

# Chapter

# A

## 「データ型」 を深く理解する

Web デザイナーが JavaScript に取り組む際に最もわかりづらいのは「データ型」です。

データ型はプログラミング特有の考え方で、HTML/CSS では基本的に考慮されません。

PDF 版特別付録では、書籍で解説しきれなかった「データ型」について、サンプルを動かしながら深く理解していきます。

データ型を学ぶことで、JavaScript へのハードルが一気に下がります。

また、Vue.js でのプログラミングが増えてくると、それに伴いバグとの格闘も増えてきます。

データ型の知識は、デバッグ（ソースコードからバグを取る作業）の際にも有効です。

それでは、学習を進めてまいりましょう！

### **A-1** JavaScript の「データ型」をおさえよう

A-1

# JavaScript の「データ型」をおさえよう

Web ページを動かすためには、検索ボックスに入力された検索ワードや検索結果の商品リストなどのさまざまなデータを、さまざまな場所で使います。たとえば、検索ボックスに入力された検索ワードは、検索ボックスに表示するときにも利用しますし、検索エンジンに問い合わせをするときにも利用します。値を使い回す場面で活躍してくれるのが変数です。変数にはさまざまな情報(文字列・数字・繰り返しリストなど、データ型と呼ばれます)を記憶することができます。さあ、プログラミングの必須知識のひとつである「変数」を確認していきましょう！

## 1 おみくじアプリでの利用方法

JavaScript の変数 (情報を記憶しておくための箱) は、var で宣言します。おみくじアプリでは、以下の部分です。

リスト A.1 変数宣言

```
001 //後で利用するために、ボタンの情報を取得して、「button」という名前の箱に保存しておく
002 var button = document.getElementById("button");
```

変数は、JavaScript を理解する上で重要な概念です。しっかりおさえて先に進みましょう。

## 2 プリミティブ型と参照型の違いについて

書籍本文の P.17、P.18 で解説したプリミティブ型と参照型について、説明より実際の動作を確認すると理解しやすいため、サンプルを動かしながら確認してみにします。Chrome デベロッパーツールに以下のとおり入力してみましょう (リスト A.2)。

リスト A.2 プリミティブ型と参照型の違い

```
001 var a = 1; //プリミティブ型 数値そのものを代入
002 var b = 2;
003 a = b;      //aにbを代入
004 //※※console.log()は確認用にWebコンソールにメッセージを出力する※※
005 console.log("a: " + a); //a: 2
006 console.log("b: " + b); //b: 2
007 //コピー先の値を変更しても、コピー元の値には影響しない (--- 要確認1 ---)
008 b = 999; //コピー先の変数 (b) を変更
009 console.log("a: " + a); //a: 2
010 console.log("b: " + b); //b: 999 -> コピー先の変数 (b) の値のみ変更され、コピー元の変数 (a) には影響しない
011 var list1 = [1, 2, 3]; //参照型 参照先を宣言
012 var list2 = list1;     //参照を代入
013 //コピー元の値を変更すると、コピー先の変数にも影響する (--- 要確認2 ---)
014 list1[1] = 99;         //コピー元の変数の 2番目 の値を 2->99 に変更する
015 console.log("list1: " + list1.toString()); //list1: 1, 99, 3
016 console.log("list2: " + list2.toString()); //list2: 1, 99, 3 -> コピー先の変数 (list2) も変更される
017 //コピー先の値を変更すると、コピー元の変数にも影響する (--- 要確認3 ---)
018 list2[2] = 100;        //コピー先の変数の 3番目 の値を 3->100 に変更する
019 console.log("list1: " + list1.toString()); //list1: 1, 99, 100 -> コピー元の変数 (list1) も変更される
020 console.log("list2: " + list2.toString()); //list2: 1, 99, 100
```

|                 |          |
|-----------------|----------|
| a: 2            | VM569:4  |
| b: 2            | VM569:5  |
| a: 2            | VM569:7  |
| b: 999          | VM569:8  |
| list1: 1,99,3   | VM569:12 |
| list2: 1,99,3   | VM569:13 |
| list1: 1,99,100 | VM569:15 |
| list2: 1,99,100 | VM569:16 |

