クラス

JavaScriptとJava風オブジェクト指向文法の歴史

クラスはプログラミング言語によってはとても重要な位置付けにあります。たとえば、Javaはすべての要素がクラスに属し、そのクラスの組み合わせでプログラムを作成していきます。クラスの利用方法を学び、クラスの実装方法を学び、よりよいクラスの設計方法を知ることがJavaにおいてはとても重要です。また、20年ほど前はJavaはプログラミングのパラダイムにおいては最先端であり、クラスを元にした設計手法、デザインパターン、アジャイルソフトウェア開発のプログラミング系のプラクティス（テスト駆動開発やリファクタリング）を通じて、多くのプログラミング言語にJava流の設計が輸出されていきました。クラスを使いこなすことで、次のことが実現可能になるとさかんに喧伝されていました。

- 大規模なコード
- 再利用性の高いコード

例えばPythonやRubyなどの今は古参扱いのプログラミング言語の多くも、当時は機能拡張を積極的に行っておりJavaやその周辺のベストプラクティスに影響を受けたと思われるライブラリなどがいくつか組み込まれています。クラスを持ってその先端の設計技法が利用できることは、当時の一級言語のステータスでした。

JavaScriptはクラスをダイレクトに表現する文法はなかったものの、昔は関数と`prototype`という属性をいじくり回してクラスを表現していました。正確には処理系的にはクラスではないのですが、コードのユーザー視点では他の言語のクラスと同等なのでここではこの当時の記法で作るものもクラスとして扱います。とはいえ、それでもJavaScriptはクラスがないことを理由に「大規模開発に向かないおもちゃ言語」と言われることもありました。

その欠点をカバーしてJavaなどのような書き味を求めて、ひと昔前のJavaScript界隈では、この`prototype`の仕組みをラップした自前の`extends`関数みたいなのを作ってクラスっぽいことを表現しようという一派も一時期いました。また、5年ほど前までは、クラスを使うためにCoffeeScriptに救いを求める人も多数いました。今のJavaScriptとTypeScriptでは、言語の標準機能として良い書き方が提供されています。

クラスをようやくサポートしたものの、近年ではクラスを使わない書き方、関数型言語のエッセンスを取り入れた書き方も流行ってきています。大規模といえばオブジェクト指向でクラス、という短絡的な言い方をする人はもはや絶滅危惧種です。

しかし、大規模なアプリケーションの部品としてではなく、末端の部品としては便利です。みなさんも、文字列や配列などの組み込み型を通じて、自然とオブジェクト指向には触れています。あのような部品を自分で作るための手段です。関数型のスタイルとも組み合わせて利用できますし、書き方を学んでおきましょう。

用語の整理

オブジェクト指向言語は、それぞれの言語ごとに使っている言葉が違うので、それを一旦整理します。本書では次の用語で呼びます。TypeScriptの公式ドキュメント準拠です。

- クラス（class）

  他の言語のクラスと一緒です。ES2015以前のJavaScriptにはかつてなかったものです（似たようなものはありました）。

- インスタンス（instance）

  クラスを元にして `new` を呼び出して作ったオブジェクトです。

- メソッド（method）

  他の言語では、メンバー関数と呼んだり、フィールドと呼んでいたりします。名前を持ち、ロジックを書く場所です。自分が属しているインスタンスのプロパティやメソッドにアクセスできます。

- プロパティ（property）

  他の言語では、メンバー変数と呼んだり、フィールドと呼んでいたりします。名前を持ち、指定された型のデータを保持します。インスタンスごとに別の名前空間を持ちます。

基本のクラス宣言

最初はコンストラクタ関数を作り、その `prototype` 属性を操作してクラスのようなものを作っていました。 今時の書き方は次のような `class` を使った書き方になり、他の言語を使っている人からも親しみやすくなりました。

なお、JavaScriptでは不要ですが、TypeScriptではプロパティの定義をクラス宣言の中で行う必要があります。定義していないプロパティアクセスはエラーになります。

クラスの表現

```
// 新しいクラス表現
class SmallAnimal {
    // プロパティは名前: 型
    animaltype: string;

    // コストラクタ（省略可能）
    constructor() {
        this.animaltype = "ポメラニアン";
    }

    say() {
        console.log(`${this.animaltype}だけどMSの中に永らく居たBOM信者の全身の毛をむしりたい`);
    }
}

const smallAnimal = new SmallAnimal();
smallAnimal.say();
// ポメラニアンだけどMSの中に永らく居たBOM信者の全身の毛をむしりたい
```

