クラス

目次

0.クラスとは

汎用的に使える鋳型(設計図)と表現されることが多い。 関数と同じく、宣言のみしている状態のため実行するためには実体を作る必要 がある。 ES2015 以前はクラスの文法がなかったため、コンストラクタでオブジェクトを 初期化していた。

```
//コンストラクタ関数(設計図)
function Goods(name, price) {
  this.name = name;
  this.price = price;
  this.show = function () {
    console.log(`${this.name}は${this.price}円です`);
  };
}
//インスタンス化(実体)
const g1 = new Goods("チョコチップクッキー", 580);
g1.show();
```

一部、汎用的な関数はプロトタイプに定義していた。クラスの 1 つ上の親のようなイメージ(デフォルトで勝手に継承されている。)メモリの消費を減らすことができる

```
function Goods(name, price) {
  this.name = name;
  this.price = price;
}
Goods.prototype.show = function () {
  console.log(`${this.name}は${this.price}円です`);
};
//インスタンス化(実体)
const g1 = new Goods("チョコチップクッキー", 580);
g1.show();
```

余談だが、this の定義もきちんとすると以下のような書き方になる

```
function Goods(this: GoodsFunc, name: string, price: number) {
  this.name = name;
  this.price = price;
  this.show = function () {
    console.log(`${this.name}は${this.price}円です`);
  };
}

type GoodsFunc = {
  name: string;
  price: number;
  show(): void;
};
```

用語の整理

用語	概要
クラス (class)	何かを作るための設計図。必要な材料や処理が書かれている。どれだけの容量(メモリ)が必要か分からないのでまずは箱を用意しているようなイメージ
インスタンス(instance)	クラスを元にnew演算子を使い、作った実体のオブジェクト(集合)。
メソッド (method)	他の言語では、メンバー関数と呼ばれる。ロジックを書く場所
プロパティ (property)	他の言語では、メンバー変数、フィールドと呼ばれる。名前を持ち、指定された型のデータを保持する。インスタンスごとに別の名前空間を持つ。

1.クラスを定義する

クラス名、プロパティ名、メソッド名はルールに乗っ取り自由に設定できるが、コンストラクタは`constructo`で固定。

```
//`Goods`クラス(Goods設計図)
class Goods {
  //string型の`name`というプロパティ
  name: string;
  //number型の`price`というプロパティ
  price: number;
  constructor(name: string, price: number) {
    this.name = name;
    this.price = price;
  }
  //戻り値がstring型の`show`という名前のメソッド
  show(): string {
    return `${this.name}は${this.price}円です。`;
  }
}
//Goodsクラスからオブジェクトを生成して、`g`という変数に格納する
let g = new Goods("チーズケーキ", 690);
//コンソールにpオブジェクトのshowメソッドを呼び出す。
console.log(g.show());

//クラスに対してインスタンス化せず、直接メソッドを呼び出すのはNG。
console.log(Goods("チーズタルト", 500).show());
```

サンプルコード：3_am_sample1_1

2.アクセス修飾子

用語	概要
`public`	クラス外からも自由にアクセス可能(既定なので敢えて書く必要はないが、コメントの意味合いで書く場合がある)
`protected`	同じクラス、またはその子クラス(派生クラス)のメンバーからのみアクセス可能
`private`	同じクラスからのみアクセス可能

```
class Goods {
  //クラス内のみアクセスできるstrign型の`name`プロパティ
  private name: string;
  //クラス内のみアクセスできるnumber型の`price`プロパティ
  private price: number;
  constructor(name: string, price: number) {
    this.name = name;
    this.price = price;
  }
  //クラス外からも自由にアクセスできる戻り値がstringの`show`メソッド
  //書かなくても同じ意味だが明示的に書いている
  public show(): string {
    return `${this.name}は${this.price}円です。`;
  }
}

let g = new Goods("チョコチップクッキー", 580);
//クラス外からもアクセス可能なshowメソッドにアクセス
console.log(g.show());
//クラス内のみアクセスできるnameプロパティにアクセスしているのでエラーがでる
console.log(g.name);
```

