04 TypeScriptの基本ルール

Keyword ☑命令文 ☑コメント

TypeScriptのコードはどこから始まってどこで終わる？

TypeScriptファイルには、コンピュータに実行してほしい命令を複数行にわたって記述をします。

例えば、**リスト1-1**は3＋2を計算して、その結果を表示するコードです。人間と同じように上から順番に、1行ずつ読み込みながら実行をします。

▼リスト1-1　TypeScriptのサンプルコード

```
01: let x : number = 0;
02: x = 3 + 2;
03: console.log(x);
```

命令文の書き方の基本ルールを覚えよう

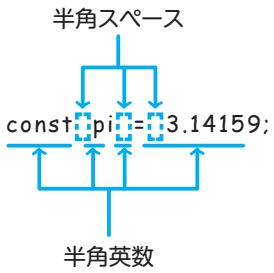
TypeScriptのソースコードを書く上で、最低限覚えておきたいルールについて説明します。

● 半角英数字を使用しよう

基本的に、ソースコードは半角英数字で記述します。スペースも全角文字は使用せず、半角のスペースを使用します（**図1-23**）。

全角文字を使用するのは、変数（後述します）に代入する文字列や、コメント（後述します）として記述する場合です。

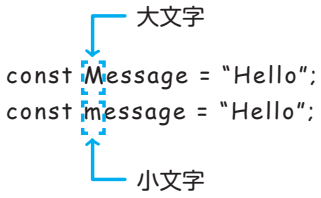
▼図1-23　半角英数字とスペースの例



● 大文字と小文字を区別しよう

TypeScriptは、大文字と小文字は別なものとして扱われます。例えば、Messageとmessageのスペルは同じですが、先頭の1文字目が大文字と小文字という違いがあります。これらは、全くの別ものとして扱われるということを覚えておきましょう（**図1-24**）。

▼図1-24　大文字と小文字の例



● 文の終わりをには「;」を付けよう

TypeScriptは、1つの命令文がどこで終わるかを自動的に判断をします。命令文は複数行に渡って書くことができますので、どこまでが命令文なのかを示すために「;（セミコロン）」を記述して読みやすくすることができます。

「;」は省略しても構いませんが、本書では1つの命令文の終わりには「;」を記述して説明をします。

04 型推論

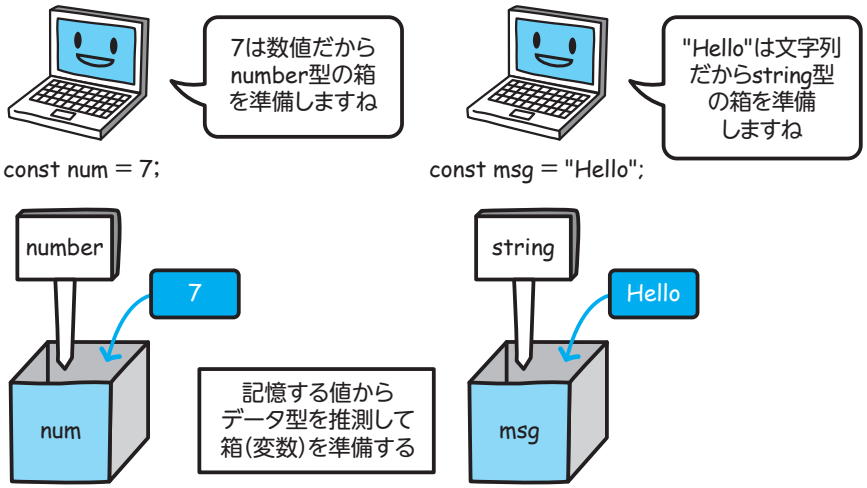
Keyword ☑ 型推論

型推論ってなんだろう？

これまでに学んだ変数の宣言方法は、データ型を指定するものでした。

TypeScript にはもう 1 つ、**構文 2-4** に示す「型推論（かたすいろん）」による変数の宣言方法があります。

▼図 2-15　型推論のイメージ



これまでの宣言方法とは異なり、データ型を指定していません。初期値を代入すると、そのデータから自動的にデータ型が決まります。このように、データからデータ型を推測して決定することから型推論と呼ばれます（図 2-15）。

