オブジェクト指向集中講座

本講座の目的

- オブジェクト指向の考え方を身につける
- 学習の過程で創造力を育み、今後のクリエイティブに活かす

本日のテーマ

- 新たな言語として JavaScript を学ぶことにより、他の言語の学習方法などを学び今後の学習の幅を広げる
- オブジェクト指向について軽く触れてみてどんなものなのかを学んでみる

あなただれですか

- 小飯田といいます、インターネット上の名前は Sotono です
- AI 活用の会社をやっている、ボンギンカン株式会社のエンジニアをやっています
- もともと PC が好きで、RPG ツクールでプログラミングの基礎を学びました。
- 新潟県の美術大学でグラフィックデザインを学んで、就職先の地元の会社でプログラミングを用いた効率化を行って業績を 3 倍くらい増やしました

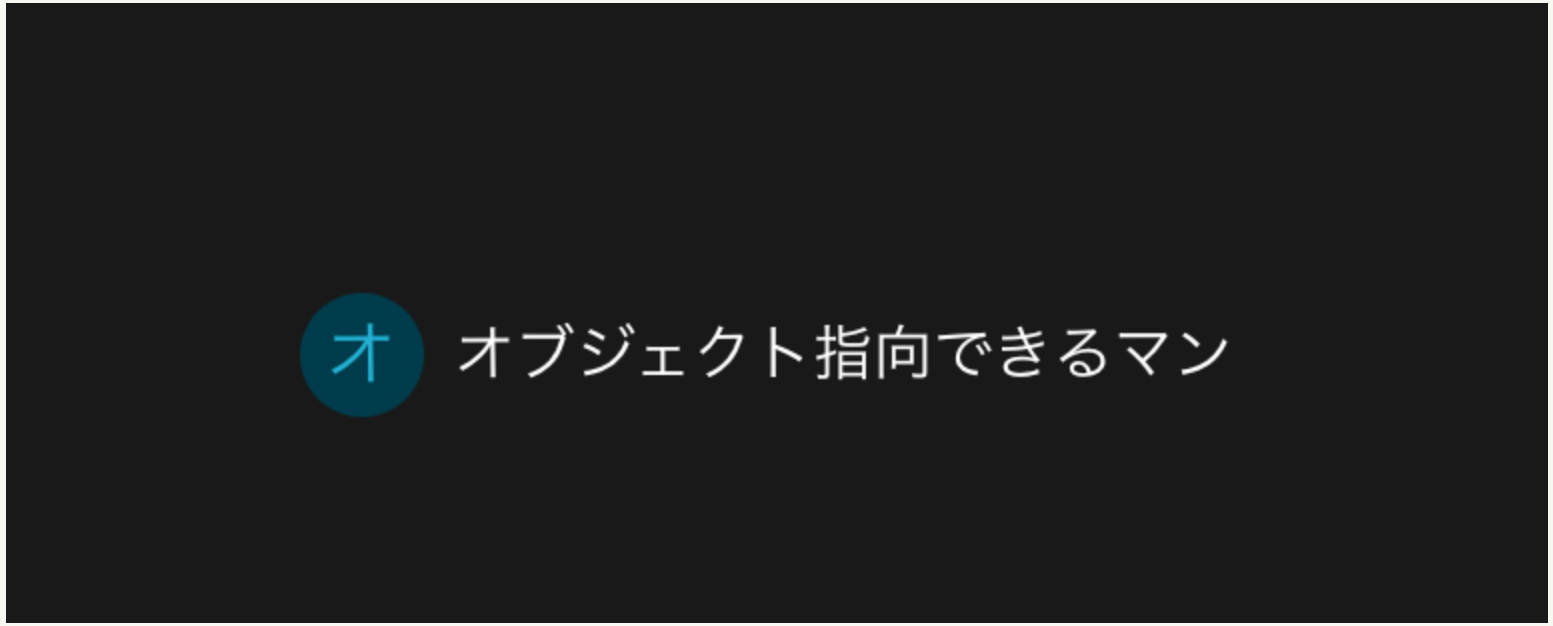
あなただれですか

- 小学校で ICT 支援員をやっていました
- Scratch を使った軽いプログラミングも教えてました
- その後、ワチャワチャしてボンギンカンに入社
- 好きな言語は Python です、名前がかっこいいので
- 趣味は音楽や読書、ゲームも好きです

https://x.com/_Sotono_046

オブジェクト指向について

...の前に



本講座ではカスタムモデル Gemini を使います
わからないことがあったら何でもバシバシ聞いて下さい
https://gemini.google.com/gem/2148289c06c1?usp=sharing

Gemini との進め方

1. まず自分で考える／手を動かす
2. わからない部分を Gemini に質問する
3. 返ってきた答えを自分の言葉に直して理解する
4. それでも難しければ講師と相談

今回使用する言語について

Java Script

以下、JS と省略します

JS 採用の理由

- 生徒間における共通言語が日本語のみ
- C、Python だと環境構築で時間や手間がかかる
- JS を使用することによって視覚的にわかりやすく結果を得ることができる
- また、C などを使ったゲーム以外にも、Web などに特化した JS を使えばいろんな道を切り開くことができる
- AI 時代で割とたくさん使うのが JS だったりする

JS の書き方について学ぶ

プログラミングの基礎は他の言語で学んでると思いますので、
下記の書き方を軽くさらっておきます。

- 変数宣言
- if
- for
- 関数

JS の書き方 〜変数

- JS における変数は `let` と `const` の二種類がある。
- `let` …後で書き換えが可能な変数
- `const` …書き換えが不可能な定数
- 基本的に理由がない限り `const` の使用を推奨します。

JS の書き方 〜変数の具体例

```
const x = 1;
const y = 2;

console.log(x + y);
```

結果

```
3
```

JS の書き方 〜if

if の書き方は下記のようにします。

```
if (条件) {
  // 処理A
} else {
  // 処理B
}
```

JS の書き方 〜if の具体例

```
const score = 75;

if (score >= 80) {
  console.log("よくできました！");
} else {
  console.log("次は80点を目指そう");
}
```

JS の書き方 〜for

- 同じ処理を繰り返すときに使う構文
- `for（初期化；条件；更新）` の 3 要素で動く
- 配列と組み合わせると一覧表示に便利

JS の書き方 〜for の例

```
const items = ["おにぎり", "お茶", "パン"];

for (let i = 0; i < items.length; i++) {
  console.log(items[i]);
}
```

JS の書き方 〜関数

- 一連の処理に名前をつけて再利用できるしくみ
- `function 名前(引数) { ... }` の形
- チームでの共通部品づくりにもつながる

JS の書き方 〜関数の例

```
function greet(name) {
  console.log(`${name}さん、ようこそ！`);
}

greet("山田");
```

オブジェクト指向って？

オブジェクト指向って？

- 「モノ」を部品として考える発想
- モノは「属性(=情報)」と「行動(=メソッド)」を持つ
- ゲームの世界観づくりと相性が良い

例題 1

- コンビニで商品が 3 つあります。
- それぞれの商品の名前と価格を決めましょう。

```
const item = "おにぎり";
const price = 120;
```

例題 1

- では、商品が増えました、10 個あります。
- それぞれの商品の名前と価格を決めましょう。

```
const item = "おにぎり";
const price = 120;
```