以前の書き方は次の通りです。

旧来のクラスのようなものの表現

```
// 古いクラスの表現
// 関数だけどコンストラクタ
function SmallAnimal() {
    this.animaltype = "ポメラニアン";
}
// こうやって継承
SmallAnimal.prototype = new Parent();

// こうやってメソッド
SmallAnimal.prototype.say = function() {
    console.log(this.animalType + "だけどMSの中に永らく居たBOM信者の全身の毛をむしりたい");
};
```

アクセス制御（`public` / `protected` / `private`）

TypeScriptにはC++やJavaのような `private` と `protected` 、 `public` 装飾子があります。 メンバー定義の時の `public` 装飾子は基本的につけてもつけなくても結果は変わりませんので、コメントのようなものです。 権限の考え方も同じで、 `private` は定義があるクラス以外からの操作を禁止、 `protected` は定義のあるクラスと子クラス以外からの操作を禁止、 `public` は内外問わず、すべての操作を許可、です。 オブジェクト指向言語だとRubyがやや特殊で、 `private` は「同一インスタンスからの操作のみを許可」ですが、これとは違う動作になります。

アクセス制御

```
// 小型犬
class SmallDog {
  // 小型犬は宝物を秘密の場所に埋める
  private secretPlace: string;

  dig(): string {
    return this.secretPlace;
  }

  // 埋める
  bury(treasure: string) {
    this.secretPlace = treasure;
  }
}

const miniatureDachshund = new SmallDog();
// 埋めた
miniatureDachshund.bury("骨");

// 秘密の場所を知っているのは小型犬のみ
// アクセスするとエラー
// error TS2341: Property 'secretPlace' is private and
// only accessible within class 'SmallDog'.
miniatureDachshund.secretPlace;

// 掘り出した
console.log(miniatureDachshund.dig()); // 骨
```

古くはJavaScriptではさまざまなトリックを使って `private` 宣言を再現しようといろいろなテクニックが作られました。 もはや使わない、と前章で紹介した即時実行関数も、すべて `private` のようなものを実現するためのものでした。 それ以外だと、簡易的に `_` をメンバー名の前につけて「仕組み上はアクセスできるけど、使わないでね」とコーディング規約でカバーする方法もありました。

また `protected` は継承して使うことを前提としたスコープですが、Javaはともかく TypeScriptでは階層が深くなる継承をすることはまずないので、使うことはないでしょう。

コンストラクタの引数を使ってプロパティを宣言

TypeScript固有の書き方になりますが、コンストラクタ関数にアクセス制御の装飾子をつけると、それがそのままプロパティになります。 コンストラクタの引数をそのまま同名のプロパティに代入します。

プロパティ定義をコンストラクタ変数に

```
// 小型犬
class SmallDog {
  constructor(private secretPlace: string) {
  }

  dig(): string {
    return this.secretPlace;
  }

  // 埋める
  bury(treasure: string) {
    this.secretPlace = treasure;
  }
}
```

これはコンストラクターの引数になったので、初期化時に渡してあげると初期化が完了します。

```
const miniatureDachshund = new SmallDog("フリスビー");

// 掘り出した
console.log(miniatureDachshund.dig()); // フリスビー
```

`static` メンバー

オブジェクトの要素はみな、基本的に `new` をして作られるインスタンスごとにデータを保持します。メソッドも `this` は現在実行中のインスタンスを指します。 `static` をつけたプロパティは、インスタンスではなくてクラスという1つだけの要素に保存されます。 `static` メソッドも、インスタンスではなくてクラス側に属します。

プロパティ定義をコンストラクタ変数に

```
class StaticSample {
  // 静的なプロパティ
  static staticVariable: number;
  // 通常のプロパティ
  variable: number;

  // 静的なメソッド
  static classMethod() {
    // 静的なメソッドから静的プロパティは ``this.`` もしくは、 ``クラス名.`` で参照可能
    console.log(this.staticVariable);
    console.log(StaticSample.staticVariable);
    // 通常のプロパティは参照不可
    console.log(this.variable);
    // error TS2339: Property 'variable' does not exist on
    //     type 'typeof StaticSample'.
  }

  method() {
    // 通常のメソッドから通常のプロパティは ``this.`` で参照可能
    console.log(this.variable);
    // 通常のメソッドから静的なプロパティは ``クラス名.`` で参照可能
    console.log(StaticSample.staticVariable);
    // 通常のメソッドから静的なプロパティを ``this.`` では参照不可
    console.log(this.staticVariable);
    // error TS2576: Property 'staticVariable' is a static
    //     member of type 'StaticSample'
  }
}
```

