オブジェクト脳のつくり方

2003年12月16日(火)

日本電気株式会社

牛尾　剛

はじめに

- プロジェクトの度に直面する「オブジェクト指向教育」で本能と反射で編み出したオブジェクト脳化テクニックをご紹介いたします。
- 著書出版後にML「オブジェクト脳オンライン」で議論された内容が追加されています。
- あくまで私の考えで構成したもので、未完成です。一緒に考えていきましょう。

目次

- 日本におけるオブジェクト指向の現状
- オブジェクト指向はなぜ「とっかかり」が難しいのか
- 経験によるオブジェクト指向技術者育成戦略
- オブジェクト脳育成講座デモンストレーション
- 豆知識：日本語と英語のインピーダンスミスマッチ
- 豆知識：オブジェクト指向とクラス指向

日本におけるオブジェクト指向の現状

●技術要素がオブジェクト指向ベースになってきている

・Java, .NET, UML
・J2EE, Webサービス

●オブジェクト指向でやるべきところでも非オブジェクト指向のケースが多い

・Java, J2EE
・.NET

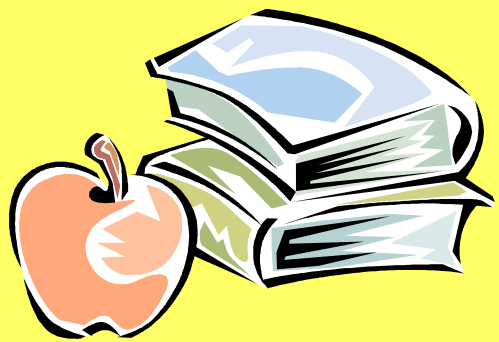
非オブジェクト指向でも
コーディング可能ただ
し効果は・・・

●オブジェクト指向に興味がある人以外でも取り組まなければいけない状況にある

●まともなオブジェクト指向技術者にはSIの現場ではほとんどお目にかかれない

オブジェクト指向はなぜ「とっかかり」が難しいか

- りんごとかみかんとか抽象的で意味不明
- 一般的なメリットの理由がわからない
- 業務で使えなさそう。使えるイメージがしない
- 習得するには体を使わないといけない
- あるときまでわかった気がしない。
- 習得に時間がかかる（3年程度）
- 正確さを重視しているので、初心者だと敷居が高い
- 初心者は入門書よりも低いレベルを求めている
- 大切な部分や必用な部分が十分解説されるまえに、他の知識を詰めこまれる
- わかりそうな気がしない
- 用語が難しい
- 周りに知っている人がいない
- 周りのプロジェクトでやっていないのでサンプルコードがない
- オブジェクターは教え下手が多い
- できないので楽しくない



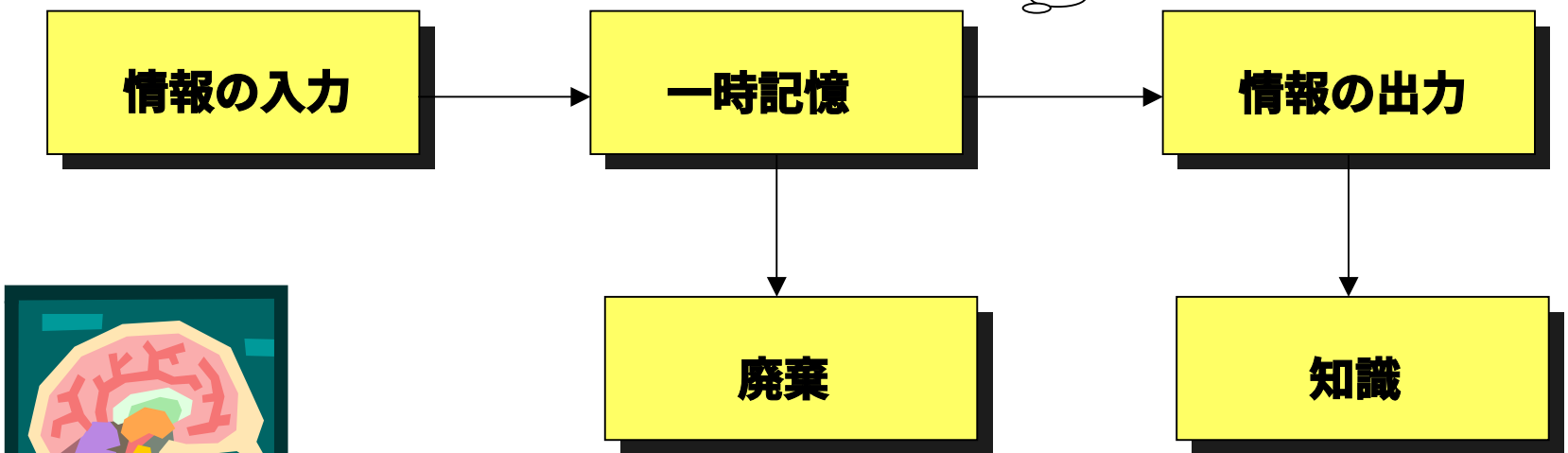
記憶のプロセス

●情報を聞いて「わかった」と思った段階ではそれは「知識」になっていない。
入力された「情報」を出力して「知識」に変える必要がある。

●出力されない情報は廃棄される

●出力の後でわかることもある（ex.英語学習）

情報と知識は
違う



通常の新技術とOO習得の違い

通常の新技術

経験する

新技術を学ぶ

イメージがわく

実践する

知識に変わる

OOがわからない人

経験する

OOを学ぶ

イメージわかない

あきらめる

OOがわかる人

経験する

OOを学ぶ

イメージわかない

実践する

オブジェクト脳発生

イメージがわく

継続学習の意欲がわく

人間の4つのタイプ

●コントローラー(Controller)

行動的でエネルギッシュ。思い通りにしたがる。他人の不正や偽善を暴露することを使命と考える。正直であけっぴろげ。過剰に快感を求め、依存しやすい。自分の内面に目を向けるのは苦手。

●プロモーター(Promoter)

アイデアが豊富で創造力がある。楽しいことがすき。計画を立てるのは苦手。悪く言えばあきっぽく、よく言えば変化への順応性が高い。包括的な仮説を立てる才能がある。アイデアは成り行気任せで非現実的だが、新しい可能性を見つける。仕切るのが好きで、得意でもある。

●アナライザー(Analyzer)

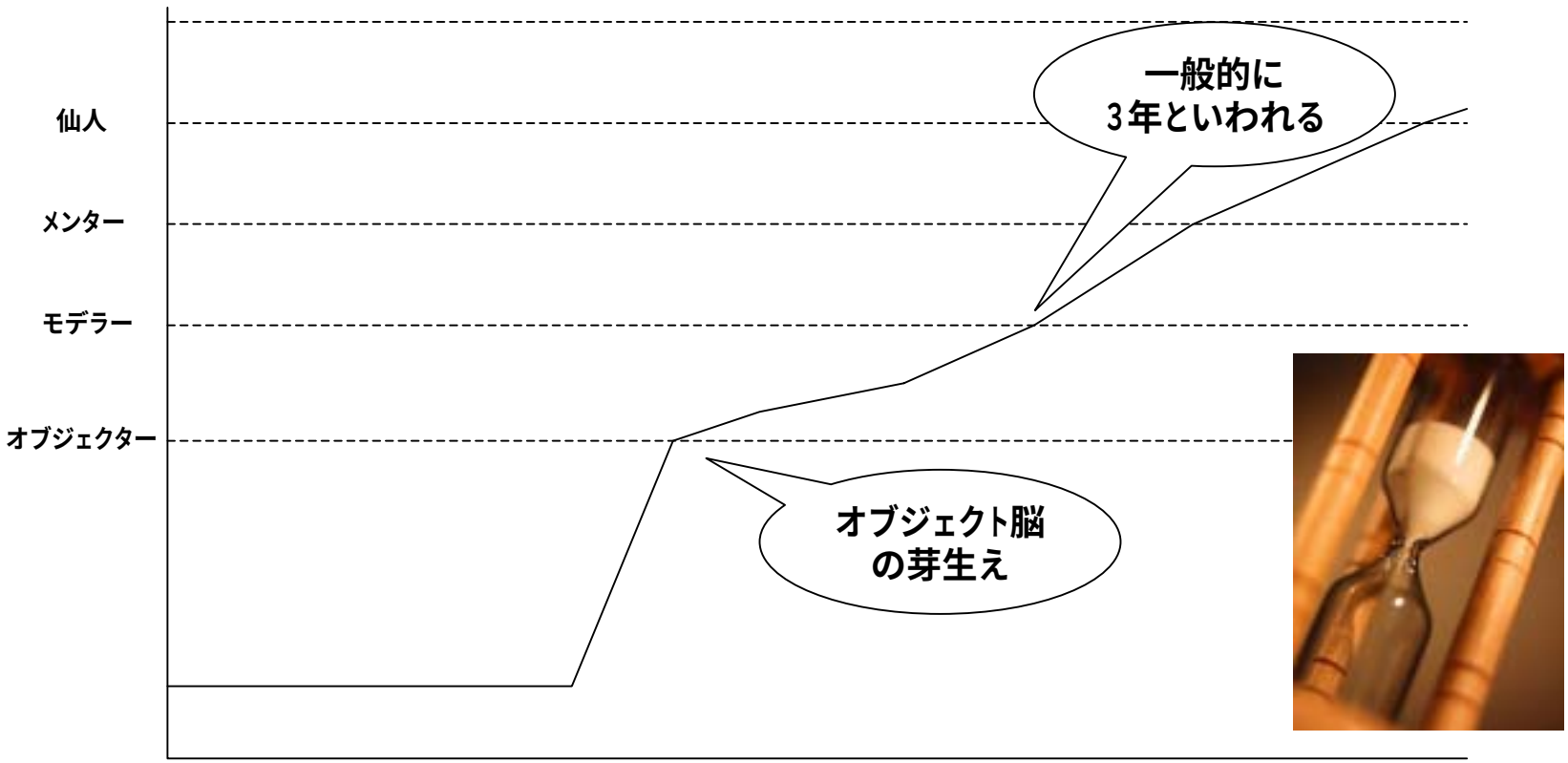
物事に取り組むとき、データに基づく分析からはじめる。計画を立て、粘り強く最後までやりとげる。変化や混乱には弱い。理解力や洞察力が高く、的確な意思決定ができる。明確で論理的な話し方をする。

●サポーター(Supporter)

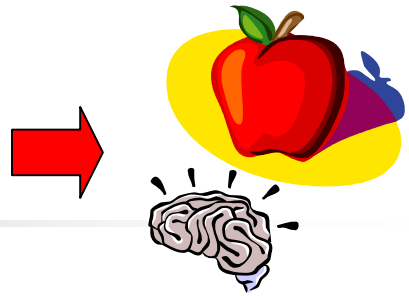
人を援助したがる。温かく、穏やか。職場では協調性が高く、意欲もある。計画や目標を立てることには関心がない。決断には時間がかかる。人の心を読むのが得意。直感力がある。感情に基づいて判断する。

http://nba.nikkeibp.co.jp/coachsp.html

It takes time.