図 A.1 プリミティブ型と参照型の違い

参照型はデータの実体を別の場所に置いておき、変数にはその実体の置かれた場所の情報 (住所のようなもの) を入れます。

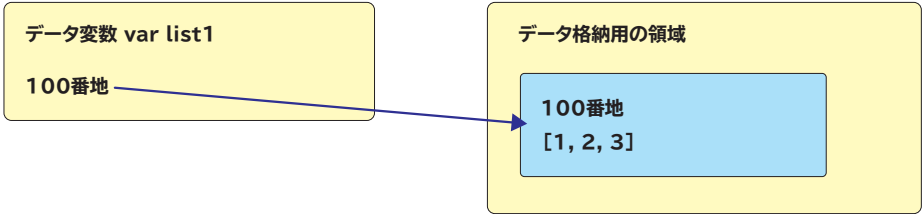


図 A.2 参照型のイメージ

プリミティブ型と参照型の違いの理解を深めるために、[リスト A.2](#) をコピーした場合の動きを比較してみましょう。（--- 要確認 1 ---）のとおり、プリミティブ型は、実際の値そのものを保持しているため、コピー先の値を変更してもコピー元の値には影響を与えません。反対に、（--- 要確認 2 ---）および（--- 要確認 3 ---）のとおり、参照型は値の置かれた場所の情報（住所のようなもの）がコピーされるため、実際の値を変更すると、コピー元、コピー先の両方の値が変更されます。

プログラムの中で値をコピーする場合、この動きを覚えておかないと思わぬバグが発生してしまいます。しっかりおさえて次に進みましょう！

なお、補足として、参照型をコピーする正しい方法を以下に紹介します。気になる人は確認しておいてください。

リスト A.3 配列をコピーする正しい方法①

```
001 var list1 = [1, 2, 3];
002 var list2 = Array.from(list1); //JavaScriptの Arrayに用意されている関数でコピーする
003 list1[1] = 99; //コピー元の変数の2番目の値を「2->99」に変更する
004 console.log("list1: " + list1.toString()); //list1: 1, 99, 3
005 console.log("list2: " + list2.toString()); //list2: 1, 2, 3 -> コピー先の変数（list2）は変更されない
```

|               |
|---------------|
| list1: 1,99,3 |
| list2: 1,2,3  |

図 A.3 配列をコピーする正しい方法①

リスト A.4 配列をコピーする正しい方法②

```
001 var list1 = [1, 2, 3];
002 var list2 = [ ...list1 ]; //ECMAScript6【※本節のコラムで解説】のスプレッド構文でシンプルにコピーする
003 list1[1] = 99; //コピー元の変数の2番目の値を「2->99」に変更する
004 console.log("list1: " + list1.toString()); //list1: 1, 99, 3
005 console.log("list2: " + list2.toString()); //list2: 1, 2, 3 -> コピー先の変数（list2）は変更されない
```

|               |
|---------------|
| list1: 1,99,3 |
| list2: 1,2,3  |

図 A.4 配列をコピーする正しい方法②

Column

const と let について

2015 年 6 月以前の JavaScript には var 宣言しかなかったのですが、ECMAScript（エクマascript）[書籍本文 P.35 のコラム参照](#)（[書籍本文 P.35 のコラム参照](#)）のバージョン 6（ES6 ※以降は ES6 と記載する）で、const および let という宣言が追加されました。

▶ const（定数）

const は定数となり、再代入が不可になります。

説明よりも実際の動作を確認の方が理解しやすいので、Chrome デベロッパーツールに以下のとおりにおのおのを入力し、プリミティブ型と参照型の動きの違いを比べてみましょう。

リスト A.5 const（プリミティブ型の確認）

```
001 const value = 1; //プリミティブ型（値を保持）の定数を定義する
002 value = 2; //NG（再代入できない）（--- 要確認4 ---）
```

```
> const value = 1;
  value = 2; //NG
✖ ▶ Uncaught TypeError: Assignment to constant
  variable.
  at <anonymous>:2:7
```