Javaと違って、すべての要素をクラスで包む必要はないため、`static` メンバーを使わずにふつうの関数や変数を使って実装することもできます。静的メソッドが便利そうな唯一のケースとしては、インスタンスを作る特別なファクトリーメソッドを実装するぐらいでしょうか。次のクラスは図形の点を表現するクラスですが、`polar()` メソッドは極座標を使って作成するファクトリーメソッドになっています。

```
class Point {
  // 通常のコンストラクタ
  constructor(public x: number, public y: number) {}

  // 極座標のファクトリーメソッド
  static polar(length: number, angle: number): Point {
    return new Point(
      length * Math.cos(angle),
      length * Math.sin(angle));
  }
}

console.log(new Point(10, 20));
console.log(Point.polar(10, Math.PI * 0.25));
```

静的なプロパティを使いすぎると、複製できないクラスになってしまい、テストなどがしにくくなります。あまり多用することはないでしょう。

インスタンスクラスフィールド

JavaScriptではまだStage 3の機能ですが、TypeScriptですでに使える文法として導入されているがインスタンスクラスフィールド[1][2]という文法です。 この提案にはいくつかの文法が含まれていますが、publicメンバーのみをここで紹介します。

イベントハンドラにメソッドを渡す時は、メソッド単体を渡すと、オブジェクト引き剥がされてしまって `this` が行方不明になってしまうため、これまでは `bind()` を使って回避していたことはすでに紹介しました。 インスタンスクラスフィールドを使うと、クラス宣言の中にプロパティ宣言を書くことができ、オブジェクトがインスタンス化されるときに設定されます。 このときにアロー関数が利用できるため、イベントハンドラにメソッドをそのまま渡しても問題なく動作するようになります。

アロー関数を単体で使っても便利ですが、Reactの `render()` の中で使うと、表示のたびに別の関数オブジェクトが作られたと判断されて、表示のキャッシュがうまく行われずにパフォーマンスが悪化する欠点があります[3]。 インスタンスクラスフィールドとして定義すると、コンストラクタの中で一回だけ設定されるだけなので、この問題を避けることができます。

```
// 新: インスタンスクラスフィールドを使う場合
class SmallAnimal {
  // プロパティを作成
  fav = "小田原";
  // メソッドを作成
  say = () => {
    console.log(`私は${this.fav}が好きです`);
  };
}
```

以前は `bind()` を使ってコンストラクタの中で設定していました。 インスタンスクラスフィールドもコンストラクタ実行のときに実行されるので、実行結果は変わりません。

```
// 旧: bindを使う場合
class SmallAnimal {
  constructor() {
    this._fav = "小春日";
    this.say = this.say.bind(this);
  }

  say() {
    console.log(`私は${this._fav}が好きです`);
  };
}
```

注釈

ECMAScript側のインスタンスクラスフィールドの仕様では `private` の定義は `private` キーワードではなくて `#` を名前の前につける記法が提案されています。

1 https://github.com/tc39/proposal-class-fields

2 Babelでは@babel/plugin-proposal-class-propertiesプラグインを導入すると使えます

3 https://medium.freecodecamp.org/why-arrow-functions-and-bind-in-reacts-render-are-problematic-f1c08b060e36

読み込み専用の変数（ `readonly` ）

変数には `const` がありましたが、プロパティにも `readonly` があります。 `readonly` を付与したプロパティは、プロパティ定義時および、コンストラクタの中身でのみ書き換えることができます。

```
class SimLockPhone {
  readonly carrier: string;
  constructor(carrier: string) {
    this.carrier = carrier;
  }
}

// キャリア変更できない！
const myPhone = new SimLockPhone("Docomo");
myPhone.carrier = "au";
// error TS2540: Cannot assign to 'carrier' because it is a read-only property.
```

なお、通常のプロパティ定義以外にも、コンストラクタを使ったプロパティ定義、インスタンスクラスフィールドの定義で使うことができます。 また、アクセス制御と一緒に使う場合は、 `readonly` をあとにしてください。

```
class BankAccount {
  constructor(private readonly accountNumber) {
  }
}
```

