歴史から学ぶオブジェクト指向

プログラミング言語 #9

機械語

10+20の算術演算をしてみる

- 結局コンピュータは2進数しか理解できない
- 2進数（16進数）で表現される命令で書かれたプログラム
- 生産性低い
- わけわからん
- プログラマすごい
- CPUごとに命令違う

```
06000A0D0014
```

アセンブラ

https://schweigi.github.io/assembler-simulator/

10+20の算術演算をしてみる

- 機械語と１対１対応
- 機械語よりもわかりやすい
- ちょっと生産性高い
- ちょっと間違えたら暴走
- CPUごとに命令違う

```
MOV A, 10
ADD A, 20
```

高級言語

10+20の算術演算をしてみる

- 高級言語
- 人間にとってわかりやすい
- （アセンブラより）遅い
- 生産性高い

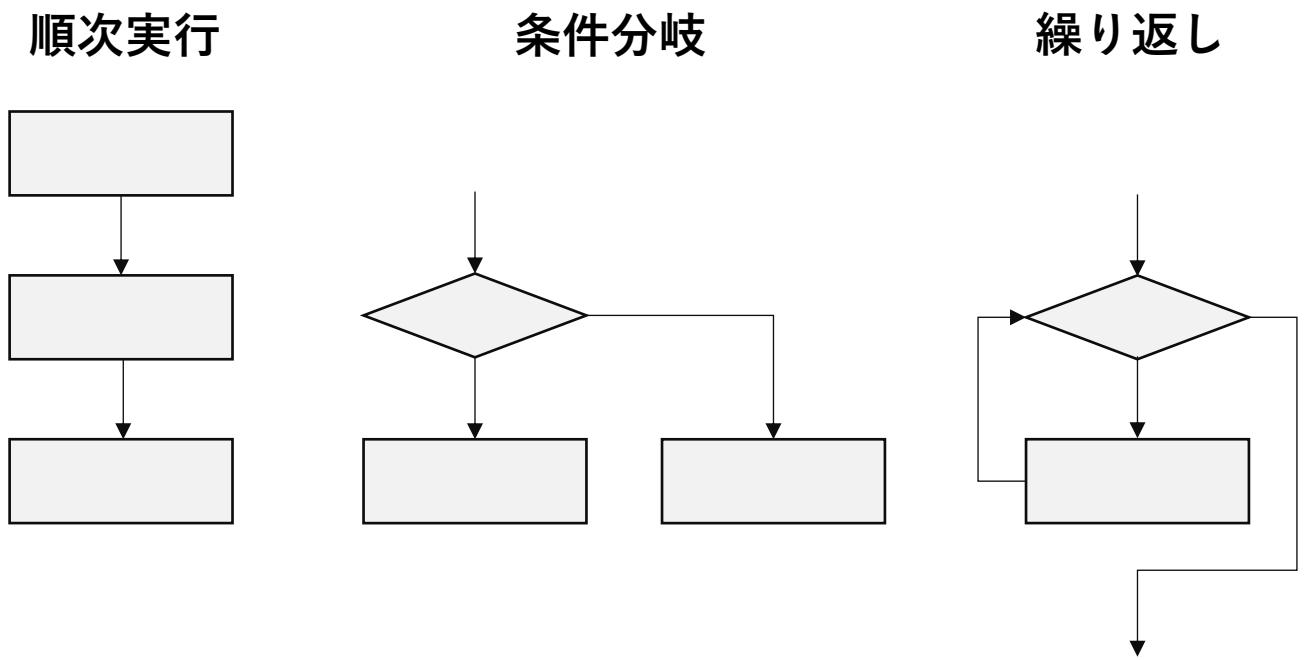
```
10+20
```

プログラムが高度化してくる

世界の全員がプログラマになっても増大するソフトウェア需要に追いつかない

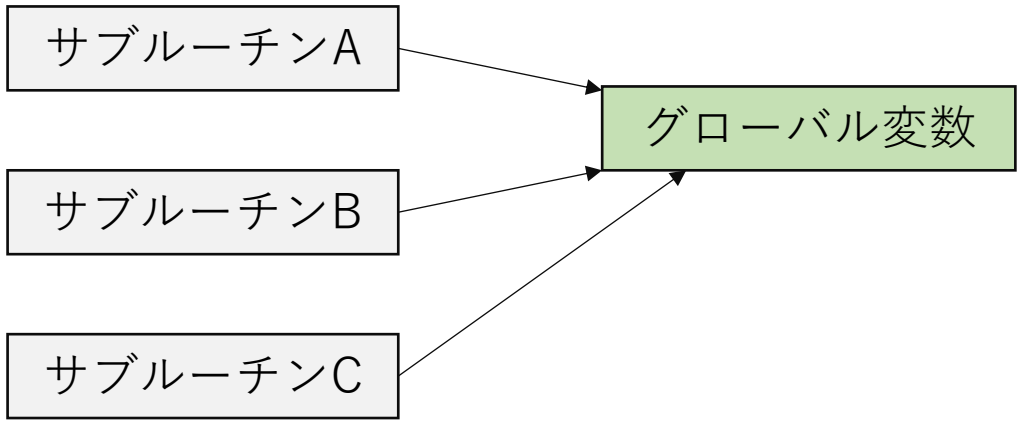
構造化プログラミング

- ダイクストラにより提唱
  - ダイクストラ法で有名
- GOTO構文によるスパゲッティコードを回避
- 正しく動作するプログラムには**構造が必要**
- 3つの構造
  - 順次実行
  - 条件分岐
  - 繰り返し



サブルーチンの問題

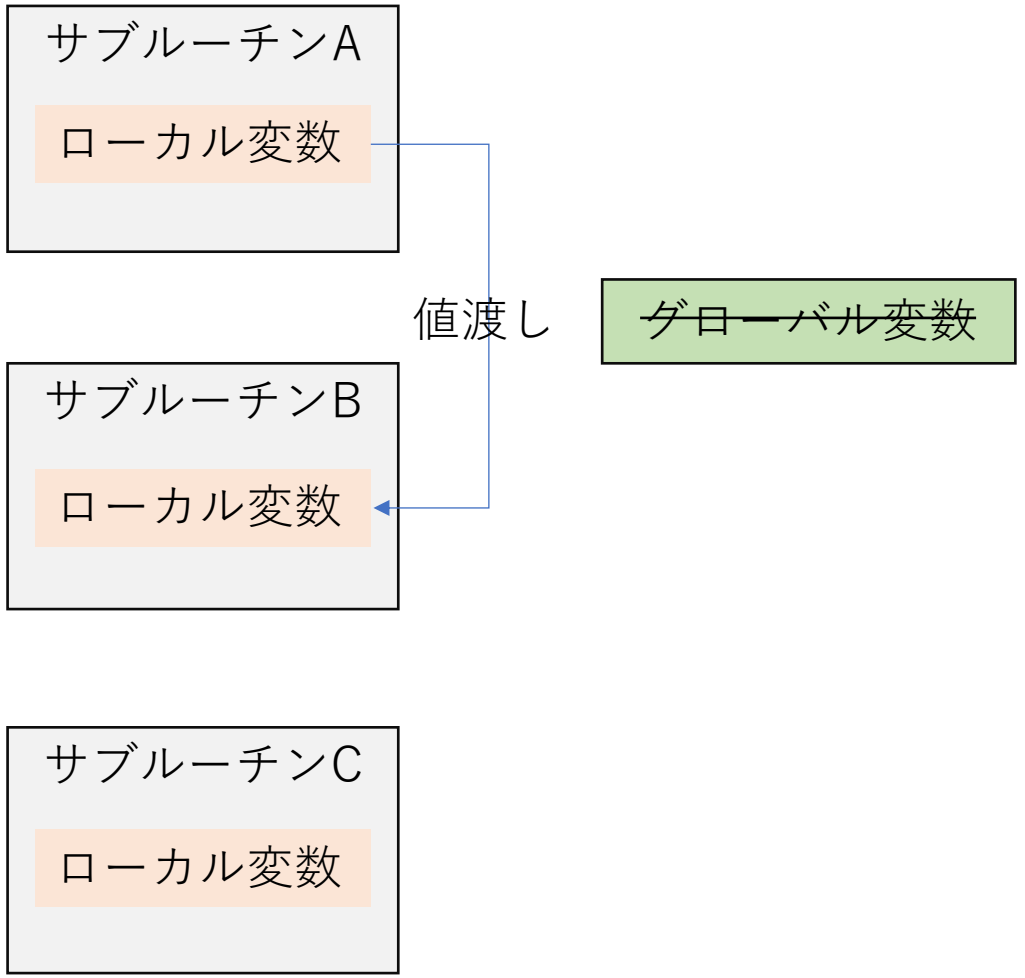
- サブルーチン（関数）間でのデータの受け渡し
  - グローバル変数を用いる
- グローバル変数の増加
  - 複雑性の増加、保守性の低下
- 保守性の向上
  - グローバル変数を減らす