サンプルコード：3_am_sample2_2

3.コンストラクタとプロパティ設定

コンストラクタはクラスからオブジェクトを生成する際に必ず実行される関数。 その性質上、与えられた引数をもとにプロパティを初期化する用途に用いられることが多い。

```
class Goods {
  name: string;
  private price: number;
  //インスタンス化する際に渡されるstring型の`name`変数とnumber型の`price`変数
  constructor(name: string, price: number) {
    //このクラスのnameプロパティに引数nameの値を代入
    this.name = name;
    //このクラスのpriceプロパティに引数priceの値を代入
    this.price = price;
  }
}

let g = new Goods("チョコチップクッキー", 580);
//クラス内からしかアクセスできないpriceプロパティに外部からアクセスするのはNG
g.price = 800;
```

サンプルコード：3_am_sample3_3

4.getter/setter アクセッサ

プロパティのように見えるが、実際は裏でメソッドを呼び出し処理を行っているもの。

アクセッサ種類	概要
getter	値を返すアクセッサ
setter	値を節制するアクセッサ

- 読み書きを制御できる
- `set` を使うことで読み取り専用のプロパティを、`get` を使うことで書き込み専用のプロパティを表せる
- 値のチェック/戻り値の加工などが可能
- getter/setter はコードブロックなので、値を取得/設定する際に値のチェック/加工などの処理を差し込める(部品としてより高い品質を保証できる)

```
class Goods {
  //class内からしかアクセスできない_priceプロパティはnumber型
  //!はプログラマがコンパイラに対してこの変数はundefinedやnullになることはないと伝える記述
  private _price!: number;
  //class内からしかアクセスできない_nameプロパティはstring型
  private name!: string;

  //getterアクセサー プロパティ名price:プロパティのデータ型
  get price(): number {
    //戻り値 プロパティの値を格納するprivate変数
    return this._price;
  }
  //setterアクセサー プロパティ名price(設定値を受け取る仮引数 : 仮引数の型)
  set price(value: number) {
    //もし仮引数valueの値が0より小さい場合
    if (value < 0) {
      //エラーを返す
      throw new RangeError("priceプロパティは正数で指定してください。");
    }
    //private変数に値を格納する
    this._price = value;
  }
}

let g = new Goods();
//setterアクセサーに1000を渡す
g.price = 1000;
console.log(g.price);

//プライベートプロパティに500を渡す
g._price = 500;
console.log(g._price);

//プライベートプロパティに"Ice Cream"を渡す
g.name = "Ice Cream";
console.log(g.name);
```

サンプルコード：3_am_sample3_4

5.静的メンバー

オブジェクトの要素は、基本的にnewによって生成されたインスタンスごとにデータを保持する。 メソッドもthisは現在実行中のインスタンスを指す。 staticをつけたプロパティは、インスタンスではなくクラスという 1 つだけの要素に保存される。 staticメソッドもインスタンスではなくクラス側に属する。

```
class Shape {
  //class外からもアクセス可能 静的プロパティ PI(プロパティ名): number型 = 値
  public static PI: number = 3.14159;
  //ckass外からもアクセス可能 静的メソッド circle(仮引数名：仮引数の型):戻り値の型
  public static circle(radius: number): number {
    //仮引数radiusの値*仮引数radiusの値*このclassのプロパティPI
    return radius * radius * this.PI;
  }
}

//new演算子でオブジェクトを生成せずとも値を参照できる
console.log(Shape.PI);
//new演算子でオブジェクトを生成せずともメソッドを使用できる
console.log(Shape.circle(5));
```

サンプルコード：3_am_sample3_5

6.継承(Inheritance)