OO化への戦略



●過去の経験との切り離し

- 概念を解説しているときにコンピュータへの置き換えを考えることを禁止

●「情報」より「知識」を重視する

- よくわからないときは必ず手を動かして体験して知識化する
- わかったと思ったときも必ず手を動かして知識化する
- 重要なことにのみフォーカスして、確実に知識にする
- カット＆ペースト禁止

情報インプットより知識重視

●時間差攻撃戦略

- よくわからない状態もOKとする
- 「正確なもの」より「わかりやすいもの」を重視
- 計画として、将来の修正を見込んでおく

人間は勘違いするし、修正もできる

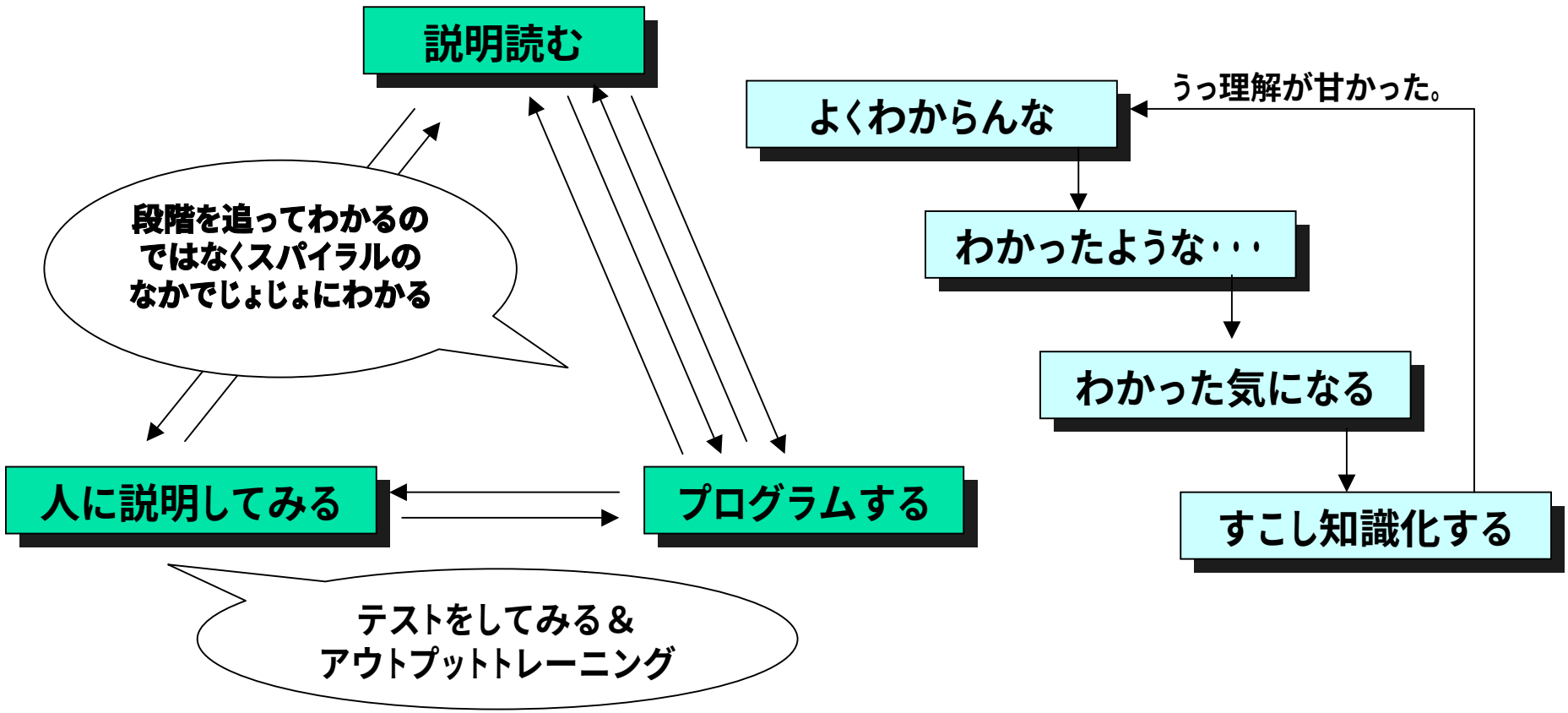
OO化への戦略

●継続的学習への意欲

・各トピックでどれだけ得をできるかを必ず解説／体感してもらう。
・自分でも「オブジェクト指向がわかった」という気持ちになれる。

オブジェクト脳をまず芽生えさせる

不完全主義のプロセス



注：このプロセスが有効じゃない人もいます。

「オブジェクト脳をつくる」とは

●オブジェクト指向がわかった（気になる）きっかけ

・多くの人はある日あるきっかけで突然オブジェクト指向がわかった気がする。
・多くの人は「ポリモーフィズム」の理解でわかった気がする人が多いようである。

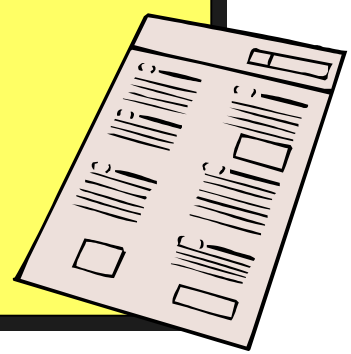
●オブジェクト脳をつくる試み

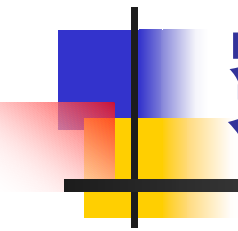
・オブジェクトマスターは難しいが「ある日突然」を人工的に起こしてみることは短期間で可能
・「わかったつもり」になったら繰り返し手を動かし、知識化を重点的に行う

オブジェクト脳判定基準

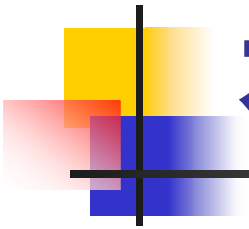
●下記の内容を以下の3点に関して説明できること
①概念的に説明できる（コンピュータに置き換えずに）
②コードに置き換えて説明できる
③メリットを説明できる

- オブジェクト指向とは
- オブジェクトとは
- クラスとは
- メッセージ
- 継承
- カプセル化
- ポリモーフィズム





究極の基礎講座デモ



本章の内容について

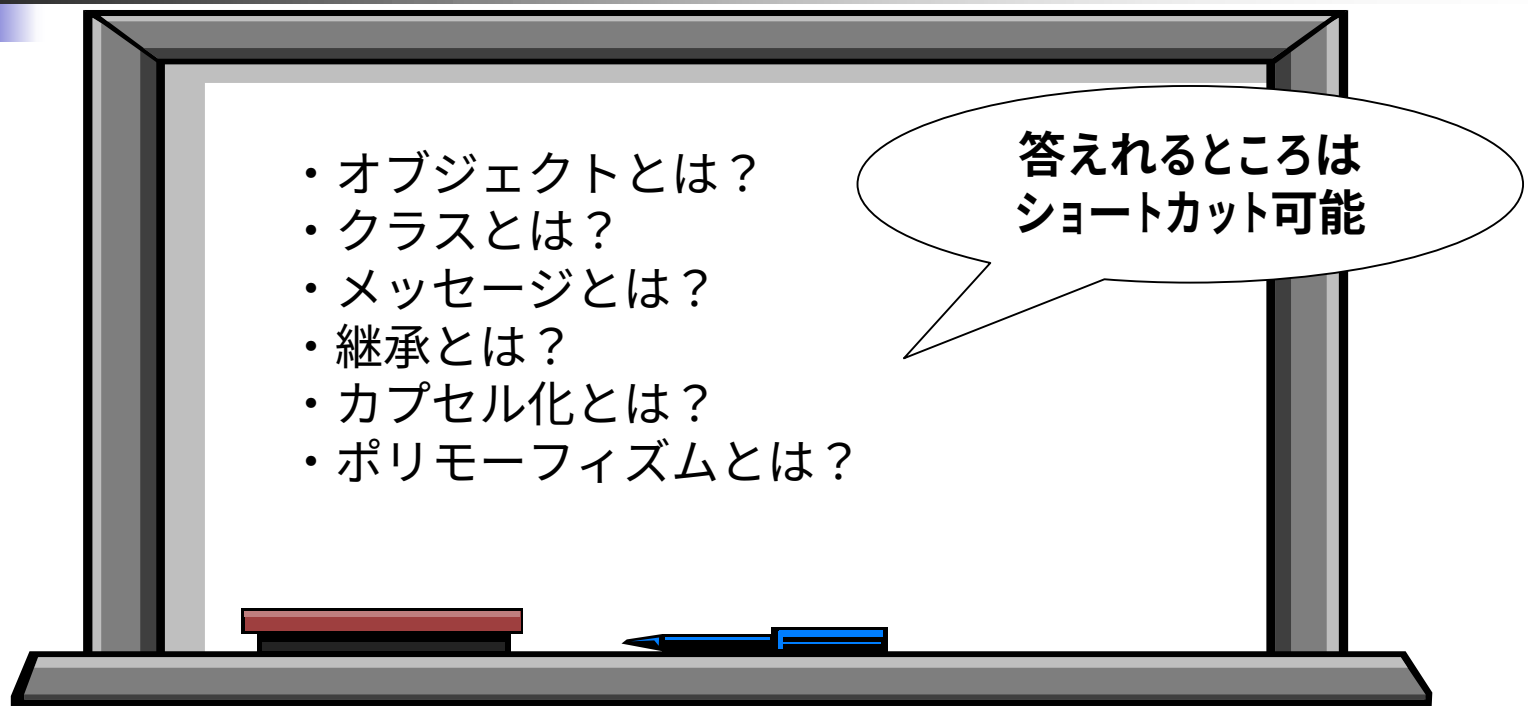
- 講演時間が限られていますので、講演時間内にオブジェクト脳発生は難しいです。
- 家に帰って演習をやるとオブジェクト脳発芽ができるように構成されています。
- サンプルはオンラインで入手できます。