大変 and めんどくさい

なぜ大変 and めんどくさいなのか

- 変数が増えると書き直しが大変
- 例えば、ここに「賞味期限」を入れると全部変数一個ずつ入れないといけない
- 一覧を作るだけでコードが散らばり、冗長になる

もっと楽にかけないか？

for でまとめてみるよう

- 楽をするといえば、for ですね
- 繰り返しの条件を探って for で書いてみましょう

```
const 商品リスト = [
  { 名前: "おにぎり", 価格: 120 },
  { 名前: "お茶", 価格: 100 },
  { 名前: "パン", 価格: 150 },
];

let 合計 = 0;
for (let i = 0; i < 商品リスト.length; i++) {
  const 商品 = 商品リスト[i];
  合計 += 商品.価格;
  console.log(`${商品.名前}: ${合計}円`);
}
```

for だと何がいけない？

- 配列化するのは良いこと
- でも for で回すとお客さんによって処理を変えることができない
- 例:おにぎり以外ほしいお客さん、パンだけほしいお客さん

クラスを覗いてみる

- クラスという概念を使うことによって、for では難しい処理を簡単に書くことができる
- 後々楽ができる
- いちいち変数宣言しなくて良くなる

例題 2 クラスで楽になる

```
const おにぎり = new 商品("おにぎり", 120);
const お茶 = new 商品("お茶", 100);

おにぎり.表示する();
お茶.表示する();
```

- `new` するだけで商品が生成できる
- 表示ロジックもクラス内にまとまっていて差し替えが簡単
- 明日はこのクラスを実際に自分たちで書けるようになる

では、商品の中身はどうなっているのか？

クラスの基本構造

```
class 商品 {
  constructor(名前, 価格) {
    this.名前 = 名前; // オブジェクト持つ共通情報
    this.価格 = 価格;
  }
  // 行動(メソッド)
  表示する() {
    console.log(`${this.名前}: ${this.価格}円`);
  }
}
```

クラスの基本構造

- `constructor` は新しいオブジェクトを初期化する特別な関数
  → 例: 「レジに新商品を登録するとき」
- `this` はその瞬間に扱っているオブジェクト自身を指すキーワード
  → 例: 「他ではない"その"商品」
- メソッドはオブジェクトの振る舞いをまとめる場所
  → 例: 「その商品で何ができる？」

クラスの考え方の広がり

- この「商品クラス」の形が **オブジェクト指向の基本**
- どんなものでも「情報」と「行動」をセットで表せる
- ゲームなら「キャラクター」「アイテム」「敵」も同じ仕組みで作れる

例題 3 学生の成績管理

```
const 名前1 = "田中";
const 国語1 = 80;
const 数学1 = 75;
const 英語1 = 85;
const 平均1 = (国語1 + 数学1 + 英語1) / 3.0;
console.log(名前1 + "の平均: " + 平均1);

const 名前2 = "佐藤";
const 国語2 = 90;
const 数学2 = 88;
const 英語2 = 92;
const 平均2 = (国語2 + 数学2 + 英語2) / 3.0;
console.log(名前2 + "の平均: " + 平均2);
```

例題 3 学生の成績管理

```
class 学生 {
  constructor(名前, 国語, 数学, 英語) {
    this.名前 = 名前;
    this.国語 = 国語;
    this.数学 = 数学;
    this.英語 = 英語;
  }

  平均を表示する() {
    const 平均 = (this.国語 + this.数学 + this.英語) / 3.0;
    console.log(this.名前 + "の平均: " + 平均);
  }
}
```

例題 3 学生の成績管理

```
const 田中 = new 学生("田中", 80, 75, 85);
const 佐藤 = new 学生("佐藤", 90, 88, 92);
田中.平均を表示する();
佐藤.平均を表示する();
```

```
田中の平均: 80
佐藤の平均: 90
```

例題 4 動物の鳴き声

```
const 犬の名前 = "ポチ";
const 猫の名前 = "タマ";

console.log(犬の名前 + "が鳴きます");
console.log("わんわん！");

console.log(猫の名前 + "が鳴きます");
console.log("にゃーにゃー！");
```

例題 4 動物の鳴き声

```
class ペット {
  constructor(名前, 鳴き声) {
    this.名前 = 名前;
    this.鳴き声 = 鳴き声;
  }

  鳴く() {
    console.log(this.名前 + "が鳴きます");
    console.log(this.鳴き声 + "！");
  }
}
```

例題 4 動物の鳴き声

```
const 犬 = new ペット("ポチ", "わんわん");
const 猫 = new ペット("タマ", "にゃーにゃー");
犬.鳴く();
猫.鳴く();
```

```
ポチが鳴きます
わんわん！
タマが鳴きます
にゃーにゃー！
```

まとめ

- クラス = 設計図
- オブジェクト = 実体（設計図から生まれるもの）
- **これを覚えると、ゲーム制作やアプリ開発に大きく役立つ！**

オブジェクト指向集中講座

本講座の目的

- オブジェクト指向の考え方を身につける
- 学習の過程で創造力を育み、今後のクリエイティブに活かす
- 新たな言語として JavaScript を学ぶことにより、他の言語の学習方法などを学び今後の学習の幅を広げる

オブジェクト指向って？

- 「モノ」を部品として考える発想
- モノは「属性(=情報)」と「行動(=メソッド)」を持つ
- ゲームの世界観づくりと相性が良い

本日のテーマ

一日目で習ったオブジェクト指向の考え方を、
実際に手を動かすことによって実感し、
またそれを後半で行う自由課題で応用してみる。

問題 1:

次のコードをクラスで書き換えてください。

```
// LINE通知
const LINE名前 = "LINE";
const LINEメッセージ = "新着メッセージがあります";
console.log("[" + LINE名前 + "] " + LINEメッセージ);
console.log("通知音: ピロン♪");

// Twitter通知
const Twitter名前 = "Twitter";
const Twitterメッセージ = "新しいツイートがあります";
console.log("[" + Twitter名前 + "] " + Twitterメッセージ);
console.log("通知音: ピロン♪");
```

問題 2: 解答例

```
class アプリ {
  constructor(名前) {
    this.名前 = 名前;
  }

  通知する(メッセージ) {
    console.log("[" + this.名前 + "] " + メッセージ);
    console.log("通知音: ピロン♪");
  }
}
```

問題 2: 解答例

```
const LINE = new アプリ("LINE");
const Twitter = new アプリ("Twitter");

LINE.通知する("新着メッセージがあります");
Twitter.通知する("新しいツイートがあります");
```

問題 2

次の情報を読み解とき、これらを設計し、
クラスで実装せよ。

- 薬草を使うと HP が 10 回復する
- ポーションを使うと HP が 50 回復する
- ハイポーションを使うと HP が 200 回復する

問題 2: 解答例

```
class アイテム {
  constructor(名前, 回復量) {
    this.名前 = 名前;
    this.回復量 = 回復量;
  }

  使う() {
    console.log(this.名前 + "を使った！");
    console.log("HPが " + this.回復量 + " 回復した！");
  }
}
```

問題 2: 解答例

```
const 薬草 = new アイテム("薬草", 10);
const ポーション = new アイテム("ポーション", 50);
const ハイポーション = new アイテム("ハイポーション", 200);

薬草.使う();
ポーション.使う();
ハイポーション.使う();
```

問題 3:

次の情報を読み解とき、これらを設計し、
クラスで実装せよ。

- ポーション: HP が 50 回復
- エーテル: MP が 20 回復
- ハイエーテル: MP が 50 回復
- エリクサー: HP が 1000、MP が 1000 回復
- ステーキ: HP が 500、MP が 50 回復

問題 2: 解答例

```
class アイテム {
  constructor(名前, HP回復量, MP回復量) {
    this.名前 = 名前;
    this.HP回復量 = HP回復量;
    this.MP回復量 = MP回復量;
  }

  // 続く
}
```

問題 2: 解答例

```
class アイテム {
  // 続き
  使う() {
    console.log(this.名前 + "を使った！");
    if (this.HP回復量 > 0) {
      console.log("HPが " + this.HP回復量 + " 回復した！");
    }
    if (this.MP回復量 > 0) {
      console.log("MPが " + this.MP回復量 + " 回復した！");
    }
  }
}
```

問題 2: 解答例

```
const ポーション = new アイテム("ポーション", 50, 0);
const エーテル = new アイテム("エーテル", 0, 20);
const ハイエーテル = new アイテム("ハイエーテル", 0, 50);
const エリクサー = new アイテム("エリクサー", 1000, 1000);
const ステーキ = new アイテム("ステーキ", 500, 50);

ポーション.使う();
エーテル.使う();
エリクサー.使う();
ステーキ.使う();
```

自由課題

自分の身の回りのものを思い出し、
クラスを使って設計及び表現ができそうなものを探し出して、
それを実装し、コンソールで動かせるようにしてください。

課題のヒント

- そもそも**身の回りに何がある**のか？
- それは**どうやって動く**のか？
- その**しくみ**は他のものに応用できないか？

課題のヒント

- そのモノの **特徴（名前・大きさ・色など）** は何か？
- そのモノに **できること（行動）** は何か？
- 似たモノ同士で**共通する部分**はないか？
- もしゲームの中に登場するとしたら、どう表す？
- 現実にはできないけど、**もしできたら面白い行動**は？

オブジェクト指向のおさらい

クラスを理解する〜構成

オブジェクト指向は`class`をベースに構成される。
`class`は下記の２つで構成される。

- `constructor`...クラスに渡された引数を下にオブジェクトを初期化する
- `method`...クラスを使って行われるアクション、動作を指す。

クラスを理解する〜this とは

クラスを形作るものとして `this` が存在する。
これは、**そのオブジェクトそのもの**を指す。

例えば、ペンが複数本あれば、一本一本が一本ずつ `this` になる。
`constructor` に渡された引数が「色」であった場合、
`this.色` は**初期状態では定義されていない**ので、これを任意の色で定義する。

例

```
class pen {
  constructor(color) {
    this.color = color;
  }

  draw() {
    console.log(`I draw a line, color is ${this.color} !`);
  }
}
```

```
const signpen = new pen("red");
signpen.draw();
```

出力: `I draw a line, color is red !`

解説〜コンストラクタ

```
class pen {
  constructor(color) {
    this.color = color;
  }
  // 省略
}
```

- `constructor` で引数を使ってオブジェクトを初期化。
- `this` の `color` が、**引数で渡された** `color` になる。
- `new pen("red")` の場合、**引数で渡されているのは** `red` なので、`this.color = "red"` となる。

解説〜メソッド

```
class pen {
  // 省略
  draw() {
    console.log(`I draw a line, color is ${this.color} !`);
  }
}
```

- `pen` クラスの中にメソッドとして仕込んでおく。
- メソッドは `function` で書かれることが多い。

解説〜メソッド

```
class pen {
  // 省略
  draw() {
    console.log(`I draw a line, color is ${this.color} !`);
  }
}
```

- `const signpen = new pen("red")` の場合 `this.color = "red"` になる
- が、 `signpen` ではメソッドは実行されない。
- `signpen.draw()` として実行すると、 `I draw a line, color is red !` と出力される。

応用問題

次の場合、どのようにクラスを書けば合理的か？

- ペンが 3 本あり、それぞれ赤、青、緑の色に分かれている。
- 赤いペンは 120 円で売られている。
- 青いペンは 200 円で売られている。
- 緑のペンは 50 円で売られている。
- ペンはそれぞれ試し書きができ、メッセージと色が表示される。

解答例

```
class pen {
  constoractor(color, price) {
    this.color = color;
    this.price = price;
  }

  testDrawing(msg) {
    console.log(`${this.color}色で試し書きをしました！`);
    console.log(`内容: ${msg}`);
  }
}
```

解答例に対する疑問

- 問題文を構成する要素は、「色」「価格」「試し書きの内容」だが、なぜ初期化を決める要素が「色」「価格」 の 2 個だけなのか？
- なぜメソッドに `msg` と書かれているのか？

コンストラクタの初期化の基準

```
class pen {
  constoractor(color, price) {
    this.color = color;
    this.price = price;
  }
}
```

- `pen` に与えるべき情報は、そのペンが独自に持つ情報であるべき。
- 今回の問題では、「色」「価格」がそのペン独自の情報となっている。

コンストラクタの初期化の基準

```
class pen {
  constoractor(color, price) {
    this.color = color;
    this.price = price;
  }
}
```

- もし、試し書きの内容を加えると、
  そのペンで書かれる試し書きの内容が固定になってしまう。
- そのペンで何を書くかは個人の自由であるため、
  ペン独自の情報に「試し書きの内容」を加える必要はない。
  じゃあどこで試し書きの内容を決定するのか？

メソッドには引数が渡せる

```
class pen {
  testDrawing(msg) {
    console.log(`${this.color}色で試し書きをしました！`);
    console.log(`内容: ${msg}`);
  }
}
```

- メソッドに引数を渡すことで、オブジェクトに依存しない要素を渡すことができる。
- 忘れがちだが、今回の場合だと `color` はオブジェクト依存の要素なので、 `this` が必要。

他の言語への応用方法

- ここまで共通言語として JS を触ってきたが、他の言語だとどうなるのか？
- C++、Python、Java を例に同じ機能を持つものを書き換えてみる。
- Gemini 使って簡単に変換することもできますが、間違うこともあるので、**正しいものを見極める力**が必要。
- 基本的に if や for などと同じで、オブジェクト指向の表現方法が違うだけで共通する考え方となる。

他の言語への応用方法〜C++

```
class Pen {
public: // どこからでもアクセスOKという意味
    // コンストラクタ(初期化)
    Pen(std::string color) {
        this->color = color;
    }

    // メソッド(振る舞い)
    void draw() {
        // 日本語訳: 私は線を引きます、色は{this->color}です!
        std::cout << "I draw a line, color is " << this->color << " !" << std::endl;
    }

private:
    // 属性(データ)
    std::string color;
};
```

他の言語への応用方法〜Python

```
class Pen:
    # Pythonでの初期化メソッド(コンストラクタ)
    def __init__(self, color):
        # 属性(データ)を設定
        self.color = color

    # メソッド(振る舞い)
    def draw(self):
        print(f"I draw a line, color is {self.color} !")
```

他の言語への応用方法〜Java

```java
public class Pen {
    // 1. 属性(データ): 型を明記する (String)
    private String color;

    // 2. コンストラクタ(初期化)
    public Pen(String color) {
        this.color = color;
    }

    // 3. メソッド(振る舞い): 戻り値の型を明記する (void: 何も返さない)
    public void draw() {
        System.out.println("I draw a line, color is " + this.color + " !");
    }
}
```