図 A.5 const（プリミティブ型の確認）

リスト A.6 const（参照型の確認）

```
001 const list = [1, 2, 3]; //参照型（住所を保持）の定数を定義する
002 list[1] = 999; //実際の値の操作は可能（--- 要確認5 ---）
003 console.log(list.toString()); //1, 999, 3
004 list = [1]; //NG（定数への再代入（住所自体の書き換え）はできない）（--- 要確認6 ---）
```

```
1,999,3 VM609:3
✖ ▶ Uncaught TypeError: Assignment to constant
  variable.
  at <anonymous>:4:6
```

図 A.6 const（参照型の確認）

（--- 要確認 4 ---）のとおり、プリミティブ型の定数は実際の値そのものを保持しているため、値の書き換えはできません。反対に、（--- 要確認 5 ---）のとおり、参照型の定数は、値の置かれた場所の情報（住所のようなもの）を保持しているため、実際の値を書き換えることはできますが、（--- 要確認 6 ---）のとおり、住所自体の書き換えはできません。

const の再代入不可の動きを確認することで、プリミティブ型と参照型の違いの理解がより深まったことでしょう。

▶ let（スコープ付き宣言）

let はスコープ（ある変数や関数が特定の名前で参照することができる範囲）付き宣言となり、再宣言が不可になります。

こちらも、説明よりも実際の動作を確認の方が理解しやすいので、Chrome デベロッパーツールに以下のとおりに入力してみましょう。

リスト A.7 let の確認

```
001 //※※説明していない記法が含まれていますが、varとの違いに注目してみてください。※※
002 //中括弧（{}）で囲った範囲は「スコープ」と呼びます
003 if (true) {
004   var val1 = -1;
005   var val1 = 1; //再宣言も可能
006 }
007 console.log(val1); //varはスコープ外でも参照可
008 if (true) {
009   let val2 = 1;
010   //let val2 = 2; <- このような再代入もできない
011 }
012 console.log(val2); //letはスコープ外では参照できない
```

```
console.log(val2); // let はスコープ外では参照できない
1 VM635:5
✖ ▶ Uncaught ReferenceError: val2 is not
  defined
  at <anonymous>:10:13
```

図 A.7 let と var の違い

上述のとおり、自由奔放な var に対し、const、let には、厳格なルールが敷かれています。今後は、const、let を必要に応じて使い分けていくことで、バグの発生率は間違いなく下げられるでしょう。つまり、const、let を使い分けていくことで、厳格で破ることのできないプログラマの意思を変数に込めることができるということです。

以下の [リスト A.8](#) および [リスト A.9](#) を確認してみましょう。

リスト A.8 const の意味

```
001 var tax = 0.08; //消費税を定数として定義
002 tax = 0.10; //コメントにはプログラム制約はないため、定数として定義したものを書き換えられてしまう
003 console.log(tax);
004 const tax2 = 0.08;
005 tax2 = 0.10; //コメントを付与する必要もなく、厳格なルールを付与できる -> 代入エラーが発生
```

0.1

✖ ▶ Uncaught TypeError: Assignment to constant variable.  
at <anonymous>:5:6

図 A.8 const の意味

リスト A.9 let の意味

```
001 if (true) {
002   var val1 = -1;   //スコープ内変数、スコープ外では利用しないでください
003 }
004 console.log(val1); //コメントにはプログラム制約はないため、スコープ外でも変数を参照できる
005 if (true) {
006   let val2 = 1;
007 }
008 console.log(val2); //コメントを付与する必要もなく、厳格なルールを付与できる -> 参照エラーが発生
```

-1

✖ ▶ Uncaught ReferenceError: val2 is not defined  
at <anonymous>:8:13

図 A.9 let の意味

なお本書では、説明のしやすさを重視し、変数の宣言には基本 var を用いますが、プロダクトコードを記述する場合は、const、let を積極的に使っていきましょう！

Column

トランスパイルとコンパイルとは

トランスパイラが行う変換をトランスパイル（transpile）といいます。トランスパイルとは、あるプログラミング言語で書かれたプログラムのソースコードを、別のプログラミング言語と同等のコードに翻訳することです。