クラスを機能拡張する方法の 1 つが継承。 元となるクラスの機能(メンバー)を引き継ぎながら、新しい機能を追加したり、元の機能の一部だけを修正したりすること。

用語の整理

クラスの分類	名称
継承元となるクラス	スーパークラス、親クラス、基底クラス
継承の結果できたクラス	サブクラス、子クラス、派生クラス

```
//親クラス Goods
class Goods {
  //子クラスでアクセスできるようにprotected プロパティ名name:string型
  protected name: string;
  //子クラスでアクセスできるようにprotected プロパティ名price:number型
  protected price: number;
  //インスタンス化されるときに実行されるconstructor(仮引数name:string型,仮引数price:number型)
  constructor(name: string, price: number) {
    this.name = name;
    this.price = price;
  }
  //showメソッド:戻り値string型
  show(): string {
    return `${this.name}は${this.price}円です。`;
  }
}
//Goodsクラスを継承する子クラスBusinessGoods
class SaleGoods extends Goods {
  //priceDownメソッド：戻り値string型
  priceDown(): string {
    return `${this.name}は今だけ50%オフです。`;
  }
}
//子クラスをnewしてオブジェクトを生成し、変数pに代入
let g = new SaleGoods("チョコチップクッキー", 580);
//親クラスのshowメソッドを呼び出す
console.log(g.show());
//子クラスのpriceDownメソッドを呼び出す
console.log(g.priceDown());

//  親クラスをnewしてオブジェクトを生成し、変数pに代入
let g2 = new Goods("チーズケーキ", 600);
//親クラスのshowメソッドを呼び出す
console.log(g2.show());
//子クラスのpriceDownメソッドを呼び出す
console.log(g2.priceDown());
```

サンプルコード：3_am_sample3_6

7.メソッドのオーバーライド

親クラスで定義された内容を、子クラスで定義しなおすこと。(上書きする)親クラスのコードを完全に書き換えてもよいが、親クラスの機能を呼び出し、子クラスで独自の機能を追加することもできる。

```
class Goods {
  protected name: string;
  protected price: number;
  constructor(name: string, price: number) {
    this.name = name;
    this.price = price;
  }
  show(): string {
    return `${this.name}は${this.price}円です。`;
  }
}

class SaleGoods extends Goods {
  protected category: string;
  constructor(name: string, price: number, category: string) {
    //親クラスのコンストラクターを呼び出す
    super(name, price);
    //子クラスでは新たに、categoryプロパティにcategoryを代入
    this.category = category;
  }
  //親クラスと同名のshowメソッドをオーバーライド
  show(): string {
    return super.show() + `${this.category}です。`;
  }
}

let g = new SaleGoods("チョコチップクッキー", 580, "製菓");
console.log(g.show());

class Snack extends Goods {
  protected type: string;
  constructor(name: string, price: number, type: string) {
    super(name, price);
    this.type = type;
  }
  tasteType(): string {
    return super.show() + `${this.type}スナックです。`;
  }
}

let s2 = new Snack("チョコチップクッキー", 580, "甘い");
//親クラスのshowを呼び出す
console.log(s2.show());
```

サンプルコード：3_am_sample3_7

8 抽象メソッド

サブクラスで機能を上書きしなければならないようにする。abstrat 修飾子で表現する。 抽象クラス=抽象メソッドを含んだクラス 抽象メソッドを継承した場合、子クラス側では抽象メソッドのオーバーライドが必須。 概念のみを記述したいときなどに使う。

```
//抽象クラス shape
abstract class Shape {
  //インスタンス化するときに実行する処理(子クラスでもアクセス可能 仮引数width:number型, 子クラスでもアクセス可能 仮引数height:number型)
  constructor(protected width: number, protected height: number) {}
  //抽象メソッド  getArea:戻り値number型 中身は未定義
  abstract getArea(): number;
}

//Shapeを継承した子クラスTriangle
class Triangle extends Shape {
  //getAreaメソッド:戻り値number型
  getArea(): number {
    //計算結果を戻す
    return (this.width * this.height) / 2;
  }
}

//new演算子でTriangleオブジェクトを生成、引数は10,5
let t = new Triangle(10, 5);
//getAreaメソッドを呼び出す
console.log(t.getArea());

//Shapeを継承した子クラスCircle
class Circle extends Shape {
  getArea(): number {
    return this.width * 3.14;
  }
}

let c = new Circle(10, 5);
console.log(c.getArea());
```

