

# オブジェクト指向プログラミング



ざっくりとイメージを伝えます。まあちょっと聞いてよ。

# オブジェクト指向とは

- オブジェクト指向と言っても以下のようにいくつかの目的があります。
  - 前期の講義で学ぶ部分は、赤字の部分です。
- **設計手法**としてのオブジェクト指向(OOD)
- **分析手法**としてのオブジェクト指向(OOA)
  - 2つを総称してオブジェクト指向分析設計(OODA)
- **プログラミング**としてのオブジェクト指向(OOP)
  - **プログラミング言語仕様**としてのオブジェクト指向
  - **開発方法論**としてのオブジェクト指向

ちなみにこれが一番  
大事なんです。

# プログラミングテクニックとしての オブジェクト指向

➤ 用語をつらつらと並べます。

- オブジェクト
- クラス
- インスタンス（実体）
- 状態と振る舞い
- インターフェイス
- 抽象化
- カプセル化（隠蔽）
- 継承
- 委譲

# オブジェクトとは

➤ ひとつのテーマを持ったデータと処理の集まり

➤ オブジェクトは、

➤ **状態（データ）** – プロパティや変数

➤ **振る舞い（処理）** – メソッド

で構成されている。

# クラスとは

## ➤ オブジェクトの設計書（定義）

- ただし、クラスベースのプログラミング言語に限る  
（クラスを持たないオブジェクト指向言語もあるということ）

# インスタンス（実体）とは

- オブジェクトが実際に使用できるよう  
コンピュータのメモリ空間上に領域を確保した状態
- 実体化、つまりメモリ上に必要なものを展開することで、  
はじめて処理が行える
- 実体化することをインスタンス化ともいう

# クラスとインスタンスの関係

- たとえ話をします。（あんまり好きじゃないけど...）
- クラス
  - 家の設計図
- インスタンス
  - 設計図通りに建てた家
- 設計図に直接住むことはできないけど、設計図通りに建てた家なら住める。というようなイメージです。

# 実体化は絶対に必要？

- 実体化は厳密に必要ではありません。
  - というより、明示的な実体化を宣言しなくても勝手に実体化されるものもあります。
- このような明示的な実体化が必要のないクラスやクラスのメンバやメソッドは静的(static)と呼ばれます。
  - (プログラム実行時に、内部的には勝手に実体化されている)
- 逆に、実体化が絶対に必要なものもあります。(こちらのほうが多い)
  - そういったものは動的(dynamic)と呼ばれます。



なんでオブジェクト指向なの？ ↗

# 処理をお任せする

- またたえ話です。  
子供に服を着替えさせることを想像してください。
- 子供に対して指示を出します。
  - 「上着のボタンを外して」「右腕から袖を通して次に左腕を通して脱いで」「シャツも右腕から.....」「かがんで靴下脱いで」「ズボン脱いで」「新しい服をダンスから出して」「シャツを首に通して...」

めんどい!!

# 処理をお任せする

- 子供に、「**服着替えなさい**」でいいじゃない。
- 処理手順を先にまとめておいて、その手順を行うことを指示することで、都度都度事細かな指示を行う負担を減らす
- 「処理を**お任せ**する」ということ

# プログラミングで考える

- 全部の処理をひとつの処理中に書くと、それぞれのデータに対して膨大な量の指示を書くことになる。
- 条件分岐などが発生すると、どんどん複雑化していく。
- 複雑な処理はバグを発生しやすくし、変更に多大な労力がかかる。
- じゃあ「データ」と「役割」ごとで処理を分割しようよ！
  - そうすれば複雑で煩雑な処理も負荷分散できるね！！

もうちょっと突っ込んでみる



# カプセル化（隠蔽）

- オブジェクトに処理をお任せした以上、あれやこれやと状態や処理に対して干渉しすぎるのはよくない
- オブジェクトが持つ状態や状態の変更処理を、オブジェクト自身に任せる
  - 外部から操作できないようにしよう！