メンバー定義方法のまとめ

外からプロパティ、メソッドに見えるものの定義の種類がたくさんありました。 それぞれ、メリットがありますので、用途に応じて使い分けると良いでしょう。 また既存のコードを読むときに、メンバーの定義のコードを確認する場合はこれのどの方法で定義されているのかを確認する

必要があります。

これ以外にも、アクセッサがあります。これについては クラス上級編 で紹介します。

サンプル	メソッド	変数	JS互換	メリット

`// プロパティ`
`secretPlace: string;`
 	
`// メンバーメソッド`
`dig(): string { return this.secretPlace; }`
	○	○	○	一番シンプルで、継承やインタフェース機

`// コンストラクタ引数` `constructor(private secretPlace: string);`		○		コンストラクタで外から定義する口とメン
`// インスタンスクラスフィールド` `private secretPlace = "フリスビー";`		○	△	初期値の設定が可能で、右辺から型が明確

継承/インタフェース実装宣言

作られたクラスを元に機能拡張する方法がいくつかあります。そのうちの1つが継承です。

```
class SmallAnimal {
  eat() {
    console.log("中本を食べに行きました");
  }
}

class Pomeranian extends SmallAnimal {
  eat() {
    console.log("シュークリームを食べに行きました");
  }
}
```

もう1つ、インタフェースについては前章で説明しました。前章ではオブジェクトの要素の型定義として紹介しましたが、クラスとも連携します。 むしろJavaで導入された経緯を考えると、こちらの用途の方が出自が先でしょう。

```
interface Animal {
  eat();
}

class SmallAnimal implements Animal {
  eat() {
    console.log("中本を食べに行きました");
  }
}
```

インタフェースは、クラスが実装すべきメソッドやプロパティを定義することができ、足りないメソッドなどがあるとエラーが出力されます。

```
// インタフェースで定義されたメソッドを実装しなかった
class SmallAnimal implements Animal {
}
// error TS2420: Class 'SmallAnimal' incorrectly implements interface 'Animal'.
     Property 'eat' is missing in type 'SmallAnimal' but required in type 'Animal'.
```

今、この `eat()` メソッドには返り値が定義されていません。もしコンパイルオプションが `compilerOptions.noImplicitAny` の場合、ここでエラーが発生します。

インタフェースの返り値の型を省略すると・・・

```
interface Animal {
  eat();
}
// error TS7010: 'eat', which lacks return-type annotation,
//   implicitly has an 'any' return type.
```

明示的に `void` をつけたり、型情報をつけるとエラーは解消されます。

返り値を返さない関数には*void*をつける

```
interface Animal {
  eat(): void;
}
```

関数のところの型定義で紹介したように、TypeScriptは実際のコードの情報を元に、ソースコードを解析して返り値の型を推測します。 しかし、このインタフェースには実装がないため、推測ができず、常に `any` （なにかを返す）という型になってしまいます。 これは型チェックを厳密に行っていくには穴が空きすぎてしまいエディタの補助が聞かなくなって開発効率向上が得にくくなります。 `noImplicitAny` というオプションを使うとこの穴を塞げます。 そのため、「何も返さない」という型も含め、手動で型をつける必要があります。

クラスとインタフェースの違い・使い分け

クラスとインタフェースは宣言は似ています。

違いがある点は以下の通りです。

- クラスをもとに `new` を使ってインスタンスを作ることはできるが、インタフェースはできない
- インタフェースはインスタンスが作れないので、コンストラクタを定義できない
- インタフェースは `public` メンバーしか定義できないが、クラスは他のアクセス制御も可能

継承とかオブジェクト指向設計とか方法論とかメソッドはメッセージで云々とか語り出すと大抵炎上するのがオブジェクト指向とかクラスの説明の難しいところです。 これらの機能は、言語の文化とか、他の代替文法の有無とかで使われ方が大きく変わってきます。

TypeScript界隈では、Angularなどのフレームワークではインタフェースが多用されています。 ユーザーが実装するコンポーネントなどのクラスにおいて、Anguarが提供するサービスを受けるためのメソッドの形式が決まっていて、実装部分の中身をライブラリユーザーが実装するといった使われ方をしています。 `OnInit` をimplementsすると、初期化時に呼び出されるといった具合です。

継承が必要となるのは実装も提供する必要がある場合ですが、コードが追いかけにくくなるとか、拡張性のあるクラス設計が難しいとかもあり、引き継ぐべきメソッドが大量にあるクラス以外で積極的に使うケースはあまり多くないかもしれません。