実際のコードをどう考えるか？

日常例: コンビニのレシート計算

あるコンビニで、購入した商品リスト（配列）から、

1. まず**割引対象**の商品だけを選び出す（**filter**）
2. 選ばれた商品の値段を**10 円引き**にする（**map**）
3. 最終的な**合計金額**を出す（**reduce**）

という一連の処理を考えてみましょう。

1. 手続き的な書き方 (手続き型)

手続き型で書くと、`for`ループを回したり、外部の合計用変数を変更したりする必要があります。

```
// 商品リスト
const items = [
  { name: "Onigiri", price: 150, isDiscount: true }, // おにぎり
  { name: "Juice", price: 120, isDiscount: false }, // ジュース
  { name: "Sandwich", price: 300, isDiscount: true }, // サンドイッチ
];

let total = 0; // 外部の変数を準備
let discountedItems = []; // 割引後のリストを準備

// 割引処理と合計計算を同時に、データを「変更しながら」進める
for (let i = 0; i < items.length; i++) {
  const item = items[i];
  if (item.isDiscount) {
    // 10円引きを適用（新しいリストに登録）
    discountedItems.push({
      name: item.name,
      price: item.price - 10, // 値段を変更
      isDiscount: true,
    });
    total += item.price - 10; // 合計に加算
  }
}

// console.log("割引後の合計:", total); // 430
```

この書き方だと、`total`や`discountedItems`という**外部の変数の値が、ループを回るたびにコロコロ変わって**しまいます。 これがプログラムを追いにくくする原因になりがちです。

---

2. 関数型的な書き方

関数型的なアプローチでは、**データを変更せず**、処理を**小さな関数**に分け、**パイプライン（メソッドチェーン）**でつなげます。

```
// 商品リスト
const items = [
  { name: "Onigiri", price: 150, isDiscount: true }, // おにぎり
  { name: "Juice", price: 120, isDiscount: false }, // ジュース
  { name: "Sandwich", price: 300, isDiscount: true }, // サンドイッチ
];

// 1. 割引対象か判断する関数（純粋関数）
const isDiscountItem = (item) => item.isDiscount;

// 2. 10円割引を適用する関数（純粋関数）
const applyTenYenOff = (item) => ({
  // 元のデータはそのままにして、新しいオブジェクトを返す
  name: item.name,
  price: item.price - 10,
  isDiscount: item.isDiscount,
});

// 3. 合計を計算する関数（純粋関数）
// prev: ここまでの途中合計, current: 現在の要素
const sumPrices = (prev, current) => prev + current.price;

// 実行（処理の流れをメソッドチェーンで表現）
const finalTotal = items
  .filter(isDiscountItem) // 割引対象だけを「選ぶ」
  .map(applyTenYenOff) // 選ばれたもの全てに割引を「適用」
  .reduce(sumPrices, 0); // 最終的な合計を「計算」（初期値 0）

console.log(`元のリストは変更されません:`);
console.log(items);

console.log(`✅ 最終的な合計金額: ${finalTotal}`); // 430
```

違いとメリット🚀

項目	手続き型 (上の例)	関数型 (下の例)
データの変更	外部変数（`total`, `discountedItems`）を**変更**する	元のデータは**変更しない**（新しい配列や値を返す）
処理の表現	`for`ループ内に複数の処理が**混在**しがち	`filter`→`map`→`reduce`のように処理を**流れ作業**でつなぐ
可読性	変数がいつ変更されたか追うのが大変	処理の流れが上から下へ**一目瞭然**

関数型プログラミングは、**小さな役割の関数を組み合わせて、データの流れを作る**のが特徴です。 元のデータをうっかり壊してしまう心配が減り、「**この関数はこれだけをする！**」と役割がはっきりするので、コードが読みやすくなります。

了解しました！

先ほどの**「コンビニのレシート計算」**の例を、今度は**オブジェクト指向（OOP）の考え方**を使って書いてみましょう。

OOP では、データ（商品名や価格）と、それに対する操作（割引する、合計する）を一つにまとめた**「モノ（オブジェクト）」**として扱います。

今回は、商品を管理し計算を行う**レジ（`Register`）**というクラスを作って表現します。

オブジェクト指向での書き方: レジのクラス化

1. データの設計: `Item` クラス

まず、商品そのものを表すクラスを作ります。

```
// 商品クラス：商品データをまとめる
class Item {
  constructor(name, price, isDiscount) {
    this.name = name; // 商品名
    this.price = price; // 価格
    this.isDiscount = isDiscount; // 割引対象か
  }
}
```

2. 動作の設計: `Register` クラス

次に、計算や管理の機能を持つクラスを作ります。

```
// レジクラス：商品のリストを管理し、計算を実行する
class Register {
  constructor(items) {
    this.items = items; // レジが扱う商品リスト（データ）
  }

  // 割引対象の商品だけを選び、割引を適用する（振る舞い/メソッド）
  applyDiscount() {
    console.log("レジ: 割引対象をチェックしています...");

    // 元のリスト（this.items）は変更しない
    // map/filterで新しいリストを生成するのがOOPでも関数型の良い点を取り入れた書き方
    const discountedItems = this.items
      .filter((item) => item.isDiscount) // 割引対象を選ぶ
      .map((item) => ({
        // 新しい商品データを作る
        name: item.name,
        price: item.price - 10, // 割引適用
        isDiscount: item.isDiscount,
```

```
      }));

    return discountedItems; // 割引後の商品リストを返す
  }

  // 最終合計を計算する（振る舞い/メソッド）
  calculateTotal(itemsToSum) {
    console.log("レジ: 合計金額を計算しています...");

    const total = itemsToSum.reduce((prev, current) => prev +
current.price, 0);
    return total;
  }
}
```

3. 実行

データ（商品リスト）を定義し、それを元に**オブジェクト**（レジ）を作り、**メソッド**を呼び出します。

```
// 1. 商品データを作成
const initialItems = [
  new Item("Onigiri", 150, true),
  new Item("Juice", 120, false),
  new Item("Sandwich", 300, true),
];

// 2. レジオブジェクトを作成
const myRegister = new Register(initialItems);

// 3. 割引を適用したリストを取得
const discountedList = myRegister.applyDiscount();

// 4. 最終的な合計を計算
const finalTotal = myRegister.calculateTotal(discountedList);

console.log(`✅ 最終的な合計金額: ${finalTotal}円`); // 430円
```

💡 オブジェクト指向のポイント

1. カプセル化 (Encapsulation)

- **データと処理を一つにまとめました。**
  - 商品データ（`name`, `price`など）は`Item`クラスに。
  - 計算処理（`applyDiscount`, `calculateTotal`）は`Register`クラスに。
- これにより、**「レジの計算ロジックは`Register`クラスを見ればわかる」**という状態になり、コードの管理がしやすくなります。

2. 日本語的な感覚

- 私たちが普段「**レジ**が**割引**をする」「**レジ**が**合計**を出す」と言うように、
- **「モノ（オブジェクト）が、動詞（メソッド）を実行する」** という自然な日本語の感覚に近くなります。

これで、データと振る舞いがまとまり、より現実世界に近いイメージでプログラムを組むことができました！

手続き型で不安を感じていた **「どこに何を書けばいいの？」** という疑問が、 **「このモノ（クラス）は、どんなデータを持って、どんなことができるの？」** という視点に変われば、大丈夫ですよ。

