クラス上級編

本章では、アプリケーション開発者は使わないかもしれないが、ライブラリやフレームワーク開発者が使うかもしれない機能を紹介します。

アクセッサ

プロパティのように見えるけど、実際には裏でメソッド呼び出しが行われ、ちょっとした気の利いたをできるようにすることをするのがアクセッサです。 メンバーのプロパティへの直接操作はさせないが、その読み込み、変更時に処理を挟むといったことが可能です。 登場する概念としては基本的には次の3つです。 `get` だけを設定すれば取得のみができる読み込み専用とかもできます。

- 外部からは見えないprivateなプロパティ
- 値を返すゲッター（getter）
- 値を設定するセッター（setter）

どちらかというと、ライブラリ実装者が使うかもしれない文法です。アプリケーションユーザーの場合、Vue.jsのTypeScriptのクラス用の実装方法では、ゲッターをcomputed propertyとして扱いますので、Vue.jsユーザーに関しては積極的に使うことになるでしょう。

例えば、金額を入れたら、入り口と出口でビット演算で難読化（と言えないような雑な処理ですが）をする銀行口座クラスは次のようになります。

アクセッサ

```
class BankAccount {
  private _money: number;

  get money(): number {
    return this._money ^ 0x4567;
  }

  set money(money: number) {
    this._money = money ^ 0x4567;
  }
}

const account = new BankAccount()

// 1000円入れた！
account.money = 1000;

// 表示すると...
console.log(account);
//   BankAccount { _money: 18063 }

// 金額を参照すると正しく出力
console.log(account.money);
// 1000
```

Javaとかでよく使われるユースケースは、`private`でメンバー変数を用意し、それに対する`public`なアクセッサを用意するというものです。 JavaにはこのようなTypeScriptと同等のアクセッサはなく、`getField()` `setField()`という命名規則のメソッドがアクセッサと呼ばれています。 Javaはバイトコードエンハンサーを使って処理を挟むなどをすることもあって、重宝されています。

TypeScript（の元になっているJavaScript）にはプライベートフィールドはES2019まではありませんでした。メンバーフィールドもメソッドも、同じ名前空間に定義されます。`name`というプロパティを設定する場合、プライベートなメンバーの名前が`name`だとすると、アクセッサ定義時に衝突してしまいます。メンバーフィルド名には`_`を先頭につけるなどして、名前の衝突を防ぐ必要があります。

ですが、JavaScriptの世界では「すべてを変更しない、読み込み専用オブジェクトとみなして実装していく」という流れが強くなっていますし、もともと昔のJavaScriptでは定義するのが面倒だったり、IDEサポートがなかったためか、Javaのオブジェクト指向的なこの手のアクセッサーを逐一実装する、ということはあまり行われません。 属性が作られた時から変更がないことが確実に分かっているなら`readonly`の方が良いでしょう。

アクセッサとして書く場合、ユーザーは単なるプロパティと同等の使い勝手を想定して利用してきます。そのため、あまりに変な動作を実装することはしないほうが良いでしょう。また、TypeScriptでは普通にプロパティを直接操作させるのを良しとする文化なので、アクセッサを使うべき場面はかなり限られます。筆者が考える用途は次の用途ぐらいでしょう。

- 通常の型システムではカバーできない入力値のチェック（例えば、正の整数のみで、負の数はエラーにする）
- 出力時の正規化（テキストはすべて半角の小文字にまとめるなど）
- 過去に実装されたコードとプロパティ名が変わってしまい、別名（エイリアス）を定義したい

抽象クラス

インタフェースとクラスの間ぐらいの特性を持つのが抽象クラスです。 インタフェースとは異なり実装を持つことができます。 メソッドに `abstract` というキーワードをつけることで、子クラスで継承しなければならないメソッドを決めることができます。 子クラスで、このメソッドを実装しないとエラーになります。 `abstract` メソッドを定義するには、クラスの宣言の前にも `abstract` が必要です。

抽象クラスは実装も渡せるインタフェース

```
abstract class Living {
  abstract doMorningTask(): void;
  doNightTask() {
    console.log("寝る");
  }
}

class SalaryMan extends Living {
  doMorningTask() {
    console.log("山手線に乗って出勤する");
  }
}

class Dog extends Living {
  doMorningTask() {
    console.log("散歩する");
  }
}
```

Javaではおなじみの機能ですが、TypeScriptで使うことはほぼないでしょう。

まとめ

クラスの文法のうち、アプリケーション開発者が触れる機会がすくないと思うものを取り上げました。

- アクセッサ
- 抽象クラス