# 継承

- 同じデータ、同じ処理を持つオブジェクトをそれぞれ作るの冗長だし、バグの発生原因になるかも...
  - 1つのプログラムの中でコピペを繰り返すと、修正するときに全部を直して確認しないといけないし大変。
- 似たデータ、処理を持つオブジェクトをクラスの時点でまとめよう！
  - このときに、役割単位できちんとまとめられると良い
- 継承はオブジェクトを分類してまとめる手段

オブジェクト指向プログラミングって  
万能なのか？？



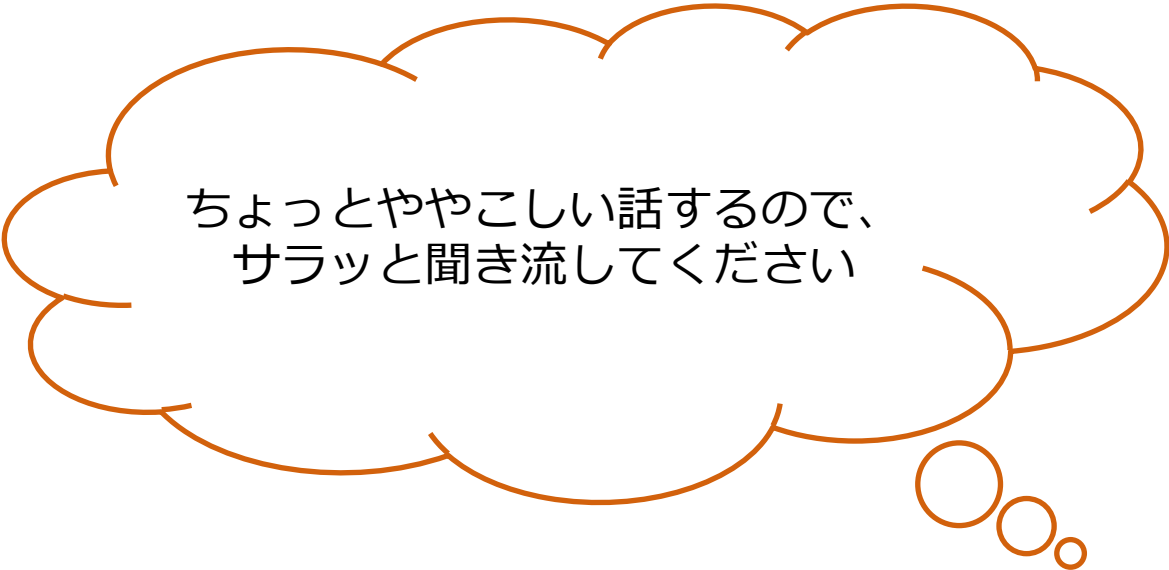


# そうでもないっす

- 現在の広く使われている考え方ですが、まだまだ発展途上で万能ではありません。
- オブジェクト指向を取り入れる場合は原則、設計ありきです。設計のない案件はかなり高い確率で破綻します。
- オブジェクト指向プログラミングを取り入れたことによる弊害もあります。
  - ごく小規模の案件などで、設計を疎かにして無理矢理オブジェクト指向を取り入れることによりコードの記述量が予定より膨大になってしまい複雑化を進行させてしまった。
  - 他には副作用などなど

# 万能なの教えてよ！と言わずに

- オブジェクト指向は、現在広く使われているということ
- 言語設計の時点でオブジェクト指向が取り入れられていること
  - その恩恵を実はすでに受けていること
- これらを踏まえてオブジェクト指向を学ぶことで、次の考え方の礎となります。