サブルーチンの独立性向上

- サブルーチン（関数）間でのデータの受け渡し
  - 引数による**値渡し**

- **ローカル変数**
  - サブルーチンの独立性向上



構造化プログラミング言語の登場

- Algol
- C言語
- Pascal

- Knuthが論争をまとめる
  - Structured Programming with Goto Statements
  - 構造化プログラミングとは、書きやすく、理解しやすいプログラミングのことである。
  - 構造化プログラミングは、goto文をなるべく使用せずにプログラミングすることである。goto文の使用はエラー処理に限定し、かつ「前方goto文」に限る

- 制御構造を明確に記述できる
- 構造化プログラミングの３要素をサポート

構造化プログラミングの限界

グローバル変数問題、プログラムの再利用の問題

プログラミング言語に求められること

高い生産性
- 高級言語の登場

高い保守性
- 構造化言語における基本3構造

品質の向上
- 複雑さの排除→GOTOの排除など

再利用の促進
- サブルーチンの独立性
- GUIなどの登場によりこれがもっと求められるように…

オブジェクト指向

カプセル化、ポリモーフィズム、継承の3概念

Simula67（初期のオブジェクト指向言語）

- ノルウェーのDahlとNygaardにより開発された
  - Algolのスーパーセットとして
- クラス、オブジェクト、継承などのオブジェクト指向の基本的概念を全て持っている
- Alan KayのSmalltalkなどに影響を与えた

- グローバル変数を使わないでいける仕組みがある
- 共通のサブルーチンだけではなくプログラムの再利用ができるような仕組みがある

オブジェクト指向の3つの概念

カプセル化

- 整理整頓をする役割
- サブルーチンと変数をひとつのまとまりとして扱う

ポリモーフィズム

- メソッドを呼び出す側を共通化して無駄を省く
- 共通のメインルーチンを作れるようになる

継承

- 重複するクラスの定義を共通化して無駄を省く
- クラスの共通部分を別のクラス（スーパークラス）にまとめる

カプセル化（クラス）

- メソッドと変数をまとめる（カプセルにする）
- クラスの内部だけで使う変数やメソッドを隠す

構造化プログラミングによるファイル操作

以下の操作ができる

- ファイルを開く
- ファイルを閉じる
- ファイルから1文字を読む

```
file = None


def openFile(filename):
        file = …
def closeFile():
        …
def readFile():
        …
```

クラスによるファイル操作

以下の操作ができる

- ファイルを開く
- ファイルを閉じる
- ファイルから1文字を読む

```
class FileReader():
        def __init__(self):
                self.file…
        def open(self, filename):
                …
        def close(self):
                …
        def read(self):
                …
```

クラスのいいところ

- 結びつきがつよい関数（メソッド）とグローバル変数を1つのクラスとしてまとめることができる
  - クラス名が重複しなければクラス内のメソッド名や変数名は気にしないでOK

クラスの機能（隠蔽）

ほんとはオブジェクト指向型言語では、クラスの外からのアクセスに対して情報を隠せる

Pythonはわりとゆるい

```
public class FileReader{
    private string file;
    public void open(String filename){ … }
    public void close(){ … }
    public char read(){ … }
}
```

クラスの機能（隠蔽）

ほんとはオブジェクト指向型言語では、クラスの外からのアクセスに対して情報を隠せる

Pythonはわりとゆるい

右の例だと

._FileReader__fileでアクセスできる

```
class FileReader():
        def __init__(self):
                self.__file…
        def open(self, filename):
                …
        def close(self):
                …
        def read(self):
                …
```

クラスの機能（Pythonでの隠蔽）

Pythonはわりとゆるい

コード規則によってインスタンス変数へのアクセス方法を規定

→インスタンス変数にアクセスするときはgetインスタンス変数名()を使いましょうねー

```
class FileReader():
        def __init__(self):
                self. file…
        def open(self, filename):
                …
        def close(self):
                …
        def read(self):
                …
        def getFile(self):
                return self.file
```

クラスのいいところ

- 結びつきがつよい関数（メソッド）とグローバル変数を1つのクラスとしてまとめることができる
  - クラス名が重複しなければクラス内のメソッド名や変数名は気にしないでOK
- クラスの変数やメソッドを隠すことができる
  - アクセスメソッドを使って変数にアクセス（getter/setter）
  - これによって、クラスの内部構造を変えても、呼び出し側には影響しない＝保守性が向上する