演習を始める前に

1. コードやコンピュータの動作をイメージして考えないこと
2. 業務で使うことを考えないこと
3. どういう風に得かを考えないこと
4. わかっても、わからなくても必ず手を動かして演習を行うこと

上記のルールで演習してください

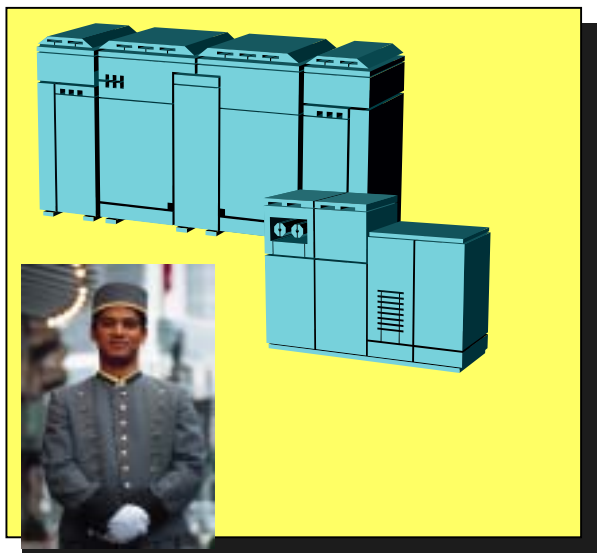
最初の質問



1. **コンピュータの動作イメージを想い浮かべたり、コードに置き換えず、それぞれについて説明してください。**
2. **コードに置き換えて、それぞれについて説明してください。**
3. **それぞれのメリットについて説明してください。**