構　文　**2-4　型推論**

```
const 変数名 = 初期値;
```

リスト 2-29 に型推論の例を示します。

▼リスト 2-29　型推論の例

```
01: const num = 7;
02: const msg = "Hello";
03:
04: console.log(num);            // 7 を出力
05: console.log(typeof(num));    // "number" を出力
06: console.log(msg);            // "Hello" を出力
07: console.log(typeof(msg));    // "string" を出力
```

1 行目は変数 num に 7 を代入しています。7 は整数ですので型推論されて number 型になります。

2 行目は、変数 msg に "Hello" を代入しています。"Hello" は文字列ですので、型推論されて string 型になります。

4 行目は変数 num に記憶されている 7 が出力されます。

5 行目で使用している「typeof(変数名)」は、その変数のデータ型を調べるための命令です。変数 num は数値の 7 が代入されていますので、データ型の "number" が出力されます。

6 行目は msg に記憶されている "Hello" が出力されます。

7 行目は変数 msg のデータ型を確認していますので、"string" が出力されます。

02 配列の操作

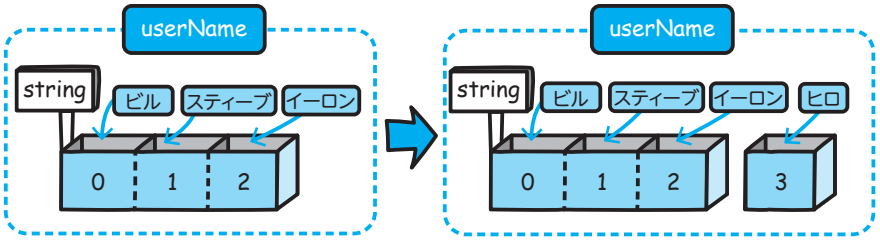
Keyword ☑非破壊的 ☑破壊的 ☑メソッド ☑push ☑slice

破壊的操作と非破壊的操作

「破壊的」と「非破壊的」という単語が出てきました。普通に考えると「破壊」という単語は、壊してダメにしてしまいそうなイメージがありますが、ここでいう「破壊的」というのは、元のデータが変わってしまうことを指します。

それでは「破壊的操作」を**図 3-6** で確認しましょう。

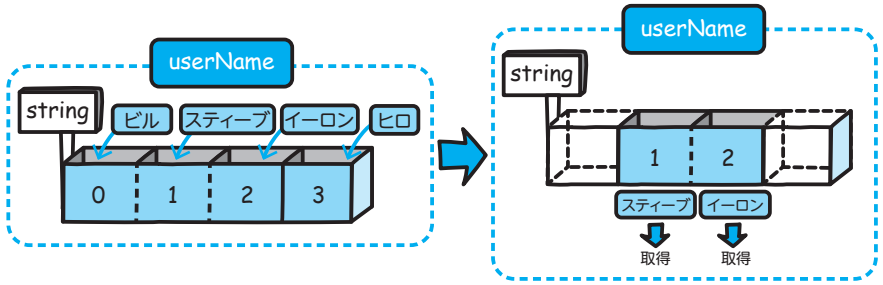
▼図 3-6　破壊的操作のイメージ



userName という配列を宣言して「ビル」「スティーブ」「イーロン」で初期化した場合は、3 つの値を持つ配列ができますね。この userName 配列に新しく仕切りを追加すると「3 つの値を持つ配列」から「4 つの値を持つ配列」に変わりますね。「ビル」「スティーブ」「イーロン」という 3 つのデータ自体は変わっていませんが、データを入れられる数が増えました。すなわち元の状態が変更されたことになりますので、これは「破壊的操作」となります。

次に**図 3-7** で「非破壊的操作」のイメージをみてみましょう。

▼図 3-7　非破壊的操作のイメージ



配列 userName には「ビル」「スティーブ」「イーロン」「ヒロ」という文字列が格納されています。続いて、この配列からインデックスが 1 と 2 の要素だけを取り出して表示する場合を考えてみましょう。取り出すとは言っても、配列の中身を参照（見に行く）するだけですので、配列そのものを変更することはありませんね。このように、元のデータに変更を加えないような操作が「非破壊的操作」となります。