構造化プログラミングによるファイル操作

以下の操作ができる

- ファイルを開く
- ファイルを閉じる
- ファイルから1文字を読む

```
file = None


def openFile(filename):
        file = …
def closeFile():
        …
def readFile():
        …
```

クラスによるファイル操作

以下の操作ができる

- ファイルを開く
- ファイルを閉じる
- ファイルから1文字を読む

```
class FileReader():
        def __init__(self):
                self.file…
        def open(self, filename):
                …
        def close(self):
                …
        def read(self):
                …
```

構造化プログラミングでは…

- たくさん複製することが難しい
  - どんどん名前の異なる変数を定義しないと…

オブジェクト指向プログラミングでは…

- たくさん複製することが簡単
  - インスタンスを増やせば良い

クラスのいいところ

- 結びつきがつよい関数（メソッド）とグローバル変数を1つのクラスとしてまとめることができる
  - クラス名が重複しなければクラス内のメソッド名や変数名は気にしないでOK
- クラスの変数やメソッドを隠すことができる
  - アクセスメソッドを使って変数にアクセス（getter/setter）
  - クラスの内部構造を変えても、呼び出し側には影響しない＝保守性が向上する
- クラスを定義すれば、インスタンスを複数作れる
  - 複数同時に扱うような処理も簡単で保守性高く書けるようになる

オブジェクト指向の3つの概念

カプセル化

- 整理整頓をする役割
- サブルーチンと変数をひとつのまとまりとして扱う

ポリモーフィズム

- メソッドを呼び出す側を共通化して無駄を省く
- 共通のメインルーチンを作れるようになる

継承

- 重複するクラスの定義を共通化して無駄を省く
- クラスの共通部分を別のクラス（スーパークラス）にまとめる

ポリモーフィズム

- 多態性

- 結局は…　呼び出す側を修正しなくても呼び出される側のクラスのインタフェースを統一する仕組み

Readerは共通のインタフェースを持つ

- TextReader
  - テキストファイルを扱う
- NetworkReader
  - ネットワークを扱う

open
close
read
は共通

呼び出し側は相手がTextReaderかNetworkReaderかを気にしなくていい。

オブジェクト指向の3つの概念

カプセル化

- 整理整頓をする役割
- サブルーチンと変数をひとつのまとまりとして扱う

ポリモーフィズム

- メソッドを呼び出す側を共通化して無駄を省く
- 共通のメインルーチンを作れるようになる

継承

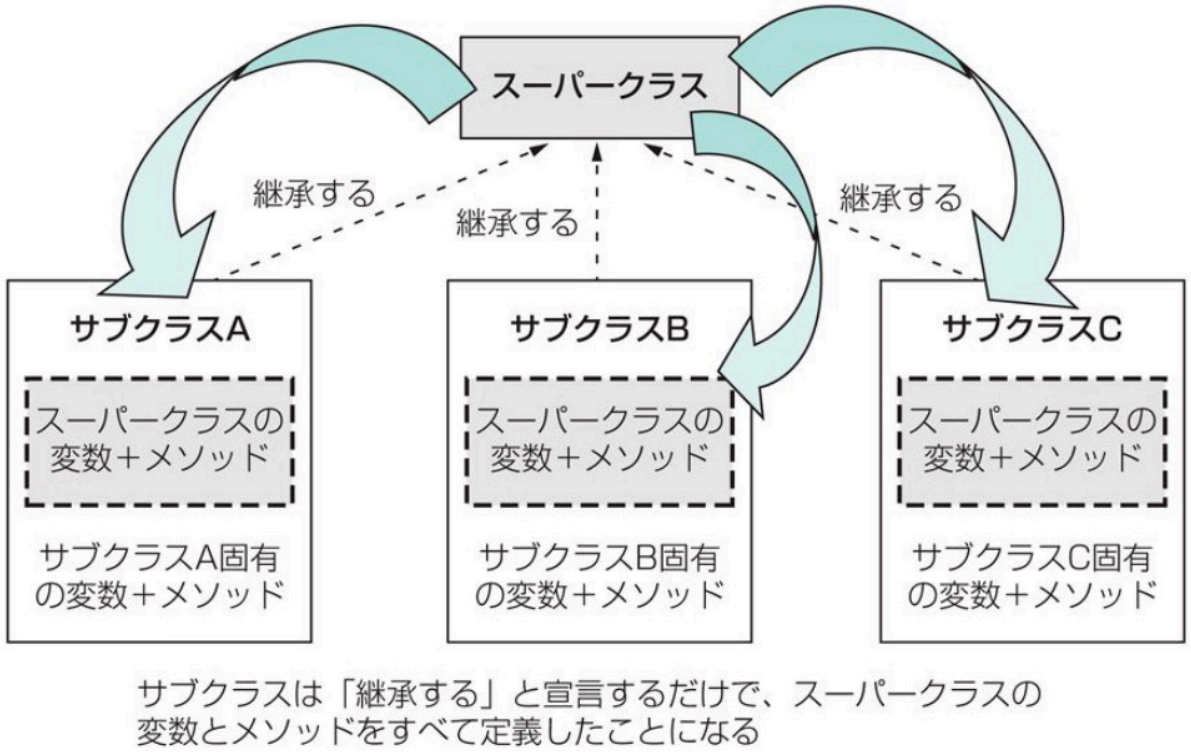
- 重複するクラスの定義を共通化して無駄を省く
- クラスの共通部分を別のクラス（スーパークラス）にまとめる