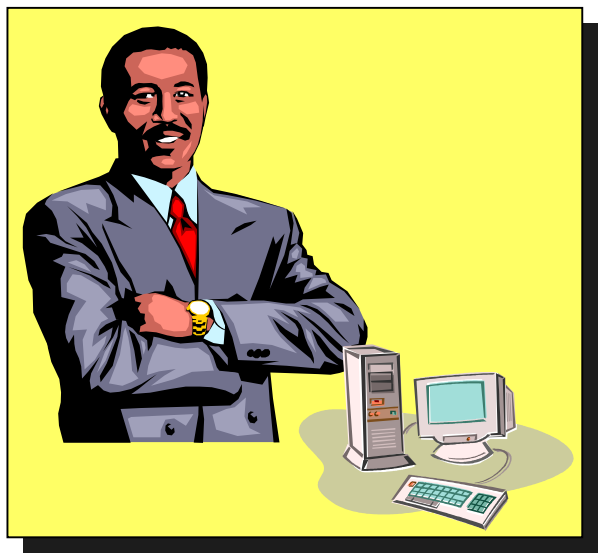
オブジェクト指向開発とは

従来型



コンピュータさまに都合のいいようなコーディングをしていた

オブジェクト指向



人にわかりやすいコーディングをこれからする

●オブジェクト指向は人間の都合に合わせて、人間の世界（業務など）をモデル化するとそのままプログラムができてしまうような開発方法

オブジェクトとは？

・オブジェクトとは実際にある「もの」や「概念」である。

具体例
牛尾　剛、あなた、あなたの持っているＰＣ、受注、本

1. 現実社会に存在するもの
2. 「受注」とか「サービス」など物体はないが存在する物も「もの」としてとらえる。

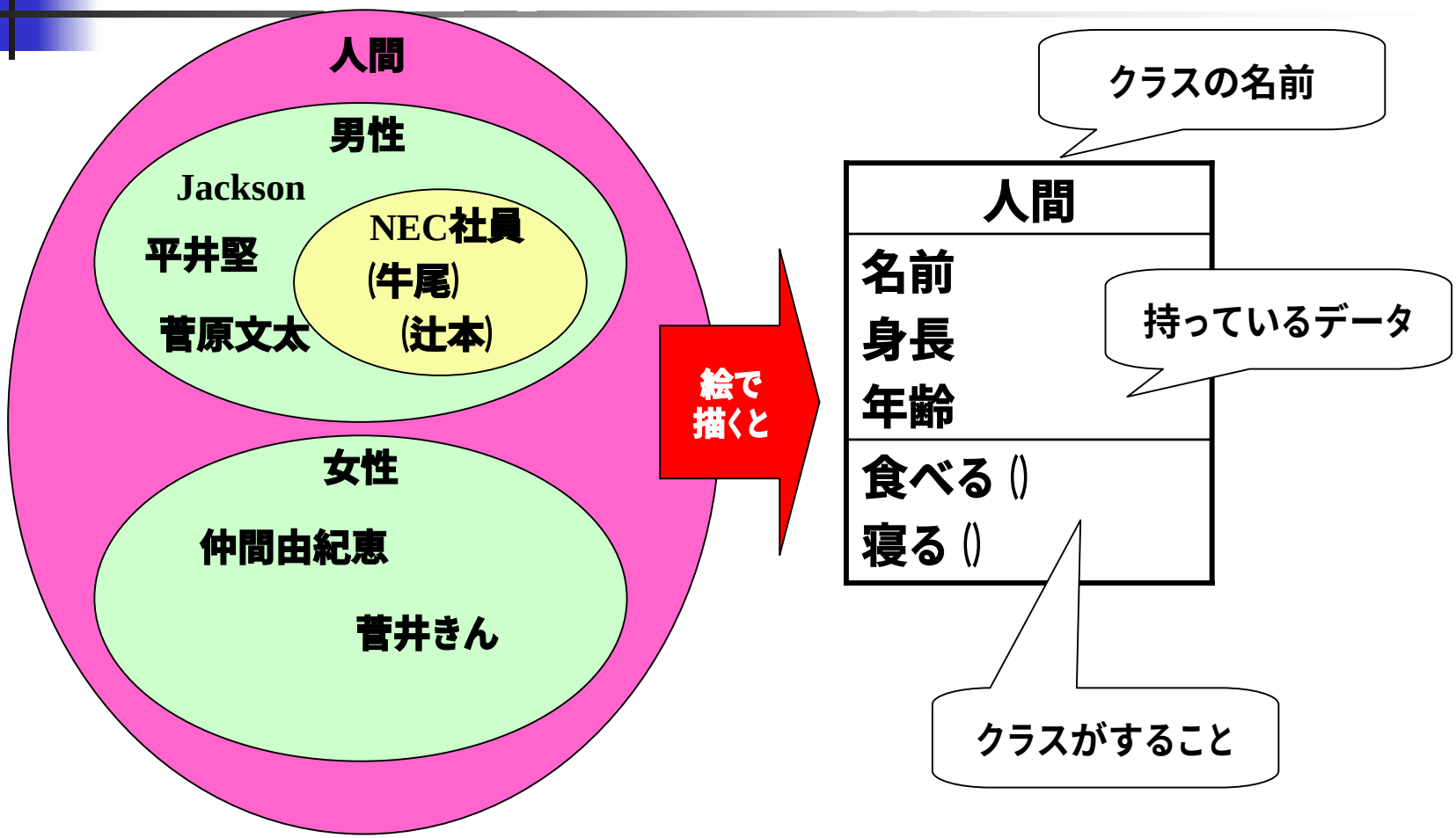
クラスとは？

・もの（オブジェクト）のうち共通の性質を持ったものの「くくり」をクラスと呼ぶ

オブジェクトとクラスの違いは？

- 具体的に存在するもの、「牛尾剛」「あなた」「あなたの持っているＰＣ」「牛尾のＰＣ」はオブジェクト。
- ひとくくりにした定義「人間」「ＰＣ」がクラス。

クラスの表し方



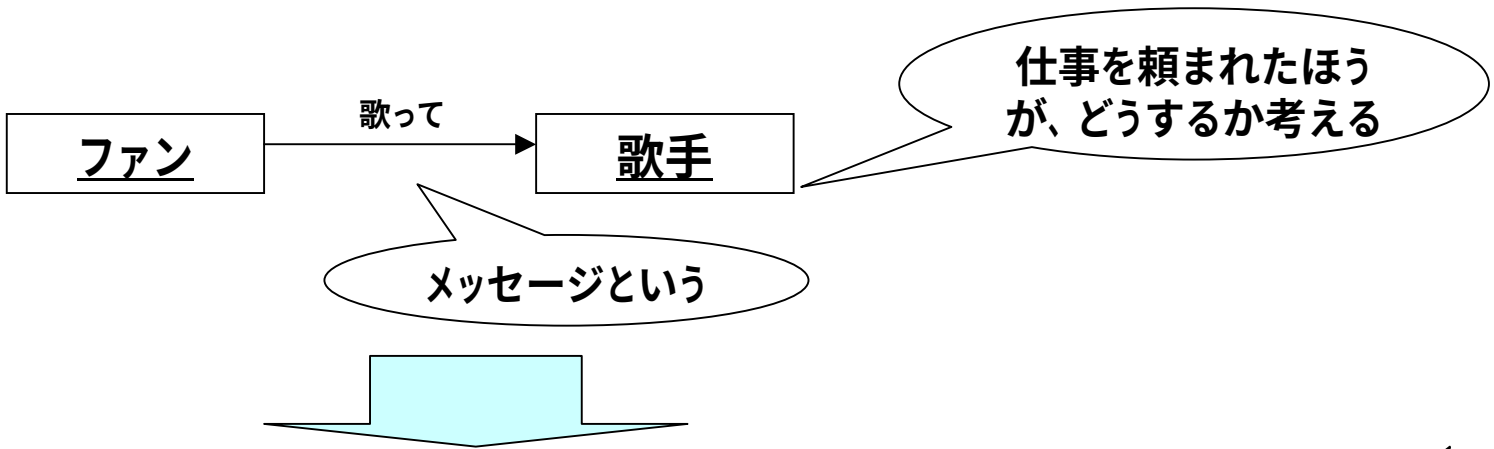
括り方の表現は、いろいろできる。

メッセージ

モデル化はなんか仕事をしてもらうことをモデル化する

●オブジェクト指向システムの動き方

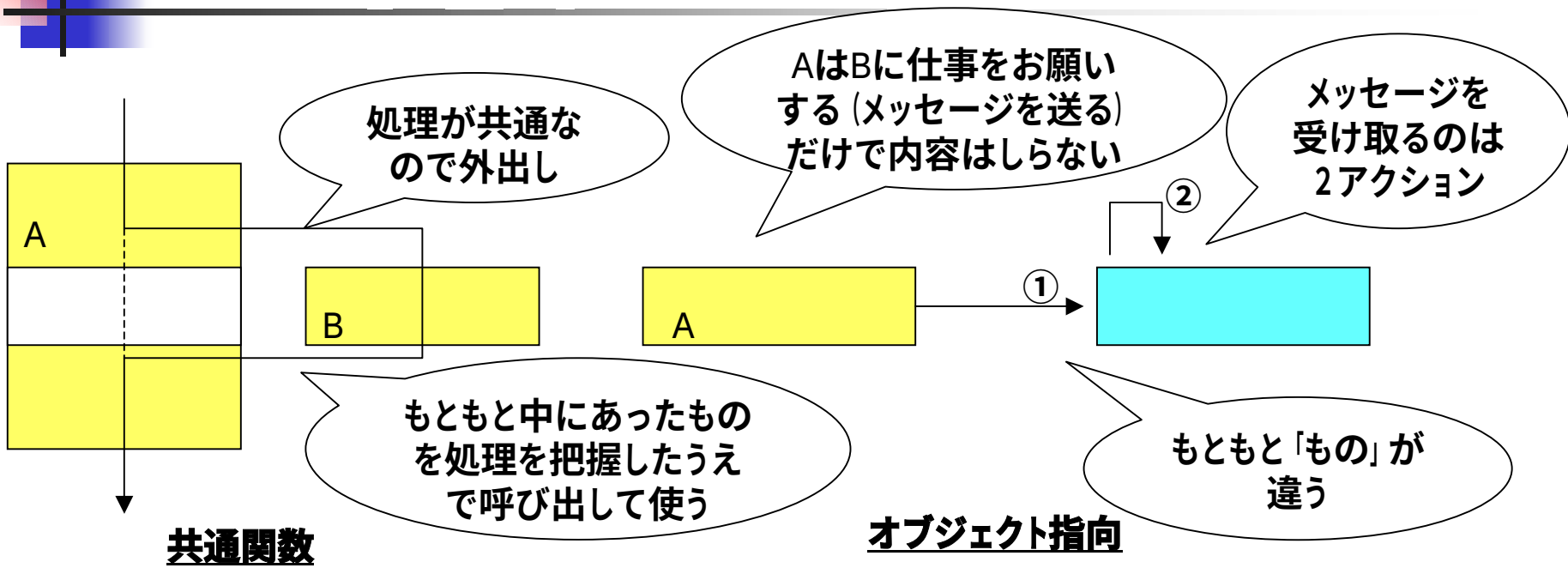
Ex.ファンが歌手に「歌を歌ってもらう」を単純にモデル化してみた。



メッセージを受け取ったのが
和田○き子なら「あぁ～悩み無用！」と歌うし、
Michael ○cksonなら「Hoo!」とか歌うはず

コンピュータに
置き換え厳禁！

メッセージと関数呼び出しの違い(参考)



●従来型の場合、Bにある必要な関数をAが呼び出す。
●オブジェクト指向の場合、Aは仕事をお願いするだけ(メッセージをおくるだけ)
Bがメッセージを受け取ってどうするか考える

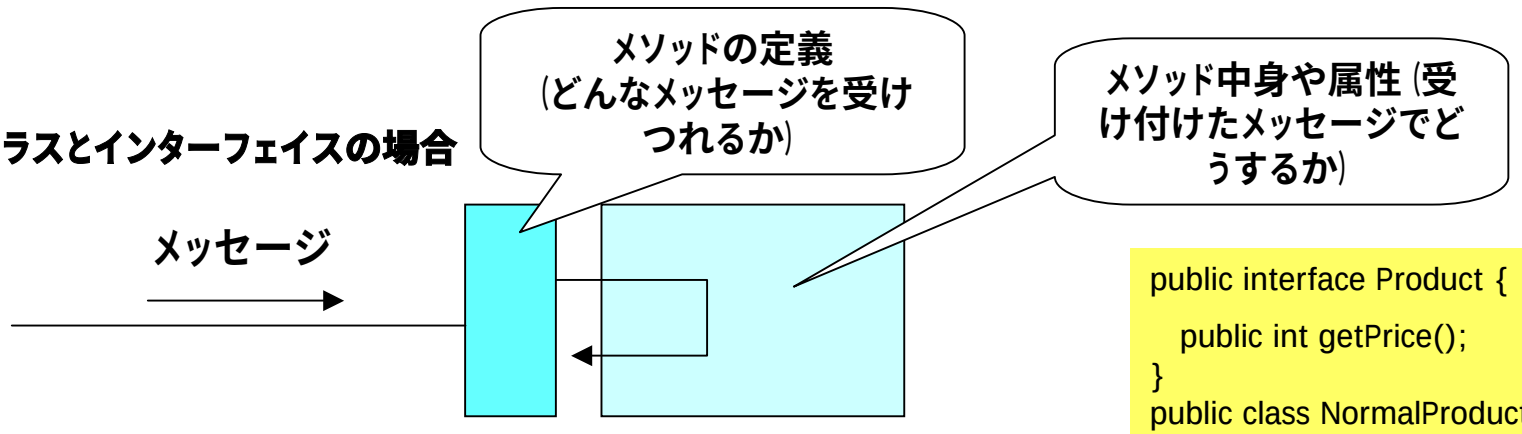
1. Aがメッセージを送る
2. Bがメッセージを受け取る
3. Bが受けたメッセージに対してどういう処理をするか考える

クラスとインターフェイスのイメージ(おまけ)

●Javaなどでは、メッセージの感覚が沸きにくいが、私はこう考えていますという例です。(いわゆるJavaの正確な動作を表しているのではありません)

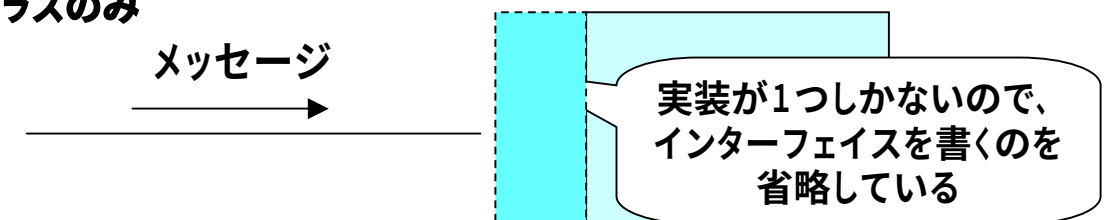
・普通はインターフェイス＋クラス(実装)でひとつ
・クラスが1このみの場合はデフォルトでインターフェイスがついているイメージ

①クラスとインターフェイスの場合



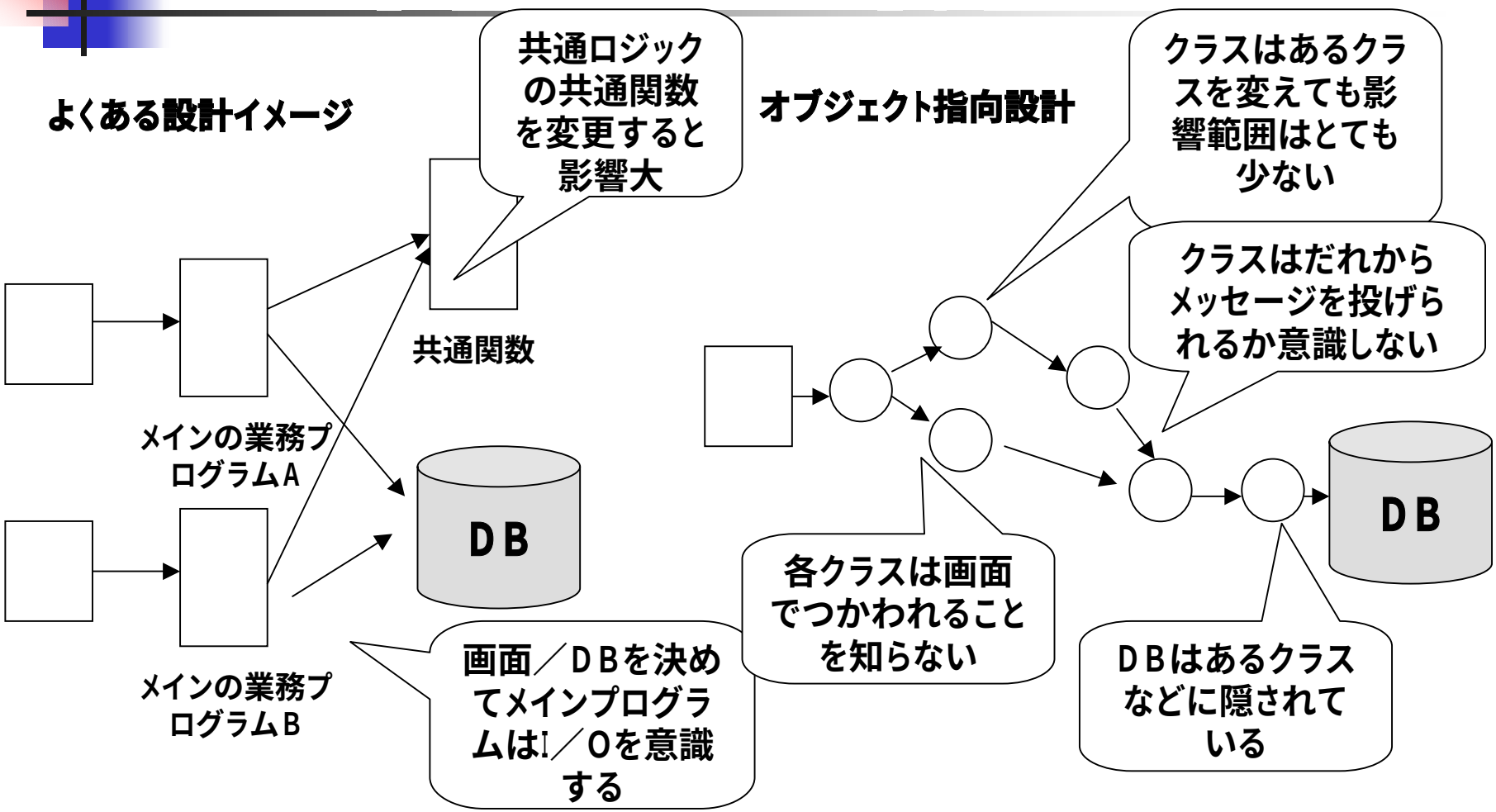
インターフェイス　クラス(実装)