これを他言語に応用する

C++で書いてみる

ここでは同じ設計思想を C++ のクラスで表現します。 データ（Item）と処理（Register）を明確に分け、 必要な操作をメソッドとしてまとめます。

```cpp
#include <iostream>
#include <numeric>
#include <string>
#include <utility>
#include <vector>

class Item {
public:
  Item(std::string name, int price, bool isDiscount)
      : name_(std::move(name)), price_(price), isDiscount_(isDiscount) {}

  Item discounted(int amount) const {
    return Item{name_, price_ - amount, isDiscount_};
  }

  int price() const { return price_; }
  bool isDiscount() const { return isDiscount_; }
  const std::string& name() const { return name_; }

private:
  std::string name_;
  int price_;
  bool isDiscount_;
};

class Register {
public:
  explicit Register(std::vector<Item> items) : items_(std::move(items)) {}

  std::vector<Item> applyDiscount(int amount) const {
    std::vector<Item> discounted;
    discounted.reserve(items_.size());
    for (const auto& item : items_) {
      if (item.isDiscount()) {
        discounted.push_back(item.discounted(amount));
      }
```

```
    }
    return discounted;
  }

  int calculateTotal(const std::vector<Item>& items) const {
    return std::accumulate(
        items.begin(), items.end(), 0,
        [](int acc, const Item& item) { return acc + item.price(); });
  }

private:
  std::vector<Item> items_;
};

int main() {
  std::vector<Item> items{
      Item{"Onigiri", 150, true},
      Item{"Juice", 120, false},
      Item{"Sandwich", 300, true},
  };

  Register reg{items};
  auto discounted = reg.applyDiscount(10);
  int total = reg.calculateTotal(discounted);

  std::cout << "合計金額: " << total << "円" << std::endl;
}
```

Python で書いてみる

Python では `dataclass` を使うと、値オブジェクトを簡潔に表現できます。`Register` クラスが責務を担い、リスト操作や合計計算をメソッドとしてまとめています。

```
from dataclasses import dataclass
from typing import List

@dataclass(frozen=True)
class Item:
    name: str
    price: int
    is_discount: bool

    def discounted(self, amount: int) -> "Item":
        new_price = max(0, self.price - amount)
        return Item(self.name, new_price, self.is_discount)

class Register:
    def __init__(self, items: List[Item]) -> None:
        self._items = items

    def apply_discount(self, amount: int) -> List[Item]:
```

```
            return [
                item.discounted(amount)
                for item in self._items
                if item.is_discount
            ]

        def calculate_total(self, items: List[Item]) -> int:
            return sum(item.price for item in items)

    def main() -> None:
        items = [
            Item("Onigiri", 150, True),
            Item("Juice", 120, False),
            Item("Sandwich", 300, True),
        ]

        register = Register(items)
        discounted = register.apply_discount(10)
        total = register.calculate_total(discounted)
        print(f"合計金額: {total}円")

    if __name__ == "__main__":
        main()
```

Java で書いてみる

Java ではクラス定義とアクセサを明示的に書くことで、 オブジェクト指向の構造をわかりやすく表現できます。`Register` が責務を持ち、`Item` は値オブジェクトとして振る舞います。

```
    import java.util.ArrayList;
    import java.util.List;

    class Item {
      private final String name;
      private final int price;
      private final boolean isDiscount;

      Item(String name, int price, boolean isDiscount) {
        this.name = name;
        this.price = price;
        this.isDiscount = isDiscount;
      }

      Item discounted(int amount) {
        return new Item(name, price - amount, isDiscount);
      }

      int getPrice() {
        return price;
      }
```

```
    boolean isDiscount() {
      return isDiscount;
    }

    String getName() {
      return name;
    }
  }

  class Register {
    private final List<Item> items;

    Register(List<Item> items) {
      this.items = items;
    }

    List<Item> applyDiscount(int amount) {
      List<Item> discounted = new ArrayList<>();
      for (Item item : items) {
        if (item.isDiscount()) {
          discounted.add(item.discounted(amount));
        }
      }
      return discounted;
    }

    int calculateTotal(List<Item> itemsToSum) {
      int total = 0;
      for (Item item : itemsToSum) {
        total += item.getPrice();
      }
      return total;
    }
  }

  public class RegisterExample {
    public static void main(String[] args) {
      List<Item> items = List.of(
          new Item("Onigiri", 150, true),
          new Item("Juice", 120, false),
          new Item("Sandwich", 300, true)
      );

      Register register = new Register(items);
      List<Item> discounted = register.applyDiscount(10);
      int total = register.calculateTotal(discounted);
      System.out.println("合計金額: " + total + "円");
    }
  }
```

どの言語でも「商品を表すクラス」と「計算を担うクラス」を分離し、 責務を明確にするという設計は共通です。

問題 1

体重計算プログラムの改善

- 「人」を示すクラスを作成せよ。
- 「人」は個人を識別する名前を持ち、体重が記録されていく。
- 「人」がものを食べたとき、走ったときに体重が増減するようにメソッドを構成せよ。
- 各アクションを起こすたびに現在の体重が表示されるようにせよ。

参考: 関数型プログラミングで書くと（1）

```
let weight = 60;
let name = "Tanaka"; // 👈 名前を追加
function eat() {
  weight += 1;
  console.log(`${name}の体重: ${weight}kg`);
}
function run() {
  weight -= 2;
  console.log(`${name}の体重: ${weight}kg`);
}

eat(); // 実行結果: Tanakaの体重: 61kg
```

参考: 関数型プログラミングで書くと（2）

```
// 鈴木さんを登場させたい!
name = "Suzuki"; // 👈 😱 田中さんの名前が上書きされた!
weight = 70; // 👈 😱 田中さんの体重も上書きされた!

run(); // 実行結果: Suzukiの体重: 68kg
```

- 田中さんの情報はどこへ行った？
- ひとつの `weight` と `name` しかないので、複数の人を同時に管理できない。
- とはいえ `weight_2` とか出すのも…

オブジェクト指向へのヒント

1.

- その人の体重は `this.weight` で管理されている。
- つまり、 `this.weight` の値を変更すれば、体重が変更される。

2.

- メソッドはひとつだけじゃないとだめ、というルールはない。

解答例

```
class Person {
  // コンストラクタで名前と初期体重を設定
  constructor(name, initialWeight) {
    this.name = name;
    this.weight = initialWeight;
    console.log(`[${this.name}] が誕生！初期体重: ${this.weight}kg`);
  }

  // 食べるメソッド
  eat() {
    this.weight += 1;
    console.log(`[${this.name}] が食べた。現在の体重: ${this.weight}kg`);
  }

  // 走るメソッド
  run() {
    this.weight -= 2;
    console.log(`[${this.name}] が走った。現在の体重: ${this.weight}kg`);
  }
}
```

期待する動作例

```
const tanaka = new Person("Tanaka", 60);
const suzuki = new Person("Suzuki", 75);

tanaka.eat();
suzuki.run();
tanaka.run();
```

```
[Tanaka] が食べた。現在の体重: 61kg
[Suzuki] が走った。現在の体重: 73kg
[Tanaka] が走った。現在の体重: 60kg
```