継承

同じようなクラスの共通部分と固有部分を整理する仕組み

- 共通部分をクラスとしてまとめてしまう

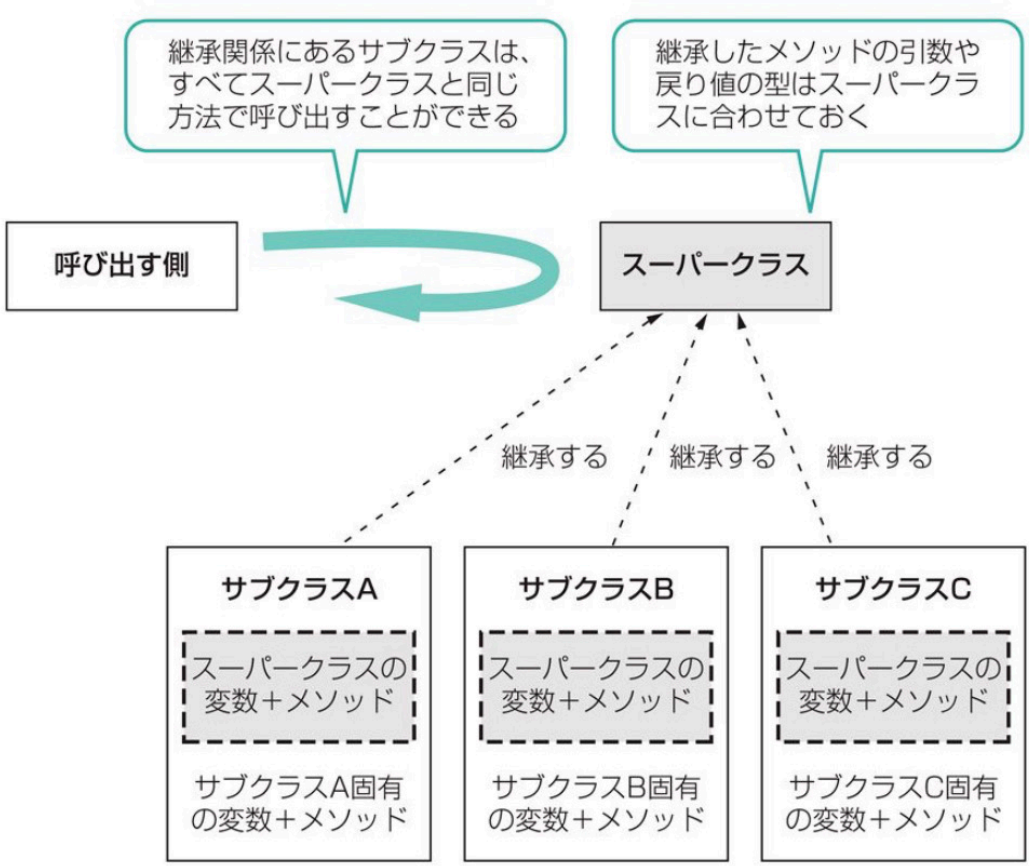
オブジェクト指向として考えることは、どういう共通化の概念を導入するか。
TextReaderとNetworkReaderだったらReaderだろうし、足し算と引き算だったら演算かもしれない



（引用：オブジェクト指向でなぜ作るか２版）

ポリモーフィズムと継承

- 継承によりポリモーフィズムが実現できる



（引用：オブジェクト指向でなぜ作るか２版）