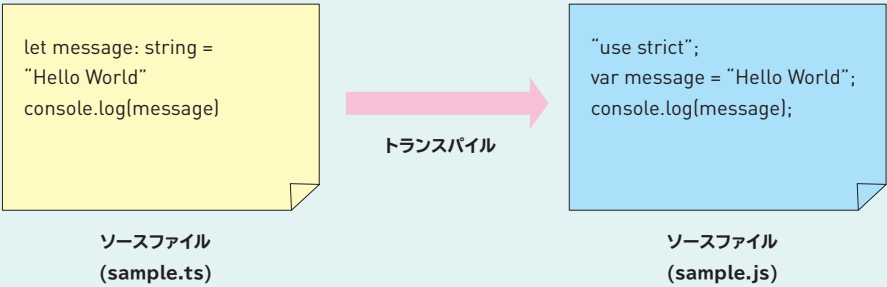


図 A.10 トランスパイルのイメージ

なお、C 言語などのコンパイル型言語では、コンパイルを行ってプログラムのソースコードをコンピュータが扱えるよう、実行可能な形式（0 と 1 の羅列（ビット列）。バイナリ形式とも言います）に変換します。

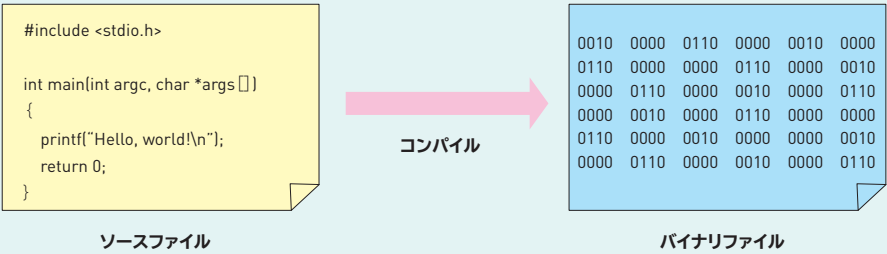


図 A.11 コンパイルのイメージ

Note | ポリフィルとは

JavaScript は、新しい仕様が追加された際に、ブラウザによって対応状況が大きく変わってきます。ポリフィルとは、利用したい機能に対応していないブラウザでも使えるように、同等の機能を JavaScript で自動で置き換えることです。

例として、「Fetch API」は、Internet Explorer（IE）では利用できませんが、ポリフィル「window.

fetch polyfill」を利用することで、Chrome などの対応ブラウザと同等の機能を利用できるようになります。

なお、使い方のサンプルは「window.fetch polyfill」に掲載されていますので、気になる方はぜひ参照してみてください。

Fetch API

[https://developer.mozilla.org/ja/docs/Web/API/Fetch\\_API](https://developer.mozilla.org/ja/docs/Web/API/Fetch_API)

window.fetch polyfill

<https://github.com/github/fetch>

undefined と null についての動作を確認するために、Chrome デベロッパーツールに以下のとおりに入力してみましょう。

リスト A.10 undefined と null について

```
001 var hoge;           //変数のみ定義した状態
002 var fuga = null;    //明示的に値がない状態を設定した状態
003 console.log(hoge);  //undefined
004 console.log(fuga);  //null
005 //条件式：カッコ内が真のとき、プログラムが実行される
006 if (hoge === fuga) console.log("hogeとfugaは同一です(1)"); //この式は評価されない
007 hoge = null;
008 if (hoge === fuga) console.log("hogeとfugaは同一です(2)。");
```

|                    |          |
|--------------------|----------|
| undefined          | VM1447:3 |
| null               | VM1447:4 |
| hogeとfugaは同一です(2)。 | VM1447:8 |
| ← undefined        |          |

図 A.12 プリミティブ型 (null) の確認

var hoge と var fuga = null のどちらにプログラムの意図を感じるでしょうか。= null がついた var fuga = null の方が、意図を持って「値がない」ことを表しているとは思いませんか？自分が書いたプログラムでも、数ヶ月後に見返すと忘れてしまうものです。そのため「値がない」ことを設定したい場合は、「null」を使うようにしましょう。