サンプルコード：3_am_sample3_8-1

9.シングルトンパターン

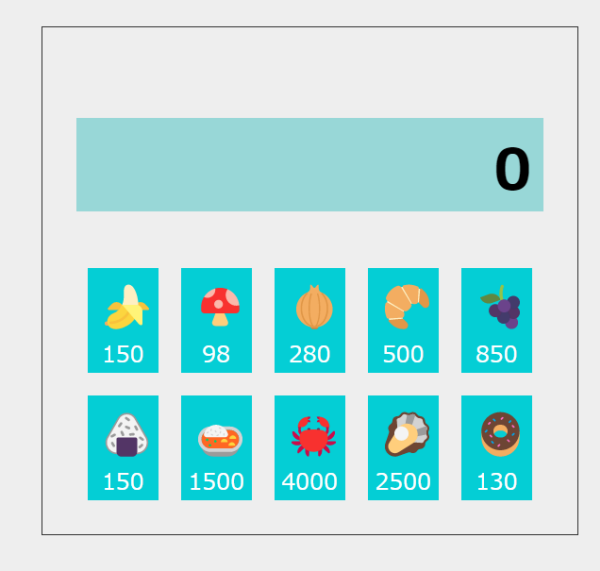
- オブジェクト指向プログラミングにおけるデザインパターン
- 特定のクラスのインスタンスが必ず 1 つしか存在しないようにする
- `constructor`に`private`をつけることで外から`new`できなくなる
- クラスのインスタンスではなくクラス自体に 1 つしか存在しない静的フィールド`static instance`を作る

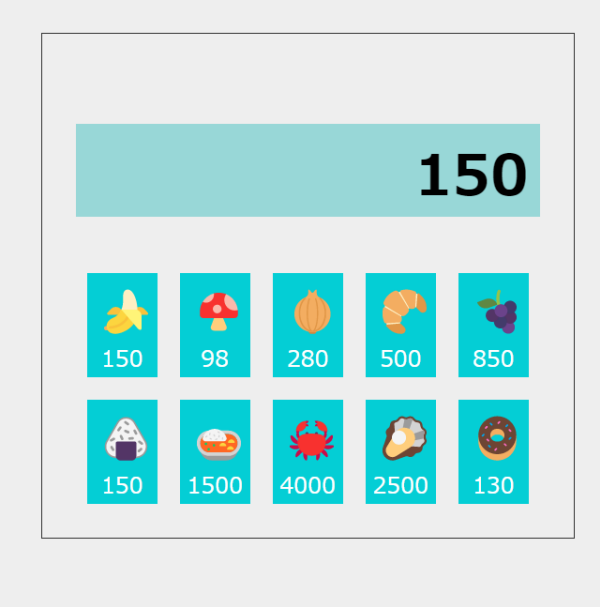
```
class Singleton {
  private static instance: Singleton;
  private constructor() {}

  public static getInstance(): Singleton {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }
  public logMethod(): void {
    console.log("Singleton!!!");
  }
}

//呼び出し
Singleton.getInstance().logMethod();
```

サンプルコード：3_am_sample3_8-2

10.クラスを使った確認問題

レジ打ちアプリを作ろう

- 初期表示
  - 

  - 10 個のアイテムと金額が表示されたボタンと合計金額エリアが表示される
- ボタンクリック時の動作
  - 

  - 例：バナナをクリックすると合計金額が 150 となる

- 例：カニをクリックすると合計金額が 4150 となる

- 例：カニをもう一度クリックするとキャンセルされ、合計金額は 150 となる

実装手順

- コードのひな形：3_am_class をダウンロードする
- `index.html`と`style.css`が含まれていることを確認
- tsconfig.json を作り、typescript 開発環境を作成
- `index.ts`を作成
- コンパイルして`dist/index.js`ができるようにする
- ブラウザで表示を確認する