScript 入門:オブジェクト

オブジェクト

オブジェクトは、関連する情報をグループ化するためのデータ構造で、名前（キー）と値（プロパティ）のペアからなります。

オブジェクトの作成

JavaScript のオブジェクトは、中括弧 {} 内にプロパティを記述して作成します。

```
let person = {
    name: "John",
    age: 30,
    city: "New York"
};
```

オブジェクトのプロパティへのアクセス

オブジェクトのプロパティには、ドット . 演算子またはブラケット [] 演算子を使用してアクセスできます。

```
console.log(person.name); // "John"
console.log(person["age"]); // 30
```

オブジェクトのプロパティの追加・変更・削除

オブジェクトのプロパティは、新しいプロパティの追加や既存のプロパティの変更、削除が可能です。

```
// プロパティの追加
person.gender = "Male";

// プロパティの変更
person.age = 31;

// プロパティの削除
delete person.city;
```

オブジェクトのプロパティの繰り返し処理

オブジェクトのプロパティを繰り返し処理するために、for...in ループが使用されます。

for...in ループ

for...in ループは、オブジェクトのプロパティを反復処理するために使用されます。

```
let person = {
    name: "John",
    age: 30,
    city: "New York"
};

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

オブジェクトのネスト

オブジェクトは、他のオブジェクトをプロパティとして持つことができます。

```
let car = {
    make: "Toyota",
    model: "Camry",
    year: 2022,
    owner: {
        name: "Alice",
        age: 25
    }
};
```

JavaScript 入門:関数

関数

特定の処理や計算を実行するためにまとめられたコードのブロックです。
関数はプログラム内で再利用可能なコードを作成するための重要なツールであり、コードの可読性と保守性を向上させるのに役立ちます。

関数の作成

JavaScript の関数は、function キーワードを使用して作成します。関数には名前があり、関数名の後に括弧 () が続きます。関数の処理は中括弧 {} 内に記述します。

```
function greet(name) {
    console.log("Hello, " + name + "!");
}
```

上記の例では、greet という名前の関数が定義されています。この関数は name というパラメータを受け取り、その値を用いて挨拶を表示します。

関数の呼び出し

関数を呼び出すことで、関数内のコードが実行されます。関数を呼び出す際には、関数名の後ろに括弧 () を付けて、必要な引数（関数に渡す値）を指定します。

```
greet("Alice"); // "Hello, Alice!"
```

関数の戻り値

関数は、処理の結果として値を返すことができます。関数内で return 文を使用して戻り値を指定します。

```
function add(a, b) {
    return a + b;
}

let sum = add(5, 3); // 8
```

上記の例では、add 関数が 2 つの引数を受け取り、それらの値を足して戻り値として返し

ています。

パラメータと引数

関数のパラメータは、関数定義内で宣言される変数のようなものです。引数は、関数を呼び出す際に渡される値です。関数が呼び出されると、パラメータに引数の値が割り当てられます。

```
function multiply(x, y) {
    return x * y;
}

let result = multiply(4, 3); // 12
```

無名関数と即時関数

JavaScript では、無名関数（名前のない関数）を定義して使用することもできます。無名関数は通常、変数に代入して使用されることが多いです。

```
let sayHello = function(name) {
    console.log("Hello, " + name + "!");
};

sayHello("Bob"); // "Hello, Bob!"
```

また、無名関数を定義して即座に呼び出す場合、即時関数として知られています。

```
(function() {
    console.log("This is an immediately invoked function expressio
})();
```

高階関数

JavaScript では、関数を引数として受け取る関数や、関数を戻り値として返す関数を作成できます。これを高階関数と呼びます。高階関数は、コールバック関数や関数型プログラミングの実現に用いられます。

```
function operate(a, b, callback) {
    return callback(a, b);
}

function add(x, y) {
    return x + y;
}

function subtract(x, y) {
    return x - y;
}

let result1 = operate(5, 3, add);      // 8
let result2 = operate(5, 3, subtract); // 2
```

アロー関数の基本構文

アロー関数は次のような基本構文を持ちます。

```
const functionName = (パラメータ1, パラメータ2, ...) => {
    // 関数の本体
    return 返り値;
};
```

通常の関数と異なり、アロー関数では {} と return キーワードを省略できる場合があります。また、引数が 1 つの場合は () も省略できます。

アロー関数の例

以下は、通常の関数定義とアロー関数の比較例です。

通常の関数定義：

```
function add(x, y) {
    return x + y;
}
```

アロー関数：

```
const add = (x, y) => x + y;
```

また、引数が 1 つの場合は（）を省略できます。

```
const square = x => x * x;
```

JavaScript 入門:DOM 操作

DOM

DOM（Document Object Model）は、HTML や XML 文書のためのプログラミングインターフェースであり、これを使用して文書の構造や内容にアクセスし、変更することができます。JavaScript を使用して DOM と連携し、ページの動的な操作やユーザーとの対話を実現できます。
以下はどんな操作があるのか例をまとめています。
それぞれの具体的な使い方は処理名を検索して調べましょう。

1. 要素の取得

```
// id属性を使って要素を取得
const elementById = document.getElementById('myElementId');

// クラス名を使って要素を取得
const elementsByClassName = document.getElementsByClassName('myClassName');

// タグ名を使って要素を取得
const elementsByTagName = document.getElementsByTagName('div');

// CSSセレクタを使って要素を取得
const elementBySelector = document.querySelector('#myElementId');

// CSSセレクタを使って要素を取得（複数）
const elementsBySelectorAll = document.querySelectorAll('.myClassName');
```

2. 要素の作成・追加・削除

```
// 新しい要素を作成
const newElement = document.createElement('div');

// 親要素に新しい要素を追加
parentElement.appendChild(newElement);

// 親要素から子要素を削除
parentElement.removeChild(childElement);
```

3. テキストと属性の操作

```
// 要素のテキストを変更
element.textContent = '新しいテキスト';

// 要素の属性を設定
element.setAttribute('id', 'newId');

// 要素の属性を取得
const attributeValue = element.getAttribute('id');

// 要素の属性を削除
element.removeAttribute('id');
```

4. クラスの追加・削除・切り替え

```
// 要素にクラスを追加
element.classList.add('newClass');

// 要素からクラスを削除
element.classList.remove('oldClass');

// クラスがなければ追加、あれば削除
element.classList.toggle('active');
```

5. 要素のスタイルの変更

```
// 要素のスタイルを変更
element.style.color = 'blue';
element.style.fontSize = '16px';
```

6. イベントリスナーの設定

```
// クリックイベントに対するリスナーを設定
element.addEventListener('click', () => {
  console.log('クリックされました！');
});
```

7. 要素の位置の取得

```
// 要素の位置を取得
const rect = element.getBoundingClientRect();
console.log('上:', rect.top, '右:', rect.right, '下:', rect.bottom, '左:', rect.left);
```

8. 要素の非表示・表示

```
// 要素を非表示にする
element.style.display = 'none';

// 要素を表示する
element.style.display = 'block';
```