undefined と null の動作をもう少し確認するために、Chrome デベロッパーツールに以下のとおりに入力してみましょう。

リスト A.11 undefined と null の違いについて

```
001 var hoge;           //変数のみ定義した状態
002 var fuga = null;    //明示的に値がない状態を設定した状態
003 //条件式：カッコ内が真のとき、プログラムが実行される
004 //未設定の変数を判定するためによく用いられる条件式
005 if (hoge) console.log("undefinedも"); //この条件式も正とならない = undefined と null が同じ扱いになる
006 if (fuga) console.log("nullも");      //この条件式も正とならない = undefined と null が同じ扱いになる
007 console.log("実行されない");
008 if (hoge === null) console.log("意図を持ってnullを設定した場合のみ実行される case undefined");
009 if (fuga === null) console.log("意図を持ってnullを設定した場合のみ実行される case null");
```

|                                    |
|------------------------------------|
| 実行されない                             |
| 意図を持ってnullを設定した場合のみ実行される case null |

図 A.13 undefined と null の違いについて

JavaScript の if( 変数名 ) という条件式 (5、6 行目) では、undefined (未定義) も null も同じ扱いとなります。変数の宣言 (2 行目) も、条件式の判定 (8、9 行目) も可能な限りわかりやすく具体的にプログラムに記載しておきましょう ( 書籍本文 Chapter 2-4 )。たとえば、税関のチェックを少しでも緩くすると、意図しないものが入ってきてしまい危険ですよね。

▶ データ型の変換

JavaScript は変数宣言の際、数値、文字列などの型を自動的に変換 (動的型付け言語) します。そのため、変数宣言の際にデータ型を指定する必要がなく、一度定義した変数に異なるデータ型を代入することもできます (リスト A.12)。

では実際に、Chrome デベロッパーツールに以下のとおりに入力して、具体的な動作を確認してみましょう。

リスト A.12 データ型の変換

```
001 var result = 42;      //変数に数字を代入
002 console.log(result);  //数字が表示される
003 result = "回答は42です"; //数字を入れた変数 (箱) に文字も代入できる
004 console.log(result);  //文字も表示される
```



|             |          |
|-------------|----------|
| 42          | VM1361:2 |
| 回答は42です     | VM1361:4 |
| ← undefined |          |

図 A.14 異なるデータ型の代入

Column

動的型付け言語と静的型付け言語の違いとは？

JavaScript は前述のとおり動的型付け言語のため、[リスト A.12](#) のとおり、数字を代入した変数に「文字」を再代入できます。

一方、静的型付け言語ではどうでしょうか。静的型付け言語である Java の例を見てみましょう。

リスト A.13 静的型付け言語の Java (jshell) の例

|  |
|--|
| \$ jshell  |
| JShellへようこそ -- バージョン11.0.1   |
| 概要については、次を入力してください: /help intro                                      |
|  |
| jshell> Integer number = 42; //変数に数字を代入、変数のデータ型 (Integer) も定義する必要がある |
| number ==> 42  |
|  |
| jshell> System.out.println(number); //数字が表示される                       |
| 42   |
|  |
| jshell> number = "回答は42です"; //数字を入れた変数に文字を代入するとエラーになる                |
| エラー:   |
| 不適合な型: java.lang.Stringをjava.lang.Integerに変換できません:                   |
| number = "回答は42です";  |
| ^-----^  |

動的型付け言語は、本来実現したいことのみがコードとして残る（変数の宣言が簡略化される、など）ため、記述量が減ります。静的型付け言語は、コンパイル（前出のコラム参照）時にエラーが発生するため、プログラム実行前にバグやエラー箇所を特定できます。

ここでは、動的型付け言語、静的型付け言語のいずれの言語にも特徴があり、JavaScript は動的型付け言語であるということを紹介しました。

3 リテラルとは？

JavaScript では、データ型の値の代入に「リテラル」を使います。

リテラルとは、データ型に格納できる値そのもの、または、値の表現方法のことです。文章で説明するとわかりにくいので、Chrome デベロッパーツールに以下のとおりに入力して、実際の動きを確認してみましょう。