①クラスのみ




```
public interface Product {
   public int getPrice();
}
public class NormalProduct implements
Product {
              :
  public int getPrice() {
      return price;
 }
   private int price;
}
```

よくある設計とオブジェクト指向のイメージの違い



※よくある設計とは特定の手法(OO,DOA,構造化など)をさすものではなく、現在開発の現場で多く見受けられる設計です

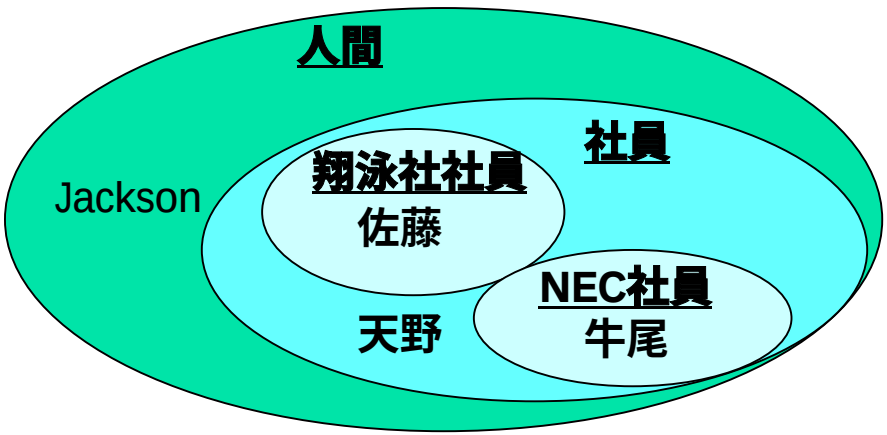
継承とは？

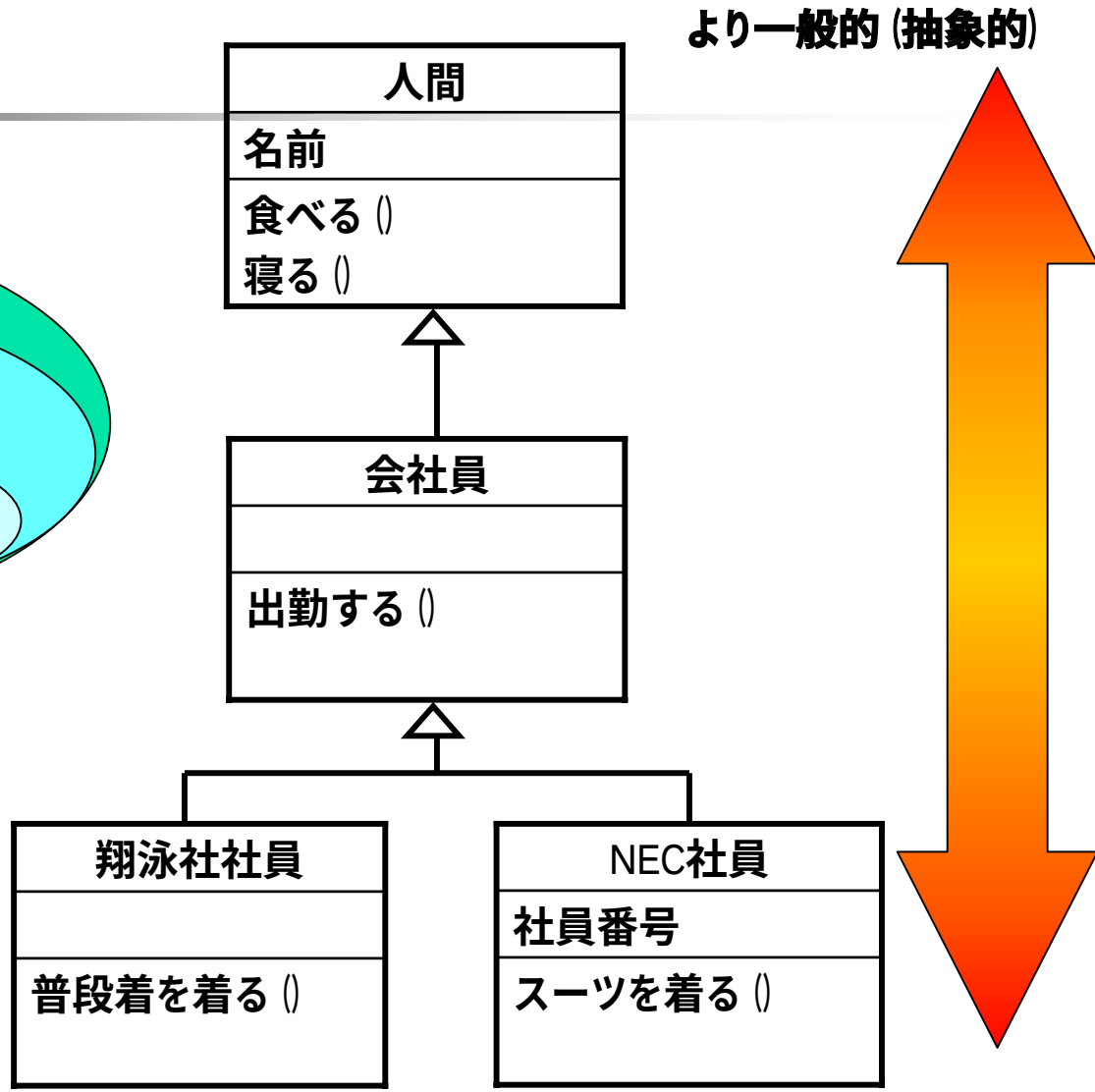
・具体的なクラスは一般的なクラスの性質を引き継いでいる

世の中のものをより一般的に括ることを
　・抽象化
より具体的に括ることを
　・具象化
と呼ぶ

1. 具体的なクラスは、一般的なクラスの性質を引き継ぐ。
2. 具体的なクラスは、一般的なクラスの性質を持っている。
3. 一般的なクラスは、具体的なクラスの性質をもっているとは限らない。

継承とは？


人間
社員
翔泳社社員
佐藤
Jackson
NEC社員
牛尾
天野


より一般的 (抽象的)
人間
名前
食べる ()
寝る ()
会社員
出勤する ()
翔泳社社員
普段着を着る ()
NEC社員
社員番号
スーツを着る ()
より具体的 (具象的)