次節では、実際に配列にデータを追加するプログラムと、中身を参照するプログラムの作成方法について学習します。

配列にデータを追加してみよう

配列にデータを追加するには、いくつかの方法があるのですが、特に使用頻度が高いのが、push という機能を利用する方法です。

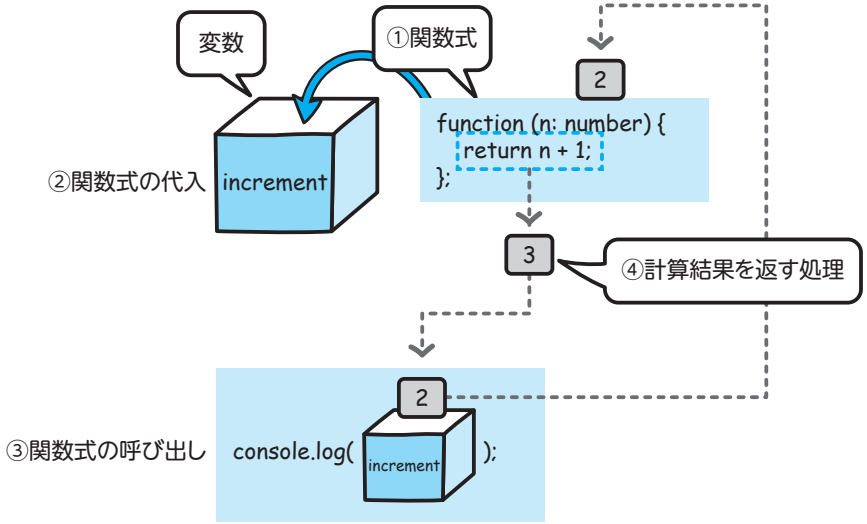
メモ

TypeScript では、「機能」のことをメソッドといいます。「メソッド」は「第 8 章　クラスの基本」で詳しく学習をします。今は「機能のことをメソッドというのだな」という理解で構いません。

▼リスト 7-8　関数式の例

```
01: // インクリメントを行う関数式
02: const increment = function (n: number) {
03:     return n + 1;
04: };
05:
06: console.log(increment(2));   // 3を出力
```

▼図 7-6　リスト 7-8 の実行イメージ



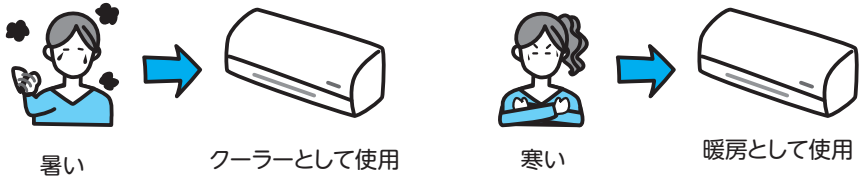
関数に関数式を渡してみよう

関数式は変数に代入できるので、関数の引数として渡すことも可能です。関数の引数として渡される関数は**コールバック関数**といいますので覚えておきましょう。

それでは、関数に関数式を渡せるようになるとどのようなメリットがあるのかをエアコンを例に考えてみましょう（**図 7-7**）。エアコンは、空気の温度調節をするという機能を持ちますが、暑ければクーラーとして、寒ければ暖房として機能しますよね。このようにその時々に応じて用途を変更することができます。

関数式を引数に渡せることは、エアコンと同様に、場面に合わせて機能を変更できる関数を作成できることを意味します。

▼図 7-7　エアコンの機能



それでは関数に関数式を渡すプログラムを作成してみましょう（**リスト 7-9**）。

この例では applyOperation という関数を定義して、引数で関数式を受け取って実行します。関数式は 2 つ定義しており、引数で受け取った値に 1 を加算する increment と、受け取った値から 1 を減算する decrement 関数式を定義しています。