リスト A.14 リテラルとは

|     |   |
|-----|---|
| 001 | var hello = "Hello, Vue.js"; // "Hello, Vue.js" がリテラル（文字列）です。 |
| 002 | //文字列を表現する際は、シングルクォート、ダブルクォートで文字を囲みます。                        |
| 003 | console.log(hello);   |
| 004 |   |
| 005 | var number = 11; //11 がリテラル（数値）です。                            |
| 006 | console.log(number);  |
| 007 | var pi = 3.14e-3; //3.14e-3 がリテラルです。                          |
| 008 | //数値リテラルは指数表記 3.14e-3 (= 0.00314) でも表現できます。                   |
| 009 | console.log(pi); //0.00314                                    |

|               |
|---------------|
| Hello, Vue.js |
| 11            |
| 0.00314       |

図 A.15 リテラルとは

変数 hello に格納している値そのものが "Hello, Vue.js" で、"Hello, Vue.js" がリテラル（＝文字列リテラル）です。

リテラル（＝数値リテラル）の場合は、11 のように整数のみ、また、3.14e-3 のように指数表記でも表現できます。

この節は、JavaScript で扱うことができる、以下のリテラルについて説明します。

- 真偽値リテラル
- 数値リテラル（浮動小数点リテラル）
- 文字列リテラル
- 配列リテラル
- オブジェクトリテラル
- 関数リテラル

▶ 真偽値リテラルとは

真偽値型は 2 つのリテラル値、true または false があります。

▶ 数値リテラル（浮動小数点リテラル）とは

整数は 10 進数、16 進数、8 進数、2 進数で表現することができます。

数値リテラルの表現方法のルールは、以下のとおりです。

117、-345      ……数値リテラル (10)  
3.1415926      ……浮動小数点リテラル  
3.14e-3        ……0.00314、指数表記でも表現できる

▶ 文字列リテラルとは

文字列リテラルとは、テキストの値を表す連続した文字を、シングルクォーテーションもしくはダブルクォーテーションで括ったものです。

文字列の動作を確認するために、Chrome デベロッパーツールに以下のとおりに入力してみましょう。

リスト A.15 文字列リテラルの動作

```
001 var str = "Hello! JavaScript";  
002 console.log(str);           //文字列をそのまま表示  
003 console.log(str.length);    //文字列のプロパティ（長さ）を表示  
004 console.log(str.indexOf("JavaScript")); //文字列から「JavaScript」文字列が存在する位置を検索（文字列に対する操作）
```

|                   |          |
|-------------------|----------|
| Hello! JavaScript | VM1343:2 |
| 17                | VM1343:3 |
| 7                 | VM1343:4 |
| ← undefined       |          |

図 A.16 文字列リテラルの動作

プリミティブ型はメソッドを持たないと説明しましたが、例文の 4 行目はメソッドを呼び出しています。これは JavaScript の仕様で、文字列リテラルが代入された変数は、String オブジェ

クトのあらゆるメソッドを呼び出すことができるためです。

また、[リスト A.15](#) のとおり、String.length プロパティを文字列リテラルで用いることもできます。

▶ 配列リテラルとは

配列リテラルは、カンマで区切った値をブラケット（[]）で囲った形式で表し、それぞれの要素にはインデックス番号（0 から始まる数字）をキーにアクセスします。

配列の動作を確認するために、Chrome デベロッパーツールに以下のとおりに入力してみましょう。

リスト A.16 配列リテラルの動作

```
001 var genre = ["pop", "rock", "edm"];  
002 console.log(genre[2]);        //edm  
003 console.log(genre[5]);        //undefined  
004 console.log(genre.length);    //3 -> 配列の長さを表示  
005 console.log(genre.toString()); //pop,rock,edm -> 文字列リテラル同様、メソッド呼び出し可能  
006 genre.forEach(value => console.log(value)); //配列の一覧を表示（アロー関数を利用。※後述、本Chapterの5項）  
007 //pop  
008 //rock  
009 //edm  
010 console.log(genre.includes("edm")); //true -> 配列に値が含まれているか確認  
011 console.log(genre.includes("punk")); //false -> 配列に値が含まれているか確認
```

|              |        |
|--------------|--------|
| edm          | VM58:2 |
| undefined    | VM58:3 |
| 3            | VM58:4 |
| pop,rock,edm | VM58:5 |
| pop          | VM58:6 |
| rock         | VM58:6 |
| edm          | VM58:6 |
| true         | VM58:7 |
| false        | VM58:8 |