カプセル化適当説明

上司が部下へ仕事を
指示する場合。

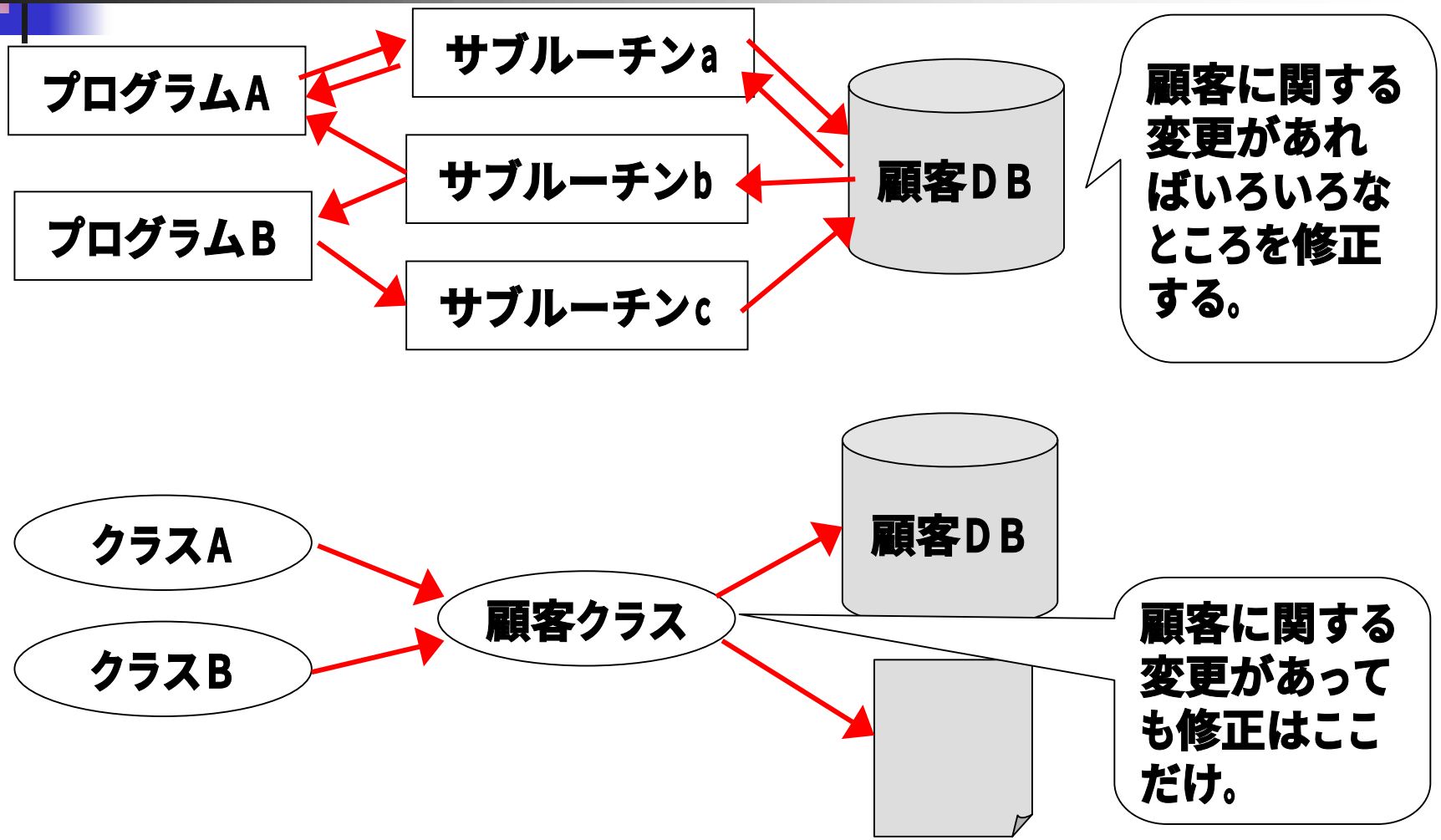
スタイル1
あれ、これ、それを
こういう手順でこの
仕事をしなさい。

スタイル2
どんな方法でもよ
いから、この仕事を
しなさい。

仕事は頼まれ
たほうがどう
するか考える

オブジェクト指向
的教えでは、方
法は、上司は知
らんほうがいい。

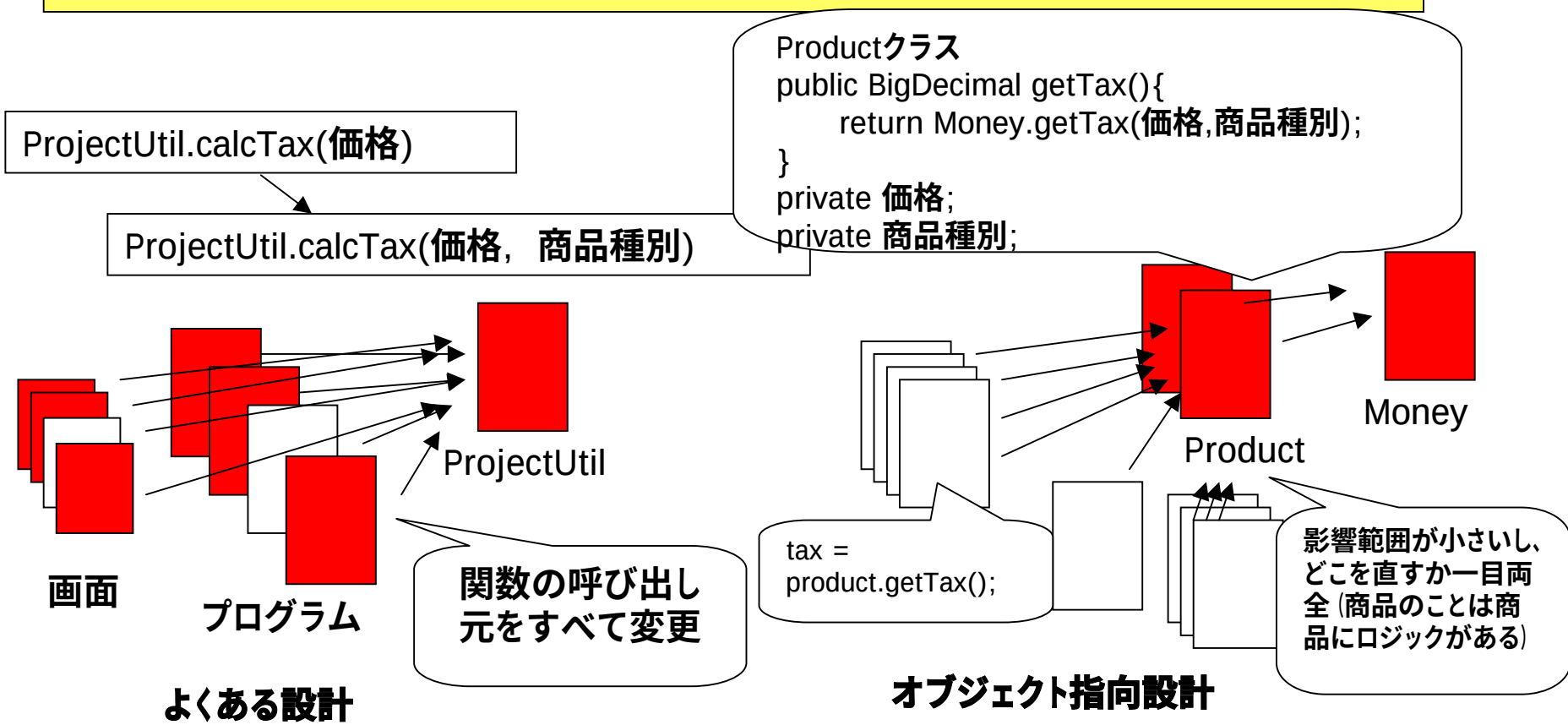
カプセル化のお得なイメージ



カプセル化のお得なイメージ

●税金の計算が変更になった場合

・税金の計算が「価格 x 0.05」だったものが
商品の種類によりあるものは「価格 x 0.05」、あるものは「価格 x 0.07」に変更された。



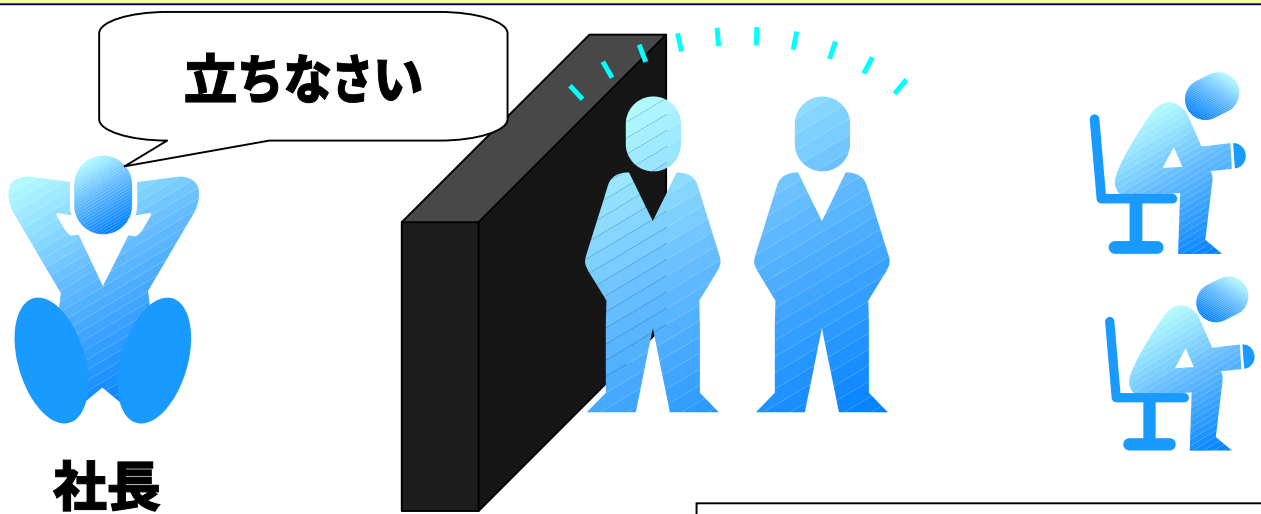
カプセル化とは

・クラスに仕事を割り振るときに、仕事の具体的なやり方はその仕事の責任を持った担当（クラス）だけが知っているようにして、その担当に仕事を頼む人は、その中身を知らないようにしなさい。