ちょっとややこしい話するので、  
サラッと聞き流してください

# オブジェクト指向の歴史的経緯



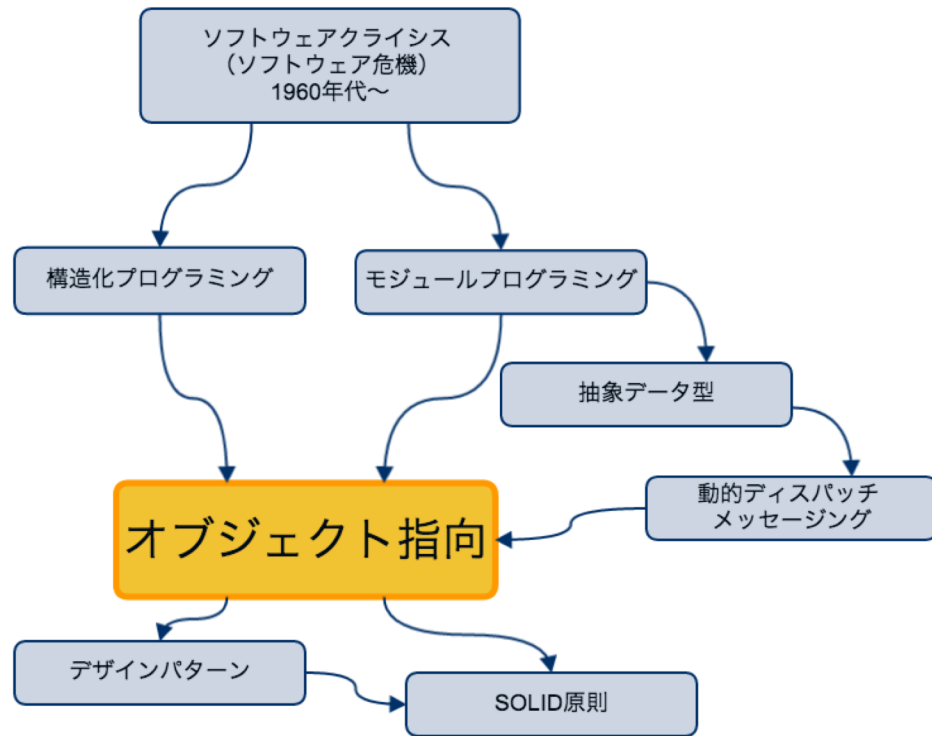
# プログラミングに限らない話をします

- オブジェクト指向と言っても以下のようにいくつかの目的があります。
- **設計手法としてのオブジェクト指向(OOD)**
- **分析手法としてのオブジェクト指向(OOA)**
  - 2つを総称してオブジェクト指向分析設計(OODA)
- **プログラミングとしてのオブジェクト指向(OOP)**
  - **プログラミング言語仕様としてのオブジェクト指向**
  - **開発方法論としてのオブジェクト指向**

# というわけで歴史的経緯をば

- 現在のオブジェクト指向は、いくつかの歴史的な経緯があり、それぞれのアイデアを統合し、再編されて(かつ現実的な制約により歪みを生じながら)生まれてきた考え方
- オブジェクト指向に限らず、どのプログラミングパラダイムにおいても、プログラミングという課題をどのように整理するか、という目的がある

# オブジェクト指向が生まれるまでを図で



図が小さくて申し訳ないです。  
詳細は配布資料よりPDFを  
ダウンロードして閲覧ください

# ソフトウェアクライシス

- 1960年代の後半ごろから、コンピュータの進化により複雑なソフトウェアが求められ始める。
  - このとき、コンピュータを動かすためには、コンピュータが理解できる0,1の情報のみを与えていました。
  - しかし、それはあまりにも難しく、うまくコントロールできる方法がなかった。
  - アセンブリなどの0,1で表される命令セットにアルファベットで名前を付けた言語もあるが、結局はコンピュータの専門的な知識が要求されてしまう。
- その状況を打破すべく生まれたのが、  
**「構造化プログラミング」**