問題 2

営業成績プログラムの改善

- 「**営業担当者**」を示すクラスを作成せよ。
- 「**営業担当者**」は**名前**を持ち、現在の**売上数**を記録する。
- **売上を一件追加**するメソッドと、**トップ営業員か判定**するメソッドを構成せよ。
- トップ営業員とは、売上数が 5 件を超えている者とする。
- アクション（売上追加）を起こすたびに、現在の**売上数**を表示せよ。

期待する動作例

売上が表示され、最終的にトップ営業員かどうかを判断する

```
[Yamada]売上追加！現在の売上： 1件
[Yamada]売上追加！現在の売上： 2件
[Sato]売上追加！現在の売上： 1件
// 省略

Yamadaはトップ営業員です！
Satoはまだトップではありません。
```

参考: 関数型プログラミングで書くと（1）

```
let salespersonName = "Yamada";
let salesCount = 4; // 👈 担当者とデータが別々
let targetSales = 5;

function addSale() {
  salesCount += 1;
  console.log(`${salespersonName}の売上: ${salesCount}件`);
}

function isTopSalesperson(count) {
  return count > targetSales; // 👈 判定ロジックも独立
}

addSale(); // Yamadaの売上: 5件
console.log(isTopSalesperson(salesCount)); // false
```

参考: 関数型プログラミングで書くと（2）

- 別の担当者「Sato」さんの処理を追加したい場合、salesCount や salespersonName を都度、手動で切り替える必要がある...
- 関数 isTopSalesperson には、誰の salesCount を渡したか**常に意識しないといけない**！

オブジェクト指向へのヒント

- `this.sales` が**インスタンス自身の売上数**を指します。
  `this` を使って、メソッド内でその売上数 (`this.sales`) を操作できます。
- `this.〜` の初期化は、必ずしも引数でなくて良い。

解答例

```
class Salesperson {
  // 名前と売上(初期値は 0)を設定
  constructor(name) {
    this.name = name;
    this.sales = 0;
    console.log(`[${this.name}] が担当開始。初期売上: ${this.sales}件`);
  }

  // 売上を一件追加するメソッド
  addSale() {
    this.sales += 1;
    console.log(`[${this.name}] 売上追加！現在の売上: ${this.sales}件`);
  }

  // トップ営業員か判定するメソッド
  isTop() {
    if (this.sales > 5) {
      console.log(`${yamada.name}はトップ営業員です！`);
    } else {
      console.log(`${yamada.name}はまだトップではありません。`);
    }
  }
}
```

期待する動作例

```
const yamada = new Salesperson("Yamada");
const sato = new Salesperson("Sato");

yamada.addSale();
yamada.addSale();
sato.addSale();

// 省略

yamada.isTop();
sato.isTop();
```

問題 3

チームメンバーリストの操作

- **チーム**を示すクラスを作成せよ。
- **チーム**は「**メンバーリスト**」という配列を持っている。
- メンバーを**追加**するメソッドと、**全員の名前を一覧表示**するメソッドを構成せよ。
- メンバー一覧表示は `for` を使って行うこと。

期待する動作例

```
[Alpha] チーム発足！
[Beta] チーム発足！
TanakaをAlphaチームに追加しました。
YamadaをAlphaチームに追加しました。
SatoをBetaチームに追加しました。

--- [Alpha] メンバー一覧 ---
メンバー: Tanaka
メンバー: Yamada
--- [Alpha] メンバー一覧 ---
メンバー: Sato
```

参考: 関数型プログラミングで書くと（1）

```
let teamAMembers = ["田中", "山田"];
let teamBMembers = ["佐藤"];

// メンバーを追加する汎用関数
function addMember(teamList, name) {
  teamList.push(name); // 👈 どのリストに?を引数で指定
}

// メンバーを一覧表示する汎用関数
function listMembers(teamList) {
  for (let i = 0; i < teamList.length; i++) {
    console.log(`メンバー: ${teamList[i]}`);
  }
}

addMember(teamAMembers, "鈴木");
listMembers(teamAMembers);
```

参考: 関数型プログラミングで書くと（2）

- メンバーリストと、操作がバラバラに管理されている！

オブジェクト指向へのヒント

1.

- `this.members` が**そのチームが持つ**メンバーの配列です。
- メソッド内では、`this.members.push()` のように、自分の配列に対して操作ができます。

2.

- コンストラクタで `this.members` を空の配列で初期化しておくと、後からすぐに追加できます。

解答例

```
class Team {
  // コンストラクタでメンバーリストを空の配列として初期化
  constructor(teamName) {
    this.teamName = teamName;
    this.members = []; //  チーム自身がメンバーリストを持つ
    console.log(`[${this.teamName}] チーム発足！`);
  }

  // メンバーを追加するメソッド
  addMember(name) {
    this.members.push(name);
    console.log(`${name}を${this.teamName}チームに追加しました。`);
  }

  // メンバーを一覧表示するメソッド
  listMembers() {
    console.log(`--- [${this.teamName}] メンバー一覧 ---`);
    for (let i = 0; i < this.members.length; i++) {
      console.log(`メンバー: ${this.members[i]}`);
    }
  }
}
```

```
const teamA = new Team("Alpha");
const teamB = new Team("Beta");

teamA.addMember("Tanaka");
teamA.addMember("Yamada");

teamB.addMember("Sato");

teamA.listMembers(); // 👈 Alpha チームのリストだけが表示される
teamB.listMembers(); // 👈 Beta チームのリストだけが表示される
```

演習問題 5 問

終わったら自主学習してててね

共通問題文

次の関数型プログラミングで書かれたコードは、
このまま関数型で書くと冗長になってしまう構造のため、
対策を考えたい。

オブジェクト指向を使ってこれを対策したいと考えたとき、
クラスをどのように設計し、メソッドを組めば適切か？

期待される動作にそって動作するように
オブジェクト指向を使いながら
コードを再設計してください。

1問目

現状 1

```
const orders = [
  { customer: "佐藤", drink: "抹茶ラテ", temperature: "hot" },
  { customer: "鈴木", drink: "カフェラテ", temperature: "ice" },
];

function makeMatchaLatte(order) {
  return [
    `${order.customer}さんへの提供準備`,
    "ベース: 抹茶",
    `温度: ${order.temperature}`,
    "仕上げ: スチームミルク",
  ].join(" / ");
}
```

現状 2

```
function makeCafeLatte(order) {
  return [
    `${order.customer}さんへの提供準備`,
    "ベース: エスプレッソ",
    `温度: ${order.temperature}`,
    "仕上げ: スチームミルク",
  ].join(" / ");
}

function serve(order) {
  if (order.drink === "抹茶ラテ") return makeMatchaLatte(order);
  if (order.drink === "カフェラテ") return makeCafeLatte(order);
  return `${order.drink}は未対応`;
}
```

現状使用例

```
orders.forEach((order) => console.log(serve(order)));
```

期待される動作例

```
佐藤さんへの提供準備 / ベース: 抹茶 / 温度: hot / 仕上げ: スチームミルク
鈴木さんへの提供準備 / ベース: エスプレッソ / 温度: ice / 仕上げ: スチームミルク
```

ヒント

- クラスは 3 つ作られる、「お客さん」「ドリンクメニュー」「提供側」
- それぞれにどんなデータが現状を見て読み取れるか？ (書いてないものは省略して良い、価格とか)
- 現状だとデータにかかわらず固定になっているものがないか？

