第1章 JavaScriptの基本構文

JavaScriptは、ウェブブラウザ上で動作する代表的なスクリプト言語であり、近年ではサーバーサイド開発にも使用されている。本章ではまず、もっとも基本的な構文に

明朝系の本文フォントで、強調スタイルを同シリーズの明朝にすると、見づらくなることが多い

1.1 変数宣言

JavaScriptにおける変数宣言は、**初学者にとって最も基本的でありながら、実は極めて奥が深く、また仕様変更と歴史的背景を理解しなければ正しく使い分けることが難しい**領域でもあるという事実を、我々はまず強調しておきたい。たとえば、varというキーワードはES5以前から存在する伝統的な宣言方法であり、関数スコープという一見扱いやすそうで実はトラップの温床になりやすい特徴を持っているが、これに対してES6（ECMAScript 2015）で導入されたletとconstは、ブロックスコープを備え、より厳密なスコープ管理が可能になった一方で、開発者がその使い分けを正確に理解しないまま形式的に使ってしまうことで、**かえってコードの可読性や意図の伝達性が損なわれるケース**も散見される。

特に、定数を示すconstであっても、参照型（Object, Arrayなど）の場合は中身の変更が可能であるという仕様が初学者に混乱をもたらす要因になっており、「constなのに変更できるってどういうこと？」という疑問が頻出することからもわかるように、単純に「変更不可」とだけ理解してしまうことの危険性を理解する必要がある。さらに、**型推論に依存した宣言は、静的型付け言語に慣れた開発者にとっては直感的でない部分があり、TypeScriptなどの導入が視野に入る段階では、変数宣言という行為そのものに明示的な型情報を加えることの重要性が再確認されるが、あくまで本書の範囲ではJavaScriptの素の文法にフォーカスするため**、まずはvar、let、constのそれぞれの意味と、使い分けの基本原則、つまり「原則const、再代入が必要ならlet、varは基本使わない」という一般的なルールを押さえつつ、それぞれの実行コンテキストにおける動作を理解するところから始めたい。

```
// 変数の宣言と基本的な代入
var message = "こんにちは、JavaScript!";
let count = 0;
const PI = 3.14159;

// 関数の定義と呼び出し
function greet(name) {
  return "こんにちは、" + name + "さん!";
}

console.log(greet("太郎")); // => こんにちは、太郎さん!

// if文による条件分岐
if (count === 0) {
  console.log("カウントはゼロです。");
} else {
  console.log("カウントはゼロではありません。");
}
```

特徴の強いフォントは一見、新鮮ですが可読性に問題がある場合があり、複数行にわたる段落では避けたほうが無難

上記の例では、関数**greet**が名前を引数に取り、挨拶文を返している。これはJavaScriptの関数定義の最も基本的なスタイルである。また、**console.log()**はデバッグ用途に頻繁に使用される関数であり、これを使うことで変数の状態や処理の流れを

本文中にコードフォントを使う場合でも、フォントの太さ（ウェイト）感があっていないとチグハグな印象

確認することができる。

ごちゃごちゃした背景があると、可読性が落ちる
ツメや柱は機能を大事に

1.2 変数スコープと関数のネスト

JavaScriptにおける変数のスコープは、その宣言方法によって大きく異なる。特に、関数内で宣言されたletやconstはブロック単位のスコープを持つため、意図しないバグを引き起こしやすい。

雰囲気とあわないフォント

▸▸▸ column: varの怖い話

なお、varで宣言された変数は関数スコープとなるため、ブロックをまたいで値が残ることがある。これによって、forループ内で非同期処理を行う場合に想定外の挙動が発生する。以下は悪い例である：

```
for (var i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(i); // 3, 3, 3
  }, 100);
}
```

※ これを防ぐには、letを使ってブロックスコープにする必要がある。

特色文字も可読性を落とす原因になりがちなので、部分的な使用にとどめるべき

明朝系の白抜き文字は読みづらいことが多い。ゴシックでも、細いと可読性が落ちる

1.3 非同期処理

JavaScriptにおける非同期処理というものは、プログラミング初心者にとっては非常に難解に映るものであり、**特にsetTimeoutやsetIntervalに代表されるコールバックベースの処理から、近年ではPromiseオブジェクトを活用した非同期制御へと進化し、さらにその先にはasyncおよびawaitという構文糖衣が登場**したことにより、コードの可読性と保守性の向上が図られる一方で、実行タイミングの直感的な理解を逆に損なうケースもあるため、単純に「awaitで書けばOK」とは言い切れず、むしろ非同期の根本的なモデル、すなわちJavaScriptの**イベントループおよびマイクロタスク／マクロタスク**の挙動について十分な理解が求められるわけだが、それを理解するにはNode.jsの内部動作やブラウザの実行環境との違いにも言及せざるを得ず、結果として「JavaScriptは簡単そうに見えて実は深い」という、よく言われる印象が現実味を帯びてくるのであり、特にこれからJavaScriptを学ぼうという初学者にとっては、**この非同期処理の壁を越えられるかどうかが最初の関門であり、それを乗り越えるためには理論だけでなく実践を通じた学習、つまり小さなサンプルコードを自分で書き、挙動を試し、予測と結果が一致するかを確認しながら進めるという地道な積み重ねが**重要であり、しかしながらそのような努力をしないまま流行りのフレームワークに飛びついてしまうと、いざエラーが出たときに何が原因なのかまったく見当がつかず、Stack Overflowを右往左往することになりがちで、そうならないためにも、本書では非同期処理の基礎から丁寧に解説し、実例と共にエラーの対処法までカバーすることを目指しており、読者諸

文字のつまりすぎも読みづらいが、アキすぎも間が抜ける。太字もさらにわかりづらい。

きたいと切に願うものである。

フロントエンド開発におけるJavaScriptの役割は年々拡大しており、かつては静的HTMLにちょっとしたインタラクションを加えるだけだったスクリプトが、いまやReactやVue、あるいはより抽象度の高いNext.jsのようなフレームワークを通じて、**仮想DOMによるレンダリングやSSR（Server Side Rendering）、さらにはHydrationやCode Splittingといったパフォーマンスチューニング技法にまで及んでいる**ことを考えると、もはやJavaScriptを「ちょっとした補助的な言語」

全体的に余白が小さく（版面が大きい）息が詰まる