カプセル化は基礎の中で一番むずかしいのでわからんでも気にしないこと。イメージをつかもう。

究極奥義：ポリモーフィズム

・社長が命令すれば、「どんな役職の社員でも起立する」という性質をポリモーフィズムと呼ぶ



社長は、壁の向こうに誰がいるかは知らないけれど社員と思って「立ちなさい」と命令をする。

社員は立ち方はどうであれ立つ
部長：だるそうに立つ
主任：すばやく立つ
担当：普通に立つ

ポリモーフィズムの図

ここには社員の誰かがくる

社長 → 起立しろ → 社員

ここにきて起立命令を受けた人は自分でどうするか考える

①社長から命令を受けたのが平社員の場合の図

社長 → 起立しろ → A君：担当

普通に立つ

②社長から命令をうけたのが主任の場合の図

社長 → 起立しろ → B君：主任

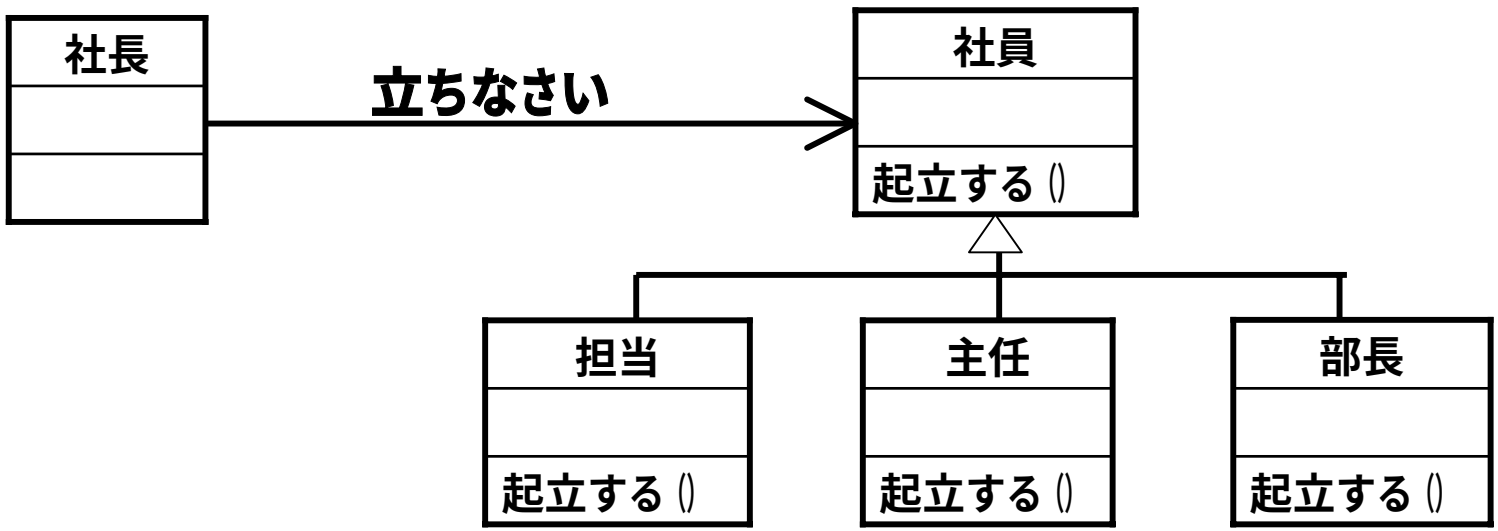
すばやく立つ

③社長から命令をうけたのが部長の場合の図

社長 → 起立しろ → C君：部長

だるそうににに立つ

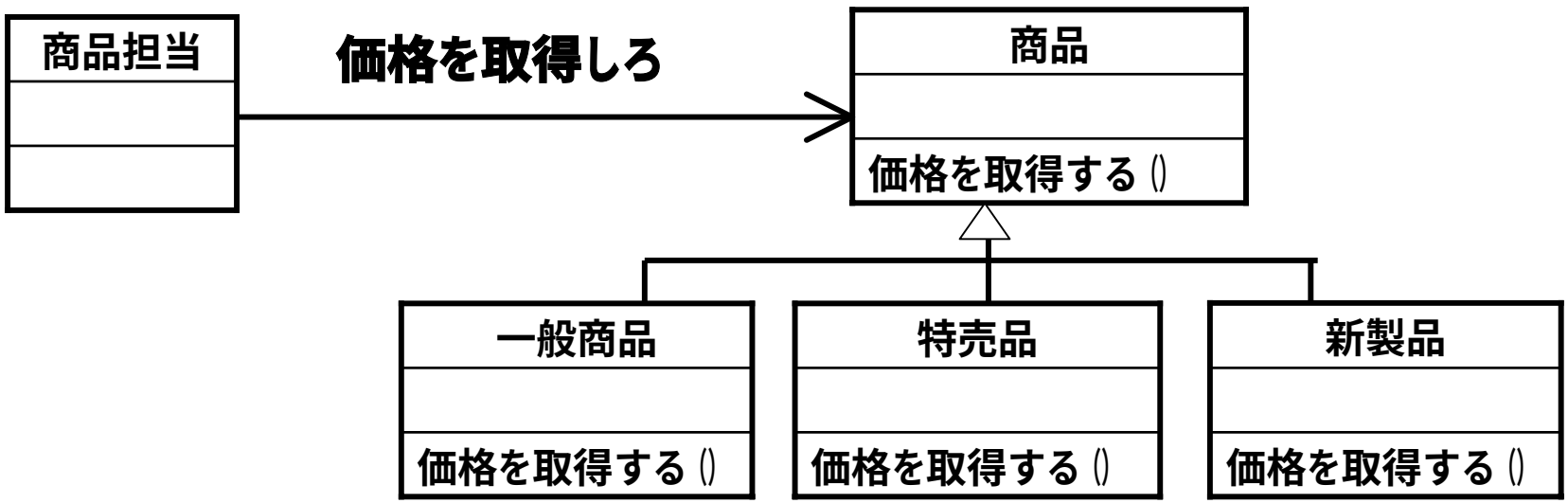
ポリモーフィズムの図



社長は、結果的に担当のAさんや、部長のBさんに対して「立ちなさい」と命令しているが、社長自身は「社員」と思って「起立命令」を出す。

※モデルは厳密に言うと正解ではないが、気にしたら負けだ。

他のポリモーフィズム例



- 商品担当は、「一般製品」や「特売品」を意識することなく「商品」に対して「価格」を取得する。

他のポリモーフィズムの例

・価格の計算方法

一般商品	通常価格
特売品	通常価格 × 0.5
新製品	通常価格 × 2 ＋ 100

・商品担当者は、商品に対して仕事を頼んでいるので、一般商品や特売品の価格計算方法が変わっても、新製品が追加されても影響を受けない。

この性質こそが、オブジェクト指向をして
「**変化に強い**」といわしめているもの

演習

- オブジェクト指向は人間のご都合で、人間の世界をモデル化したものらしいので、実際にそのままプログラムを作って見ましょう。

※本章のコードは以下のＵＲＬからダウンロードできます。

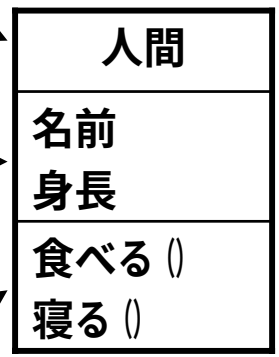
http://www.geocities.jp/objectbrain/
社長命令･起立！ Javaオリジナルバージョンのリンク

前提知識：クラスとコード

クラス名

属性
(private変数)

メソッド



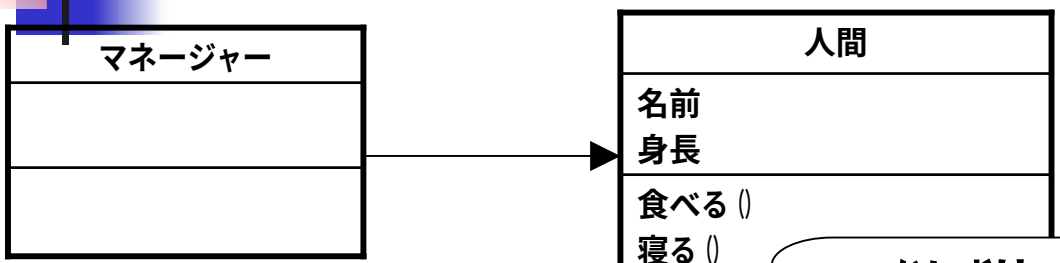
コンストラクタ。クラスに魂をいれてオブジェクトをつくるときに呼ばれる。

Ningen.java

```
public class Ningen{
    private String name;
    private int shincho;
     public Ningen(String name, int shincho)
     {
        this.name = name;
        this.shincho = shincho;
    }
    public void taberu(){
        System.out.println("食べてます。");
    }
     public void neru(){
        System.out.println("寝ています。");
    }
}
```

taberuとかneruと命令されると“食べてます”とか“寝てます”と応える。

前提知識：人間とマネージャ



C:¥> javac *.java

とコンパイルすると
Manager.classとNingen.classができあがる。

Mangerを実行すると

C:¥> java Manager

食べてます。

寝ています。

Manager.java

```
public class Manager{
    public static void main(String[] args){
     Ningen ningen = new Ningen("牛尾", 169);
     ningen.taberu();
     ningen.neru();
    }
}
```

mainメソッドは、コマンドラインから実行可能

人間クラスに具体的な情報を与えて魂を入れる。

コンストラクタと呼ぶ。Ningenクラスを実行する。

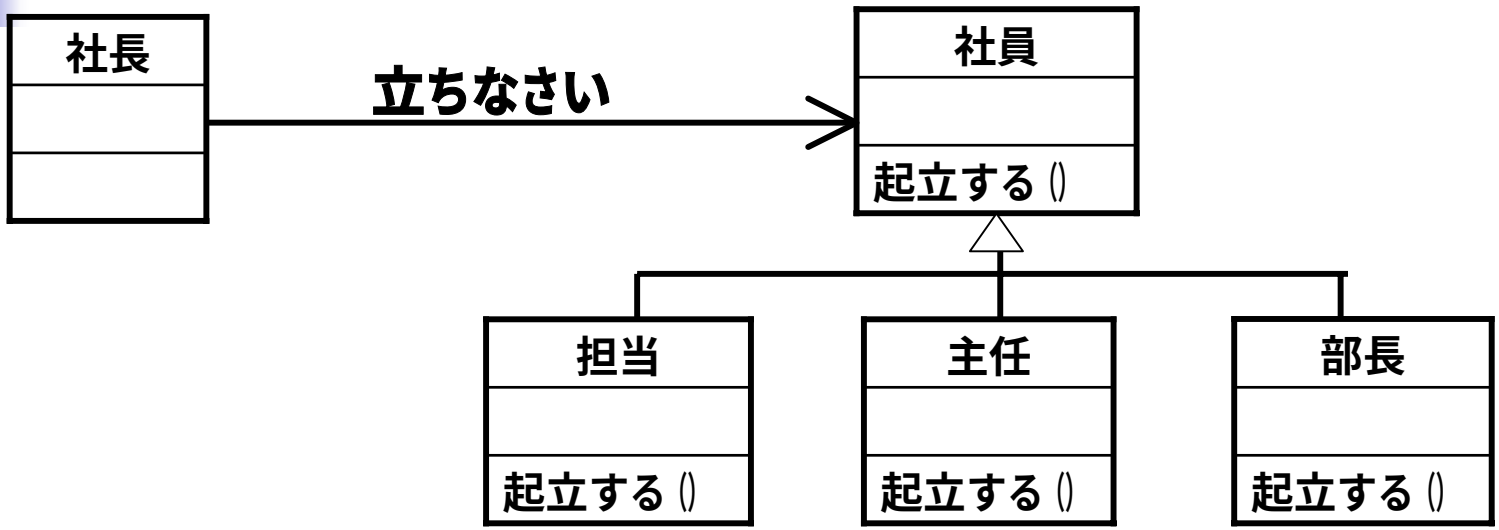
Ningenクラスにtaberuとかneruと仕事をお願いする（メッセージをおくる）。

Javaだと関数とメッセージは見分けがつかないが受け取ったほうが何するか考えるイメージ

牛尾 ： 人間
牛尾
169

※クラスは魂を入れてお部ジェクトにしないと動かせない。

演習1：社長命令・起立



実行すると次のようになるプログラムを作ってみる。

C:¥> javac Shacyo Tanto

担当が普通に立ちました。

C:¥> javac Shacyo Shunin

主任がすばやく立ちました。

C:¥> javac Shacyo Bucho

部長がだるそうに立ちました。

演習１：回答例

Shain.java

```
public interface Shain {
   public void standup();
}
```

ここでは、classではなくinterfaceを使って「社員というものは起立する。」ことを表す。Interfaceを使う場合はロジックは書かない。

Shunin.java

```
public class Shunin implements
    Shain {
   public void standup(){
      System.out.println("主任がすばやく立ちました。");
   }
}
```

Tanto.java

```
public class Tanto implements Shain
    {
   public void standup(){
      System.out.println("担当が普通に起立しました。");
   }
}
```

implementsでShainを継承し、Tantoクラスに具体的にどう起立するかをコーディングする。

Bcho.java

```
public class Bucho implements
    Shain {
   public void standup(){
      System.out.println("部長がだるそうに立ちました。");
   }
```

演習１：回答例

Shacho.java

```
public class Shacho {
    public static void main(String[] args)
     {
       Shain shain = null;
       if(args[0].equals("Tanto")) {
          shain = new Tanto();
       }
       if(args[0].equals("Shunin")) {
          shain = new Shunin();
       }
       if(args[0].equals("Bucho")) {
          shain = new Bucho();
       }
       shain.standup();
    }
}
```

Shachoクラスはmainによりコマンドラインから実行可能なクラス。
引数はStringの配列argsに入る。

社長は、部長など具体的な人に「起立」と命令しますが、誰であろうと社員には変わりないので「社員」の定義だけを記述。

args [0]が「Tanto」なら{}内の命令を実行。

Tanto tanto = new Tanto();
とコーディングすることも可能だが、オブジェクト指向言語の利点で「担当」は「社員」を継承しているので「社員」として扱うこともできる。だからここでは「担当」を「社員」に代入できる。