▼リスト 7-9　関数に関数式を渡して利用する例

```
01: // 引数に関数を取る関数
02: function applyOperation(n : number, action :
    Function) : void {
03:     console.log(action(n));
04: }
05:
06: // インクリメントを行う関数式
07: const increment = function (n: number) {
08:     return n + 1;
09: };
10:
11: // デクリメントを行う関数式
12: const decrement = function (n: number) {
13:     return n - 1;
```

次へ

03 メソッド

Keyword ☑ メソッド

メソッドを定義してみよう

クラスで定義される関数を「メソッド」と呼びます。メソッドは、モノが行う動作や操作を表すために使用し、**構文 8-6** を使用して定義します。

function キーワードがないことを除けば、関数の定義と同じであることがわかります。

構　文　**8-6　メソッドの定義**

```
メソッド名 ( 引数 1: 型 1, 引数 2: 型 2, ...): 戻り値の型 {
  // 処理
  return 戻り値 ;
}
```

リスト 8-1 で作成した House クラスにメソッドを追加してみましょう。**リスト 8-7** にセキュリティの作動と停止をするメソッドを定義する例を示します。

メソッド名は activateSecurity としました。また isOn という boolean 型の引数を持たせています。isOn はセキュリティのスイッチに見立てていて、true を受け取ると「セキュリティを作動しました」を出力し、false を受け取ると「セキュリティを停止しました」を出力します。

▼リスト 8-7　メソッドの定義

```
01: class House {
02:     activateSecurity(isOn: boolean) {
03:         if (isOn) {
04:             console.log("セキュリティを作動しました");
05:         } else {
06:             console.log("セキュリティを停止しました");
07:         }
08:     }
09: }
```

メソッドを使ってみよう

メソッドは、**構文 8-7** のように呼び出します。インスタンス化した後に「インスタンス名 . メソッド (値)」という書式で、メソッドを使用することができます。

構　文　**8-7　メソッドの使用**

```
インスタンス名 . メソッド ( 値 );
```

それでは、実際にメソッドを使用するプログラムを作成してみましょう。**リスト 8-8** は、House クラスに定義した activateSecurity メソッドを使用する例です。

はじめに、House クラスから redHouse というインスタンスを作成しています。

メソッド activateSecurity は「redHouse. activateSecurity(値)」のようにして使用します。使い方は、プロパティやセッター、ゲッターと同様です。「.」は日本語の「の」と読み替えて「赤い家 " の " セキュリティ作動」と訳すと理解が進みます。

04 ジェネリックメソッド

Keyword ☑ ジェネリックメソッド

ジェネリックメソッドってなんだろう？

TypeScript はデータ型を持つ言語であり、型情報を活用することで安全性や柔軟性を向上することができます。その中でもジェネリックメソッドは強力な機能の一つです。すでに学習したメソッドは、そのメソッド内で使用するデータの型はあらかじめ決めていました。一方でジェネリックメソッドは、あえてメソッド内で使用するデータの型を指定しないことで、汎用的に処理を行わせることができます。なお、本節では、ジェネリックメソッドについて説明をしますが、ジェネリック関数も同様の考え方で定義をすることができます。

ジェネリックメソッドを使ってみよう

例えば、数値配列を引数で受け取ってすべての値を表示する printNumberArray メソッドと、文字配列を引数として受けとってすべての値を表示する printStringArray メソッド定義する場合、**リスト 9-7** のように書くことができます。

▼リスト 9-7　数値を表示するメソッドと文字列を表示するメソッドの例

```
01: class MyUtility {
02:     printNumberArray(arr: number[]): void {
03:         for (const val of arr) {
04:             console.log(val);
05:         }
06:     }
07:
08:     printStringArray(arr: string[]): void {
09:         for (const val of arr) {
10:             console.log(val);
11:         }
12:     }
13: }
```

printNumberArray メソッドと printStringArray メソッドを比較すると、内部のコードが完全に同じであることがわかります。引数のデータ型が異なるだけであり、冗長な状態になっていますね。