# 構造化プログラミング

- エドガー・ダイクストラによって提案された技法。
- 機械と人間の間に、より人間の言葉に近いものを理解できる仮想機械を想定し、人間はそれに対してプログラミングをする。
  - 各仮想機械を階層として隔離して実装し、その変更は他の階層へ影響を与えない。
  - レイヤリングアーキテクチャのようなもの。
- 仮想機械について簡単に言うと、C言語のコンパイラがそれ



# モジュラプログラミング

- 構造化プログラミングが主流になると、もっと効率のよいプログラミング手法はないかと考えるようになる。  
そこで、プログラムのソースコードをモジュール化し、それぞれの機能ごとでまとめる手法を思いつく。その際のまとめる基準として、以下の3つに着目する。
- 凝集度
  - 良い機能のまとめ方と悪い機能のまとめ方を定義  
(悪：何を根拠に集めた機能かわからない機能群)
- 結合度
  - 悪い連携を定義しそれを避ける
    - 悪①：各機能群が依存している内部データを使ってしまう → 内容結合
    - 悪②：すべての機能群で一つのデータを使ってしまう → 共通結合
- 関心の分離
  - 「責任」と「責務」に着目して分割を行う

# 凝集についてもう少し

## ➤ 良い凝集とは何なのか

### ➤ 通信的凝集

➤ とあるデータに触れる処理をまとめること

### ➤ 情動的凝集

➤ 適切な概念とデータ構造とアルゴリズムをまとめること

### ➤ 機能的凝集

➤ ひとつのタスクをこなせる様にまとめること

# 副作用

- よりよいモジュール分割とは
  - データの存在を隠蔽し、関数化できること
    - 現実問題として、すべてにおいては不可能
- なぜそれが難しいか
  - それぞれのモジュールが状態と副作用を持つから
  - **副作用**とは
    - 何かしらの機能を行ったために他の状態を操作してしまうこと

# 副作用

## ➤ 関数 $f()$ について考える

- 入力 $x$ に対して出力 $y$ が一定(  $y = f(x)$  )
  - 純粋関数 → 副作用なし
- 隠れた出力 $b$ がある(  $y = f(x)$ により別に存在する $b$ が変化 )
  - 入力に対して出力は一定だが、他の機能に影響を及ぼす
- 隠れた入力 $a$ がある(  $y = f(x)$ だが別に存在する $a$ により $y$ が不定)
  - 入力に対して出力が不定
- 隠れた入出力がある
  - 入力に対して出力が不定かつ、他の機能に影響を及ぼす

# じゃあこの副作用どうすんのさ

- 状態や副作用に対して、名前を付けて可視化しよう。
- なるべく副作用による影響を少なくするために、結合を疎にしよう。
  - これらの可視化、疎結合化によって「**関心の分離**」を実現しようとした
    - オブジェクト指向に向かうモジュラプログラミング
- 状態も副作用もすべてコントロールすればいいのさ！

# 抽象データ型

- よりよいモジュール化の先にあるもの
  - （オブジェクト指向でいうところのクラス）
- データとそれに関連する処理をひとまとめでしたデータ型
- 状態や副作用をコントロールする方法
  - 隠蔽することで外部からの操作を避ける

- オブジェクト指向言語の最初のやつ
  - じつは1960年代の最後の方には出た。
- もともとシミュレーションを記述するために作られた言語
  - オブジェクト、抽象データ型、動的ディスパッチ、継承などの概念を持たせることで、汎用的に使われる言語になる。
- このコンセプトをC言語に取り入れたのが、C++
  - 若干逸れますが、C++をベースに、オブジェクトを通したメモリアクセス、ガーベジ・コレクション、仮想マシンなどの機能を取り入れて作られたのがJavaです。

# Smalltalk

- Simulaをベースに、できた言語で、Simulaのコンセプトに「メッセージング」の概念を加え、結合した言語。
- すべての処理がメッセージ式で記述される。
- 純粹オブジェクト指向言語と呼ばれる。
- SmalltalkのコンセプトでC言語を拡張したのが、Objective-C。