2 問目

現状 1

```
const books = [
  { title: "JavaScript入門", borrower: null },
  { title: "Python基礎", borrower: "田中" },
];

function borrowBook(book, name) {
  if (book.borrower) {
    return `${book.title}は${book.borrower}さんが借りています`;
  }
  return `${name}さんが${book.title}を借りました`;
}
```

現状 2

```
function returnBook(book) {
  if (!book.borrower) {
    return `${book.title}は貸し出されていません`;
  }
  const borrower = book.borrower;
  return `${borrower}さんが${book.title}を返却しました`;
}

function getStatus(book) {
  if (book.borrower) {
    return `${book.title}: 貸出中(${book.borrower}さん)`;
  }
  return `${book.title}: 貸出可能`;
}
```

現状使用例

```
console.log(borrowBook(books[0], "佐藤"));
console.log(getStatus(books[1]));
```

期待される動作例

```
佐藤さんがJavaScript入門を借りました
Python基礎： 貸出中(田中さん)
```

ヒント

- クラスは 2 つ、「本」と「図書館システム」
- 本の状態（貸出中かどうか）をどう管理するか？
- 貸出・返却の処理は本自身が持つべきか、システムが持つべきか？

3 問目

現状 1

```
const products = [
  { name: "りんご", stock: 10, price: 100 },
  { name: "みかん", stock: 5, price: 80 },
];

function addStock(product, amount) {
  return `${product.name}の在庫を${amount}個追加しました(合計: ${
    product.stock + amount
  }個)`;
}
```

現状 2

```
function removeStock(product, amount) {
  if (product.stock < amount) {
    return `${product.name}の在庫が不足しています(在庫: ${product.stock}個)`;
  }
  return `${product.name}を${amount}個販売しました(残り: ${
    product.stock - amount
  }個)`;
}

function checkStock(product) {
  if (product.stock === 0) return `${product.name}: 在庫なし`;
  if (product.stock < 5) return `${product.name}: 在庫少(${product.stock}個)`;
  return `${product.name}: 在庫あり(${product.stock}個)`;
}
```

現状使用例

```
console.log(addStock(products[0], 5));
console.log(removeStock(products[1], 3));
```

期待される動作例

```
りんごの在庫を5個追加しました(合計： 15個)
みかんを3個販売しました(残り： 2個)
```

ヒント

- クラスは 2 つ、「商品」と「在庫管理」
- 在庫の増減は商品自身のメソッドとして持つべき
- 在庫チェックのロジックはどこに置くか？

4 問目

現状 1

```
const tasks = [
  { name: "資料作成", status: "未着手", priority: "高" },
  { name: "メール返信", status: "進行中", priority: "中" },
];

function startTask(task) {
  if (task.status !== "未着手") {
    return `${task.name}は既に${task.status}です`;
  }
  return `${task.name}を開始しました`;
}
```

現状 2

```
function completeTask(task) {
  if (task.status === "完了") {
    return `${task.name}は既に完了しています`;
  }
  return `${task.name}を完了しました`;
}

function getTaskInfo(task) {
  return `[${task.priority}] ${task.name}: ${task.status}`;
}
```

現状使用例

```
console.log(startTask(tasks[0]));
console.log(getTaskInfo(tasks[1]));
```

期待される動作例

```
資料作成を開始しました
[中] メール返信: 進行中
```

ヒント

- クラスは 2 つ、「タスク」と「タスク管理」
- タスクの状態遷移（未着手→進行中→完了）をメソッドで表現
- ステータスの検証はどこで行うべきか？

5 問目

現状 1

```
const accounts = [
  { owner: "山田", balance: 10000 },
  { owner: "佐々木", balance: 5000 },
];

function deposit(account, amount) {
  if (amount <= 0) {
    return `金額が不正です`;
  }
  return `${account.owner}さんの口座に${amount}円入金しました(残高: ${
    account.balance + amount
  }円)`;
}
```

現状 2

```
function withdraw(account, amount) {
  if (amount <= 0) {
    return `金額が不正です`;
  }
  if (account.balance < amount) {
    return `${account.owner}さんの残高が不足しています(残高: ${account.balance}円)`;
  }
  return `${account.owner}さんの口座から${amount}円出金しました(残高: ${
    account.balance - amount
  }円)`;
}

function getBalance(account) {
  return `${account.owner}さんの残高: ${account.balance}円`;
}
```

現状使用例

```
console.log(deposit(accounts[0], 3000));
console.log(withdraw(accounts[1], 2000));
```

期待される動作例

```
山田さんの口座に3000円入金しました(残高: 13000円)
佐々木さんの口座から2000円出金しました(残高: 3000円)
```

ヒント

- クラスは 2 つ、「口座」と「銀行」
- 入出金の処理は口座自身が持つべき
- 残高不足や金額チェックはどこで行うべきか？

演習問題 解答例

1 問目 解答例

クラス設計 1

```
class Drink {
  constructor(name, base, topping) {
    this.name = name;
    this.base = base;
    this.topping = topping;
  }
}
```

クラス設計 2

```
class Order {
  constructor(customer, drink, temperature) {
    this.customer = customer;
    this.drink = drink;
    this.temperature = temperature;
  }

  serve() {
    return [
      `${this.customer}さんへの提供準備`,
      `ベース: ${this.drink.base}`,
      `温度: ${this.temperature}`,
      `仕上げ: ${this.drink.topping}`,
    ].join(" / ");
  }
}
```

使用例と動作

```
const matchaLatte = new Drink("抹茶ラテ", "抹茶", "スチームミルク");
const cafeLatte = new Drink("カフェラテ", "エスプレッソ", "スチームミルク");

const order1 = new Order("佐藤", matchaLatte, "hot");
const order2 = new Order("鈴木", cafeLatte, "ice");

console.log(order1.serve());
console.log(order2.serve());
```

出力結果

```
佐藤さんへの提供準備 / ベース: 抹茶 / 温度: hot / 仕上げ: スチームミルク
鈴木さんへの提供準備 / ベース: エスプレッソ / 温度: ice / 仕上げ: スチームミルク
```

ポイント

- `Drink` クラスでドリンクの種類ごとの固定データ（ベース、トッピング）を管理
- `Order` クラスで注文ごとの可変データ（お客さん名、温度）を管理
- ドリンクごとに関数を増やす必要がなくなった
- 新しいドリンクを追加する場合は `new Drink()` で簡単に追加可能

2 問目 解答例

クラス設計

```
class Book {
  constructor(title) {
    this.title = title;
    this.borrower = null;
  }

  borrow(name) {
    if (this.borrower) {
      return `${this.title}は${this.borrower}さんが借りています`;
    }
    this.borrower = name;
    return `${name}さんが${this.title}を借りました`;
  }

  // つづく
}
```

クラス設計

```
class Book {
  // つづき
  bookReturn() {
    if (!this.borrower) {
      return `${this.title}は貸し出されていません`;
    }
    const borrower = this.borrower;
    this.borrower = null;
    return `${borrower}さんが${this.title}を返却しました`;
  }

  getStatus() {
    if (this.borrower) {
      return `${this.title}: 貸出中(${this.borrower}さん)`;
    }
    return `${this.title}: 貸出可能`;
  }
}
```

使用例

```
const book1 = new Book("JavaScript入門");
const book2 = new Book("Python基礎");

// 初期状態を設定
book2.borrower = "田中";

console.log(book1.borrow("佐藤"));
console.log(book2.getStatus());
console.log(book2.bookReturn());
console.log(book2.getStatus());
```