そこで、ジェネリックメソッドを使用することで、これらのメソッドを統合することができます。ジェネリックメソッドは**構文 9-4** を使用して定義します。

構　文　9-4　ジェネリックメソッド

```
メソッド名 <T>( 引数 : T): T {
  // メソッドの機能
}
```

ちなみにジェネリック関数の場合は**構文 9-5** を使用します。

構　文　9-5　ジェネリック関数

```
function 関数名 <T>( 引数 : T): T {
  // 関数の機能
}
```

01 抽象クラス

Keyword ☑抽象クラス ☑抽象メソッド ☑abstract

抽象クラスってなんだろう？

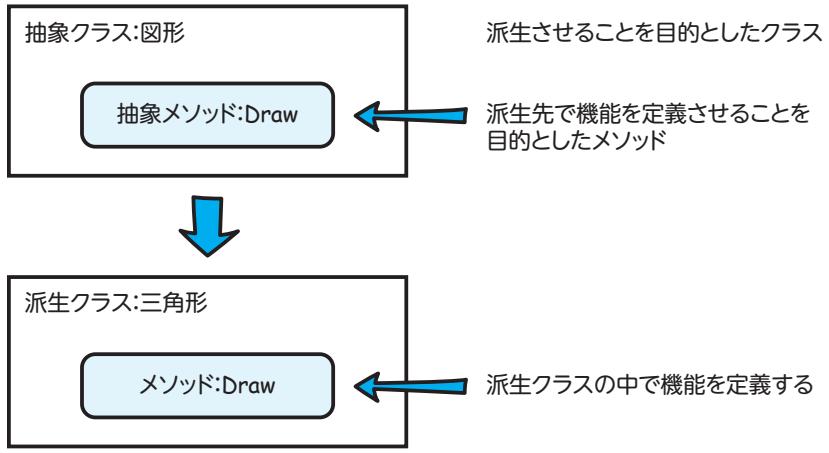
第 9 章で学んだオーバーロードは、基本クラスのメソッドを派生クラスのメソッドで上書きをするものでしたね。しかし、上書きすることを目的とするメソッドを定義したい場合は、基本クラスのメソッドに機能を実装する必要はありません。このようなシチュエーションに対応するために、**抽象クラス**と**抽象メソッド**があります。

抽象クラスは、複数のクラスに共通するメソッドを定義するために使用されます。また、抽象メソッドと呼ばれる、中身のない（機能を持たない）メソッドを定義することができます。抽象メソッドを含む抽象クラスを継承した派生クラスは、必ずその抽象メソッドを実装する必要があります。

例えば、図形クラスを抽象クラスとして定義し、抽象メソッド「描画する」を定義すると、三角形クラスでは必ずその抽象メソッドを実装し、図形を描画する機能を実装する必要があります（**図 10-1**）。

このように、抽象クラスは、共通の性質を持つクラスをまとめ、メソッドのオーバーライドを強制することで、ソフトウェア設計の柔軟性や効率性を向上させることができます。

▼図 10-1　抽象クラスのイメージ



抽象クラスを定義しよう

抽象クラスは**構文 10-1** を使用して定義をします。

抽象クラスは abstract キーワードの後ろにクラス名を記述し、抽象メソッドは abstract キーワードの後ろに、メソッド名、引数、戻り値のみを記述し、メソッドの中身は記述しません。

抽象クラスには、抽象メソッド以外にコンストラクタやプロパティ、中身を実装した通常のメソッドも記述することもできます。

構　文　10-1　抽象クラス

```
abstract クラス名 {
    abstract メソッド名 ( 引数リスト ): データ型 ;
}
```

それでは実際に抽象クラスを定義して、派生クラスに継承させてみましょう（**リスト 10-1**）。

02 try catch finally

Keyword ☑ try ☑ catch ☑ finally ☑ throw ☑ Error

throw による例外の発生

プログラミングにおいて例外処理をすることは重要な要素です。

throw 文を使用すると、意図的に例外を発生させることができ、後に説明する try ～ catch ～ finally 構文を使用することで、効率的に例外処理を行うことができます。

throw 文は**構文 11-1** を使用し、throw の後ろにはエラーオブジェクトのインスタンスを置きます。また、エラーオブジェクトのインスタンスを生成するには**構文 11-2** を使用します。ちなみに、例外を発生させることを「例外を投げる」または「例外をスローする」といいます。