# 動的ディスパッチ

- どの処理を呼び出すかを決めるメカニズム
  - 同じ機能名であっても、データの種類ごとで行われるべき機能は違う。 → **多態性（ポリモーフィズム）**
- 多態性を実現するために必要な要素
  - データに自分自身が何者かを教える機能
  - メソッドを呼び出した際にそれを探索する機能
  - オブジェクト自身を参照できるように引数に束縛する機能
- データそれぞれを**オブジェクト**として扱う

# 継承・委譲

- モジュラプログラミングで分割するのは良いけど、似たような処理をまとめられないの？

## ➤ 継承

- 親子関係を結んで共通の機能は自分がもつ親の部分が行う
- 抽象データ型を持たない言語では、処理を**委譲**することで対応する。動的継承とも呼ばれたりする場合がある。
  - プロトタイプベースのオブジェクト指向と呼ばれたりする。

# オブジェクト指向の要素

- オブジェクト指向に必要な機能はなんだろう
  - データと処理を紐付ける
  - 情報を隠蔽できる
    - 抽象データ型
  - データ自身が何者かを知っている
    - オブジェクト
  - オブジェクト自身がデータの処理法を自由に探せる
    - 動的ディスパッチ → 多態性
  - 処理法をどこから探すか
    - 継承、委譲

# メッセージングだけはちよい違う

- メッセージと関数呼び出しは似て非なるもので、メッセージは形式に囚われない。
  - 受け取った関数が自由に解釈を行い、処理を行う。
- メッセージングの特徴
  - カスケード式
    - 複数のメッセージをまとめて送る
  - メッセージ転送
    - 受け取ったメッセージに対する処理が定義されていない場合でも自由に取り扱える
  - 非同期送信
    - メッセージはそれぞれが独立して処理されるため同期的に待ち受けている必要はない

# オブジェクト指向の更にその先

## ➤ デザインパターン

➤ 実現可能な処理をパターン化したもの。

➤ 無闇矢鱈と使うものではなく、オブジェクト指向プログラミングにおける、こんな時はこういう設計・実装方法があるよ！という参考資料です。

➤ 言語仕様にすでに組み込まれているパターンもあります。

➤ オブジェクト指向では、GoF(Gang of Four)なんかが有名で、プログラミングに慣れてきた頃合いに調べてみると、その考え方や手法はとても参考になるでしょう。

# オブジェクト指向の更にその先

- よりよいプログラム(アプリケーション)を作成するためのルールや規約ができる。
- SOLID原則
  - 単一責務の原則(Single Responsibility Principle)
  - 開放/閉鎖の原則(Open-Closed Principle)
  - Liskovの置換原則(Liskov Substitution Principle)
  - インターフェイス分離の原則(Interface Segregation Principle)
  - 依存関係逆転の原則(Dependency Inversion Principle)
- デメテルの法則
- などなど

# 今すぐ全て理解する必要はありません

- 授業を受けた、本を読んだ、それだけでオブジェクト指向プログラミングを理解するのは大変難しいです。
- オブジェクト指向を学ぶ方法の一つとして、Javaでのプログラミングを学びながら、少しずつオブジェクト指向についての理解を深めていきましょう。
- どうしてもオブジェクト指向を極めたい！という方は、「オブジェクト指向入門 第2版 原則・コンセプト」と「オブジェクト指向入門 第2版 方法論・実践」を買って読みましょう。私の講義では、この内容を網羅できません。

# まとめ

まとまってないまとめ





# オブジェクト指向

- プログラミングをもっと快適にするために生まれた考え方
- 特に大規模・多人数での開発を行う際に役に立つ
- コードを書く前に、きちんと設計を考える必要がある
- 先人たちが残した考え方を学ぶことで、自分のスキルにできる
- オブジェクト指向で作られたものを使うことは簡単にできる