図 A.17 配列リテラルの動作



Column

配列のメリット

配列のメリットを、プリンタの待ち行列で考えてみましょう。1枚あたりの印刷時間が1秒かかるとして、太郎さんがいつ印刷をはじめられるか計算したいとします。

この場合は、印刷の終わった人は待ち行列の先頭から削除され（※図では「花子」さん。JavaScriptではArray.shift()で削除できます）、印刷したい人は待ち行列の最後に追加します（※図では「太郎」さん。JavaScriptではArray.push()で追加できます）。

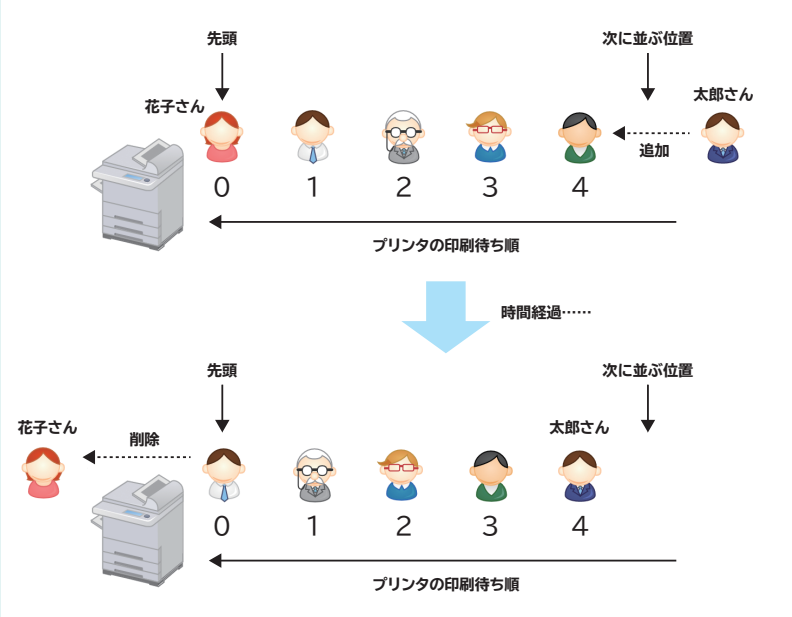


図 A.18 配列イメージ

では、「配列を使わない場合」と「配列を使う場合」、それぞれの場合を確認してみましょう。

リスト A.17 配列を使わない場合

```
001 //----- 「配列を使わない場合」の例
002 //1人1人の変数を愚直に定義する必要がある
003 //{name: '花子', number: 15} はオブジェクトリテラル（※本節の次の目参照）の表現方法
004 var people1 = {name: '花子', number: 15};
005 var people2 = {name: '一郎', number: 30};
006 var people3 = {name: '二郎', number: 45};
007 var people4 = {name: '三郎', number: 60};
008 var people5 = {name: '四郎', number: 90};
009 var people6 = {name: '太郎', number: 15};
010
011 //1人1人のコピー枚数をタイプエラーに気をつけながら、集計する必要がある
012 //※集計に用いる算術演算子は、書籍本文 P.15 で説明
013 var wait = (people1.number + people2.number + people3.number +
014             people4.number + people5.number);
014 console.log(`待ち時間は、${wait}秒です`);
015 console.log(`先頭から2人目は、${people2.name}です`); // 「一郎」を直接指定して取得する
```

待ち時間は、240秒です  
先頭から2人目は、一郎です

図 A.19 配列を使わない場合

リスト A.18 配列を使う場合

```
001 //----- 「配列を使う場合」の例
002 //変数を集合で扱うことができ、変数は1つで済む
003 var peopleList = [ {name: '花子', number: 15}, {name: '一郎', number: 30}, {name: '二郎', number: 45},
004                   {name: '三郎', number: 60}, {name: '四郎', number: 90}, {name: '太郎', number: 15} ];
005
006 //配列を利用すると、便利な関数が利用できる
007 //->以下処理順で集計する。1000人いても同様のロジックで対応できる
008 // 1. 配列をコピー。[...peopleList] に相当
009 // 2. 太郎さん自身を除く。slice(0,5) に相当
010 // 3. 残った人のコピー枚数を集計。reduce((a,p) => a+=p.number,0) に相当
011 var wait = [...peopleList].slice(0,5).reduce((a,p) => a+=p.number,0);
012 //アロー関数（書籍本文 P.30 で解説）を利用
013 console.log(`待ち時間は、${wait}秒です`);
014 console.log(`先頭から2人目は、${peopleList[1].name}です`);
015 //インデックス番号でアクセスできる
```

|               |          |
|---------------|----------|
| 待ち時間は、240秒です  | VM371:10 |
| 先頭から2人目は、一郎です | VM371:11 |