構　文　11-1　throw 文

```
throw エラーオブジェクト;
```

構　文　11-2　エラーオブジェクト

```
new Error( エラーメッセージ )
```

リスト 11-1 は、x ÷ y を計算するプログラムです。しかし、数学的には 0 で割り算をすることはできないため、計算を行う前に y が 0 かどうかを確認し、0 の場合には throw を使用して例外を発生させています。この例では「ゼロで割り算はできません」というエラーメッセージを表示します。

▼リスト 11-1　例外を発生させる例

```
01: const x: number = 5;
02: const y: number = 0;
03:
04: if (y === 0) {
05:     // yが0のときは例外をスローしてプログラムを終了
06:     throw new Error("ゼロで割り算はできません。");
07: }
08:
09: const ans = x / y;
```


try catch finally ってなんだろう？

プログラム中で発生する例外は if 文で回避できる場合があることはすでに説明したとおりです。

しかし、単に if 文を使うだけでは、すべての例外を回避できるとは限りません。特定の範囲内で何かしらのエラーが発生する可能性もあります。

このような場合に備え、TypeScript には例外を専用に処理する try ～ catch ～ finally という構文があります（**図 11-3**）。「try」「catch」「finally」はそれぞれのブロックに分かれています。

「try」は例外が発生する可能性のあるコードを記述するブロックです。try ブロックの中で例外が発生した場合は後続の処理を実行せずに catch ブロック内の先頭のコードに移動します。try ブロック内のコードをすべて正常に実行できた場合は catch ブロックには移動せず、finally ブロックへ移動します。

「catch」は try ブロックで発生した例外の種類を特定し、その例外を適切

01 モジュールとは

Keyword ☑モジュール ☑export ☑import

モジュールってなんだろう？

プログラミングにおいて、ソフトウェアを構成する要素を個々の独立した部品に分割し、それらを組み合わせて開発することは非常に重要です。モジュールは、この部品化の概念を実現するための仕組みです。

モジュールは、関連するクラスや関数、定数などをグループ化し、他のコードから独立して扱うことができる単位です。つまり、プログラム全体を 1 つの大きなファイルに書くのではなく、モジュール単位の小さなファイルに分割して管理することができます。

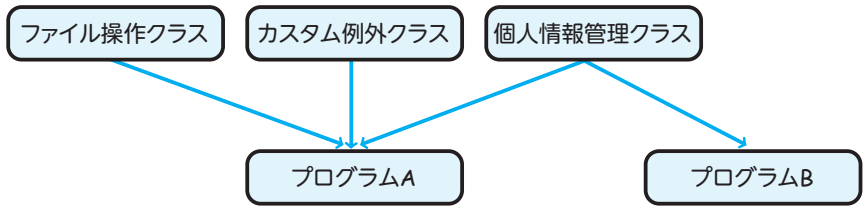
例えば、あるファイルに書かれている個人情報を読み取って、特定の地域に住む年代別の人口を求めるプログラムを作成するとしましょう。このプログラムの作成には、「ファイル操作クラス」「カスタム例外クラス」「個人情報管理クラス」が必要で、これらがすべて AppDefinition.ts という 1 つのファイル内に定義されているとします。

新しいプログラムを作成するときに、この「個人情報管理クラス」を再利用したいと考えた場合は、AppDefinition.ts から「個人情報管理クラス」を切り出して、新しいファイルに再定義し直す必要がありますね。

モジュールを使用すると、それぞれの要素を別々のファイル（モジュール）に分割することができます。これにより、「プログラム A」を作成するだけでなく、別の「プログラム B」でも再利用することが可能です（**図 12-1**）。

また、モジュールに分割することで、個々のプログラムが短くなり、可読性が向上します。そのため、プログラムの改修が必要な場合でも、特定の範囲に絞って修正を行うことができ、保守性も向上します。

▼図 12-1　モジュールのイメージ



エクスポートとインポート

モジュールを使用するには、複数のファイル間で変数や関数を共有できるようにする必要があります。TypeScript では、変数や関数などを他のファイルから再利用できるようにするために、エクスポート（export）という機能が提供されています。エクスポートは、コードの再利用性を高め、大規模なアプリケーションの開発を助ける重要な機能です。