出力結果

```
佐藤さんがJavaScript入門を借りました
Python基礎： 貸出中(田中さん)
田中さんがPython基礎を返却しました
Python基礎： 貸出可能
```

ポイント

- `Book` クラスが自分の状態（貸出中かどうか）を管理
- 貸出・返却の処理を本自身のメソッドとして実装
- 本ごとに関数を書く必要がなくなった
- 状態の変更は本が自分で行うため、データの一貫性が保たれる

3 問目 解答例

クラス設計

```
class Product {
  constructor(name, stock, price) {
    this.name = name;
    this.stock = stock;
    this.price = price;
  }

  addStock(amount) {
    this.stock += amount;
    return `${this.name}の在庫を${amount}個追加しました(合計: ${this.stock}個)`;
  }

  removeStock(amount) {
    if (this.stock < amount) {
      return `${this.name}の在庫が不足しています(在庫: ${this.stock}個)`;
    }
    this.stock -= amount;
    return `${this.name}を${amount}個販売しました(残り: ${this.stock}個)`;
  }

  checkStock() {
    if (this.stock === 0) return `${this.name}: 在庫なし`;
    if (this.stock < 5) return `${this.name}: 在庫少(${this.stock}個)`;
    return `${this.name}: 在庫あり(${this.stock}個)`;
  }
}
```

使用例と動作

```
const apple = new Product("りんご", 10, 100);
const orange = new Product("みかん", 5, 80);

console.log(apple.addStock(5));
console.log(orange.removeStock(3));
console.log(apple.checkStock());
console.log(orange.checkStock());
```

出力結果

```
りんごの在庫を5個追加しました(合計： 15個)
みかんを3個販売しました(残り： 2個)
りんご： 在庫あり(15個)
みかん： 在庫少(2個)
```

ポイント

- `Product` クラスが自分の在庫を管理
- 在庫の増減処理を商品自身のメソッドとして実装
- 商品ごとに関数を書く必要がなくなった
- 在庫数の変更は商品が自分で行うため、データの整合性が保たれる

4 問目 解答例

クラス設計

```
class Task {
  constructor(name, priority) {
    this.name = name;
    this.status = "未着手";
    this.priority = priority;
  }

  start() {
    if (this.status !== "未着手") {
      return `${this.name}は既に${this.status}です`;
    }
    this.status = "進行中";
    return `${this.name}を開始しました`;
  }
  // つづく
}
```

クラス設計

```
class Task {
  // つづき
  complete() {
    if (this.status === "完了") {
      return `${this.name}は既に完了しています`;
    }
    this.status = "完了";
    return `${this.name}を完了しました`;
  }

  getInfo() {
    return `[${this.priority}] ${this.name}: ${this.status}`;
  }
}
```

使用例

```
const task1 = new Task("資料作成", "高");
const task2 = new Task("メール返信", "中");

// task2を進行中に設定
task2.status = "進行中";

console.log(task1.start());
console.log(task2.getInfo());
console.log(task2.complete());
console.log(task2.getInfo());
```

出力結果

```
資料作成を開始しました
[中] メール返信： 進行中
メール返信を完了しました
[中] メール返信： 完了
```

ポイント

- `Task` クラスが自分の状態を管理
- 状態遷移（未着手 → 進行中 → 完了）をメソッドで表現
- タスクごとに関数を書く必要がなくなった
- 状態の検証をメソッド内で行うため、不正な状態遷移を防げる

5 問目 解答例

クラス設計

```
class Account {
  constructor(owner, balance) {
    this.owner = owner;
    this.balance = balance;
  }

  deposit(amount) {
    if (amount <= 0) {
      return `金額が不正です`;
    }
    this.balance += amount;
    return `${this.owner}さんの口座に${amount}円入金しました(残高: ${this.balance}円)`;
  }
  // つづく
}
```

クラス設計

```
class Account {
  withdraw(amount) {
    if (amount <= 0) {
      return `金額が不正です`;
    }
    if (this.balance < amount) {
      return `${this.owner}さんの残高が不足しています(残高: ${this.balance}円)`;
    }
    this.balance -= amount;
    return `${this.owner}さんの口座から${amount}円出金しました(残高: ${this.balance}円)`;
  }

  getBalance() {
    return `${this.owner}さんの残高: ${this.balance}円`;
  }
}
```

使用例

```
const account1 = new Account("山田", 10000);
const account2 = new Account("佐々木", 5000);

console.log(account1.deposit(3000));
console.log(account2.withdraw(2000));
console.log(account1.getBalance());
console.log(account2.getBalance());
```

出力結果

```
山田さんの口座に3000円入金しました(残高： 13000円)
佐々木さんの口座から2000円出金しました(残高： 3000円)
山田さんの残高： 13000円
佐々木さんの残高： 3000円
```

ポイント

- `Account` クラスが自分の残高を管理
- 入出金の処理を口座自身のメソッドとして実装
- 口座ごとに関数を書く必要がなくなった
- 残高の変更は口座が自分で行うため、データの整合性が保たれる

まとめ

オブジェクト指向の利点

- **データとそれを操作する処理をまとめられる**
  - 関連する情報と処理が一箇所にまとまる
- **同じような処理を何度も書かなくて済む**
  - クラスのインスタンスを作るだけで新しいオブジェクトを作れる
- **データの整合性が保たれる**
  - オブジェクト自身が自分の状態を管理する
- **拡張しやすい**
  - 新しい種類を追加する場合も既存のコードを変更しなくて良い

お疲れ様でした！

オブジェクト指向集中講座

自由課題説明

自由課題

- これまでに学んだオブジェクト指向の知識を活かし、作品を作ってみてください。
- ゲーム、アプリ、形式は自由です。
- 全員で作品の鑑賞会を行い、可能ならコードを閲覧しながら講評を行います。

期限

- **10 月 10 日 集中講座コマ 前半終了時まで**
- 上記期限までに Google Classroom の課題提出フォルダに提出すること。

条件と制約

- 言語は自由、あなたの得意な分野で戦ってください。
- Google ドライブにアップロードできる形にすること、つまり**なんでもいいから閲覧できる状態にする**こと。
-
- 鑑賞会において、全員が鑑賞できる状態、つまり講師の Mac で操作できる状態が望ましい。(望ましいだけであって必須ではない、Windows でしか動かないもヨシ！)
- どうしても～というときは動画でも可とする
- 全員で鑑賞するので、公序良俗に反する内容は控えてください。

可能なら

- 僕が C 系ほぼ触ったことないので、C 系を採用するならコメント多めだと助かります！！気合で読みます
- Python、JS、TS あたりなら読めます

Gemini を活用

- わからなかったら Gemini などに聞いてみること。
- カスタムモデルはデフォルトだと JS を吐き出すので言語指定したほうが良いです。
- それでもわからなかったら講師や先生を読んでみるのもあり。
- 期間が短いので使えるものは何でも使ってください。

自由課題のコツ

- 紙でもパソコンでも、とにかくアイデアを書き出してみる
- どんなアイデアが光るかわからないのでとにかく出しまくる
- 面白そうなものがあれば手を出してみる、AI が伴走してくれるので難易度は気にしない
- オブジェクト指向だからできること、にとらわれずに、ガーッと書いて「ここオブジェクトで置き換えたほうが良さげ」ってところを書き直す形でも良い

あなたの面白い作品を期待しています！