しかし、TypeScriptはJavaScriptエコシステムと密接に関わっており、JavaScriptの世界にはインタフェースを提供することはできず、実装の保証をする機能が確実に動くとは限りません。 TypeScriptのように、フレームワーク側もTypeScriptで、実装コードもTypeScriptというケースで

なければ利用しにくいことが多々あります。 特に、ライブラリ側がJavaScriptで実装されている場合はクラスを使って継承、という使い方になります。

デコレータ

これもStage 2の機能[4]ですが、これもすでに多くのライブラリやフレームワークで利用されています。 TypeScriptではtsconfig.jsonの `compilerOptions.experimentalDecorators` に `true` 設定すると使えます。 使い方から内部の動きまでPython 2.5で導入されたデコレータと似ています。 決まった引数とレスポンスを持つ関数を作り、 `@` の記号をつけて、クラスなどの前に付与すると、宣言が完了したオブジェクトなどが引数に入ってこの関数が呼ばれます。 他の言語でアトリビュートと呼ばれる機能と似ていますが、動的言語なので型情報の追加情報として設定されるのではなく、関数を通じてそれが付与されている対象のクラスやメソッド、属性を受け取り、それを加工する、変更する、記録するといった動作をします。 たとえば、ウェブアプリケーションでURLとメソッドのマッピングをデコレータで宣言したり、関数実行時にログを出すようにする、権限チェックやバリデーションを追加する、メソッドを追加するなど、用途はかなり広いです。 また、複数のデコレータを設定したりもできます。

次のコードは引数のないクラスデコレータの例です。 クラスに付与するもの、属性に付与するもの、それぞれ引数を持つものと持たないものがあるので、書き方が4通りありますが、詳細は割愛します。

デコレータでクラスにメソッドを追加する

```
function StrongZero(target) {
  target.prototype.drink = function() {
    console.log("ストロングゼロを飲んだ");
  };
  return target;
}

@StrongZero
class SmallAnimal {
}

const sa = new SmallAnimal();
sa.drink();
```

4 Babelでは@babel/plugin-proposal-decoratorsプラグインが必要です。

まとめ

クラスにまつわる数々の機能を取り上げて来ました。 昔のJavaScriptをやっていたプログラマーから見ると、一番変化と進歩を感じるところがこのクラスでしょう。 一般的なクラスの機能を備えた上で、型チェックも行われ、さらにデコレータなど追加機能なども含まれました。 TypeScriptの

場合は、エディタによるコード補完の正答率が大幅に上がったりしてリターンが大きいため、生産性の高まりを感じられるでしょう。

いろいろと機能は多いですが、TypeScriptでは、あまりクラスの細かい機能を多用するコーディングは行われていません。 そのため、本章で取り上げた機能のうち、使わない機能も多いはずです。 ちょっとしたロジックが書ける（バリデーションなど）構造体、といった感じで使われることがほとんどでしょう。 最重要なところをピックアップするとしたら次のあたりです。

- 基本のクラス宣言
- アクセス制御（ `public` / `private` ）
- インスタンスクラスフィールド
- インタフェース実装宣言

次のものは覚えておいても損はないでしょう。

- `static` メンバー
- コンストラクタの引数を使ってプロパティを宣言
- 読み込み専用の変数（ `readonly` ）

次の機能はライブラリを提供する側が覚えておくとおしゃれな機能です。

- デコレータ

次の機能をTypeScriptで駆使するようになったら警戒しましょう。 まず、2段、3段、4段と続くような深い継承になるようなコードを書くことはないでしょう。 `private` はともかく継承を前提とする `protected` 、抽象クラスを多用するような複雑なクラス設計がでてきたら、アプリケーションコードレベルではほぼ間違いだと思います。 もしかしたら、DOMに匹敵するような大規模なクラスライブラリを作るのであれば、抽象クラスだとか `protected` も活躍するかもしれませんが、ほぼ稀でしょう。 せいぜいインタフェースを定義して、特定のメソッドを持っていたら仲間とみなす、ぐらいのダックタイピングとクラス指向の中間ぐらいがTypeScriptのスイートスポットだと思います。

- アクセス制御（ `protected` ）
- 継承

アプリケーション開発者は使わないが、ライブラリ・フレームワーク実装者は使うかもしれない機能は、上級編として、 クラス上級編 の章で紹介します。 次の要素について紹介します。

- アクセッサ
- 抽象クラス