具体的には**構文 12-1** のように export キーワードを使って変数または関数を宣言します。

構　文　12-1　export 宣言

```
export 変数 または 関数;
```

リスト 12-1 は変数と関数を 1 つのモジュール（ここではリスト 12-1.ts）にまとめ、エクスポートする例です。この例では、変数 msg と version、関数 showMsg の前に export キーワードを追加することで、これらの要素を外部ファイルから利用できるようにエクスポートをしています。

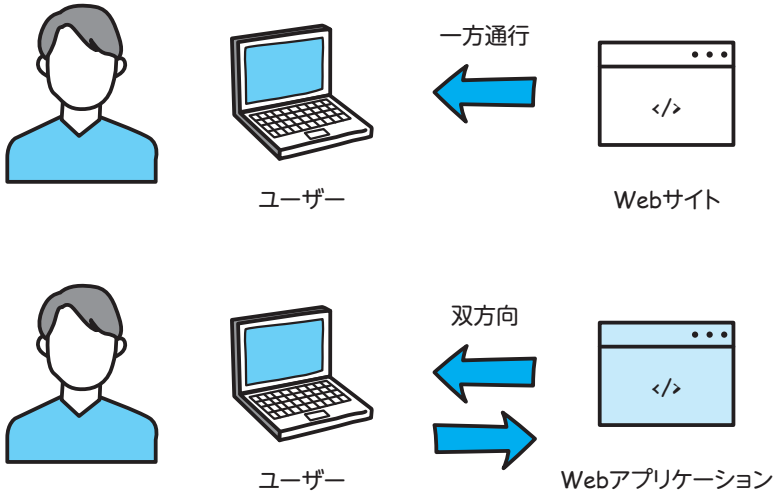
02 Reactとは

Keyword ☑ Webサイト ☑ Webアプリケーション ☑ React ☑ DOM ☑ 仮想DOM ☑ SPA

Webサイトと Webアプリケーションの違い

HTMLページがどのようなものかを理解できたら、Webサイトと Webアプリケーションの違いについて理解しておきましょう（**図 13-2**）。

▼図 13-2　Webサイトと Webアプリケーションのイメージ



Webサイトは、情報を提供するための静的なコンテンツの集まりです。情報を閲覧させることが主な目的であり、データの入力や特定の操作を行わせることはありません。Webサイトは一般的に HTMLや CSSなどで作成され、ニュースサイトや技術記事など、コンテンツが固定されていることが一般的です。

Webアプリケーションとは、ユーザーとの双方向のやり取りを可能にする Web上で使用できるアプリケーションです。ユーザーが入力した情報に基づいて、何かしらの処理を行い、結果を表示します。Webアプリケーションは、データベースと連携して情報の保存や処理を行うことが一般的です。まとめると、Webサイトは一方的な情報の提供を目的とし、Webアプリケーションはユーザーとアプリケーションの双方向でデータのやりとりをして、その場に応じた情報を作成することを目的としています。

Reactってなんだろう？

React（https://ja.react.dev/）は、冒頭でも説明したとおり、Meta社（旧 Facebook社）とコミュニティが開発した JavaScriptライブラリで、ユーザーインターフェース（UI）を構築するためのツールです。

Reactは、シングルページアプリケーション（SPA）や Webアプリケーションのフロントエンド開発に非常に人気があります。

シングルページアプリケーションとは、Webアプリケーションの一種で、1つの HTMLページで、動的に（＝必要に応じて）コンテンツを切り替えることで、あたかも複数ページから構成されているような仕組みを持つアプリケーションのことです。

Reactは、コンポーネントベースの開発を得意としています。コンポーネントとは、独立して動作する小さな部品のことです。コンポーネントは、再利用可能で、組み合わせて使用することができます。例えば、ボタンやテキスト入力、ドロップダウンやラジオボタンなどをコンポーネントとして作成し、それらを組み合わせて UIを構築します（**図 13-3**）。