図 A.20 配列を使う場合

ご覧いただいたように、配列を用いない場合は変数を 1 つ 1 つ定義する必要があり、計算などをする場合も変数を 1 つ 1 つ記述する必要があります。

一方、配列を用いると、便利な関数が利用でき、100 人、さらには 1000 人並んでいても同様の記述方法で対応できます。また、先頭から n 人目を取得する場合も、配列を用いるとわかりやすい順序（※インデックス番号は 0 からはじまるため、-1 する必要はあります）でダイナミックに要素を特定できます。

▶ オブジェクトリテラル

オブジェクトは、複数の値をまとめたものです。

配列と似ていますが、配列はインデックス番号で値を順番に管理しているのに対し、オブジェクトは値に対する順番がありません。インデックス番号を使わずに任意の名前（識別子）で値にアクセスできるため、配列に比べ扱いやすいです。

それでは、オブジェクトの動作を確認するために、Chrome デベロッパーツールに以下のとおりに入力してみましょう。

リスト A.19 オブジェクトリテラルの動作

|     |   |
|-----|---|
| 001 | var user = { sei: "hirosue", mei: "takeshi", address: "tokyo-to" };                                 |
| 002 | console.log(user.sei); //hirosue  |
| 003 | console.log(user.mei + " " + user.sei); //takeshi hirosue   |
| 004 | console.log(user["sei"]); //hirosue -> ブラケット ([]) でのアクセスも可能   |
| 005 | console.log(user.tel); //undefined  |
| 006 | console.log(user.hasOwnProperty("sei")); //true -> オブジェクトに識別子が含まれているか確認                             |
| 007 | console.log(user.hasOwnProperty("fax")); //false -> オブジェクトに識別子が含まれているか確認                            |
| 008 | console.log(Object.getOwnPropertyNames(user)); //["sei", "mei", "address"]<br>-> オブジェクトのプロパティー一覧を取得 |
| 009 | Object.getOwnPropertyNames(user).forEach(key =>   |
| 010 | console.log(key + ":" + user[key])  |
| 011 | ); //オブジェクト識別子と値の一覧を表示（アロー関数。書籍本文 P.30 で解説）   |
| 012 | //sei:hirosue   |
| 013 | //mei:takeshi   |
| 014 | //address:tokyo-to  |

|                                 |         |
|---------------------------------|---------|
| hirosue                         | VM216:2 |
| takeshi hirosue                 | VM216:3 |
| hirosue                         | VM216:4 |
| undefined                       | VM216:5 |
| true                            | VM216:6 |
| false                           | VM216:7 |
| ▶ (3) ["sei", "mei", "address"] | VM216:8 |
| sei:hirosue                     | VM216:9 |
| mei:takeshi                     | VM216:9 |
| address:tokyo-to                | VM216:9 |

図 A.21 オブジェクトリテラルの動作

▶ 関数リテラルとは

JavaScript では関数もリテラルとなり、変数に代入できます。

関数リテラルの動作確認するために、Chrome デベロッパーツールに以下のとおりに入力してみましょう。

リスト A.20 関数リテラルの動作

|     |   |
|-----|---|
| 001 | var plus = function(a, b) {                     |
| 002 | return a + b;                                   |
| 003 | };  |
| 004 | console.log(plus(1, 1)); //2                    |
| 005 | var minus = (a, b) => a - b; //アロー関数でシンプルに記載できる |
| 006 | console.log(minus(2, 1)); //1                   |

|   |         |
|---|---------|
| 2 | VM114:2 |
| 1 | VM114:4 |

図 A.22 関数リテラルの動作

なお、関数は 書籍本文 Chapter 2-4 で詳細を解説します。