最後に「社長」は相手が誰かを気にせずに「社員」に対して「起立」命令を出す。
これぞ、オブジェクト指向最大の発明「ポリモーフィズム」。

演習２：給料はいくら

・給料計算ルール

担当	基本給　が給料
主任	基本給　×　２　＋　１　が給料
部長	基本給　×　３　が給料

基本給を教えるから自分の給料を計算しなさい。
基本給は100円だよ。

C:¥> javac Shacyo Tanto 100

担当が普通に立ちました。
私の給料は100円です。

C:¥> javac Shacyo Shunin 100

主任がすばやく立ちました。
私の給料は201円です。

C:¥> javac Shacyo Bucho 100

部長がだるそうに立ちました。
私の給料は300円です。

演習2：回答例

Shain.java

```
public interface Shain {
    public void standup();
    public int getKyuryo(int kihonkyu);
}
```

基本給を教えるから君の給料を答えろ。

Shunin.java

```
public class Shunin implements Shain {
    public void standup(){
        System.out.println("主任がすばやく立ちました。");
    }
    public int getKyuro(int kihonkyu){
        return kihonkyu * 2 + 1;
    }
}
```

Tanto.java

```
public class Tanto implements Shain
    {
    public void standup(){
        System.out.println("担当が普通に
        起立しました。");
    }
    public int getKyuryo(int kihonku){
        return kihonkyu;
    }
}
```

担当は、「基本給のままです。」と答える。

Bucho.java

```
public class Bucho implements Shain {
    public void standup(){
        System.out.println("部長がだるそうに立ちました。");
    }
    public int getKyuryo(int kihonkyu){
        return kihonkyu * 3;
    }
}
```

部長は、「基本給の3倍です。」と答える。

演習2：回答例

Shacho.java

```
public class Shacho {
    public static void main(String[] args) {
        Shain shain = null;
        if(args[0].equals("Tanto")) {
            shain = new Tanto();
        }
        if(args[0].equals("Shunin")) {
            shain = new Shunin();
        }
        if(args[0].equals("Bucho")) {
            shain = new Bucho();
        }
        int kihonkyu = Integer.parseInt(args[1]);
        int kyuryo = shain.getKyuryo(kihonkyu);
        shain.standup();
        System.out.println("私の給料は" + kyuryo + "円です。");
    }
}
```

「社長」は「担当とか部長」ではなく「社員」に対して「基本給を教えるから給料がいくらか答えなさい」と命令。

パラメタで引き渡された文字列をint型の数値に変換。

Interfaceを継承したクラスのロジックが何種類あろうと、呼び出し元のロジックは1つでだけ。
「課長」が追加されても呼び出しロジックには全く変更がない。

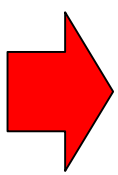
ポリモーフィズムによって、
「オブジェクト指向開発は変化に強い」
と言われる。

演習３：取締役を追加

・給料計算ルール

担当	基本給	が給料
主任	基本給 × 2 ＋ 1	が給料
部長	基本給 × 3	が給料
取締役	基本給 × 4 ＋ 2	が給料

取締役を追加して、
実行すると。



C:¥> javac Shacyo Torishimariyaku 100

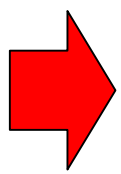
取締役がふんぞりかえって立ちました。
私の給料は402円です。

演習４：ボーナスはいくら

・ボーナス計算ルール

担当　基本給　×　3
主任　基本給　×　3
部長　基本給　×　3

ボーナスを答える能力を追加。
基本給は100円だよ。



C:¥> javac Shacyo Tanto 100

担当が普通に立ちました。
私の給料は100円です。
ボーナスは300円です。

C:¥> javac Shacyo Shunin 100

主任がすばやく立ちました。
私の給料は201円です。
ボーナスは300円です。

C:¥> javac Shacyo Bucho 100

部長がだるそうに立ちました。
私の給料は300円です。
ボーナスは300円です。

演習4：回答例

Shain.java

```
abstract public class Shain {
    public void setKihonkyu(int kihonkyu)
     {
        this.kihonkyu = kihonkyu;
    }
    public int getKihonkyu() {
        return kihonkyu;
    }
    public int getBonus(){
        return kihonkyu * 3;
    }
    abstract public void standup();
    abstract public int getKyuryo();
    private int kihonkyu;
}
```

interfaceではなく、abstract classに変更。
abstractはロジックを書けるinterfaceみたいなもの。

社員なら誰でも「ボーナスは基本給の3倍」という同じロジックなので、「社員」に1つだけボーナス計算ロジックを持たす。

Abstractで始まるメソッドは、「このクラスを継承したクラスは、このメソッドのロジックを書く必要があります。」という意味。

このクラスのデータらしき基本給を、引数で直接メソッドに渡さず、setKihonkyu()メソッドで社員クラスに保存。

演習4：回答例

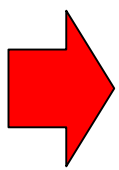
Tanto.java

```
public class Tanto extends Shain {
    public void standup(){
        System.out.println("担当が普通に
     起立しました。");
    }
    public int getKyuryo
        int kihonkyu = ge      onkyu();
        return kihonkyu;
    }
}
```

Shunin.java

```
public class Shunin extends Shain {
    public void standup(){
        System.out.println("主任がすばやく
     立ちました。");
    }
    public int getKyuryo(){
        int kihonkyu = getKihonkyu();
        return kihonkyu * 2 + 1;
    }
}
```

継承元がinterfaceではなく、abstruct classを用いる場合にはimplementsではなく、extendsを使う。



社員クラスで実装したメソッドを書かなくても、abstractで定義されたメソッドのみ書けばよい

演習4：回答例

Shacho.java

```
public class Shacho {
    public static void main(String[] args)
     {
       Shain shain = null;
       if(args[0].equals("Tanto")) {
          shain = new Tanto();
       }
       if(args[0].equals("Shunin")) {
          shain = new Shunin();
       }
       if(args[0].equals("Bucho")) {
          shain = new Bucho();
       }
       int kihonkyu =
      Integer.parseInt(args[1]);
       shain.setKihonkyu(kihonkyu);
       int kyuryo = shain.getKyuryo();
       shain.standup();
       System.out.println("私の給料は" +
      kyuryo + "円です。");
       System.out.println("ボーナスは" +
      shain.getBonus() + "円です。");
     }
}
```

演習5：ファクトリ

ShainFactory.java

```
public class ShainFactory {
    public Shain factory(String type){
        if(type.equals("Tanto")) {
            return new Tanto();
        }
        if(type.equals("Shunin")) {
            return new Shunin();
        }
        if(type.equals("Bucho")) {
            return new Bucho();
        }
        return new Tanto();
    }
}
```

Shacho.java

```
public class Shacho {
    public static void main(String[] args)
    {
        ShainFactory shainFactory = new
ShainFactory();
        Shain shain =
shainFactory.factory(args[0]);
        int kihonkyu = Integer.parseInt(args[1]);
        shain.setKihonkyu(kihonkyu);
        int kyuryo = shain.getKyuryo();
        shain.standup();
        System.out.println("私の給料は" + kyuryo
                                    ");
                                ut.println("ボーナスは" +
                                onus() + "円です。");
    }
```

担当クラスなどをnewしていたコードをShainFactoryクラスに置き換えたので,if文がなくなった。オブジェクトを生成して返すだけのクラスをファクトリと呼ぶ。

社員クラスは、、各社員を継承した役職をif文の中でnewしていたが、社員をnewするクラスにした。

まとめ

- 究極奥義
  - インターフェイス
  - ポリモーフィズム

  によって「オブジェクト脳」が芽生える。

- オブジェクト指向プロジェクトに参加するために必要なスキル
  - オブジェクト／クラス／メッセージ／継承／カプセル化／ポリモーフィズムの概念について、コンピュータの動作イメージを意識しない説明と、コードに置き換えた説明と、そのメリットについて説明ができること。
  - デザインパターンを初めから知らなくても、教えてもらえば理解できること。

- オブジェクト指向は、まず
  - オブジェクト指向は人間様のご都合で考えればいいのだ。コンピュータの仕様に合わせる必要はない。
  - ポリモーフィズム

  という2つのポイントを理解できればよい。

独断と偏見な付録1　日本語と英語のインピーダンスミスマッチ

●**英英辞書のススメ**

日本語と英語は1対1で対応しないし、翻訳できてもニュアンスが異なる場合がよくある。その場合英英辞書でしらべるとしっくりいくケースがある。

gorgeousの場合　　**英和　豪華な、(話) 楽しい、(話) すばらしい**

1.If you say that something is gorgeous, you mean that it gives you a lot of pleasure or is very attractive.
2.If you describe someone as gorgeous, you mean that you find them very sexually attractive.
3.If you describe things such as clothes and colors as gorgeous, you mean they are bright, rich, and impressive.

日本で一番よく使うゴージャスの意味とニュアンスが違うし、よくつかう意味の順番も意味も違う

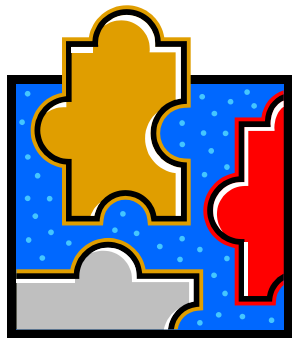
われわれオブジェクターは英語と日本語の変換で苦しんでるケースがある。用語がしっくり来ないときは、英英辞書を引いてみよう。

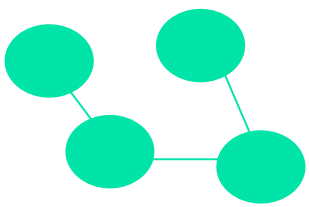
※この付録は翻訳者の方を批判したものではなく、もともと英語と日本語は変換できないということを言っています。それでも翻訳する人は本当にすごい人と思います。

独断と偏見な付録1　日本語と英語のインピーダンスミスマッチ

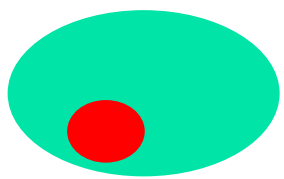
●High Cohesionパターン

オブジェクト指向設計では、凝集性はクラスの責務が殿程度強く関係し、収束されているかを示す基準です。関係性の高い責務を持ち、あまり多くの作業をしないクラスは高凝集性を持ちます。

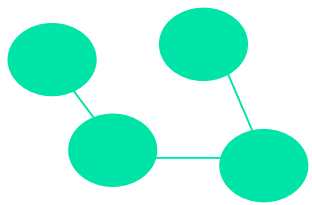




本来の意味のイメージ



日本語のイメージ（凝集度）



英語のイメージ

凝集
一箇所にこり集まること。
「全員のちからを一する」

If there is cohesion within a society, organization, or group, the different members fit together well and form a united while.

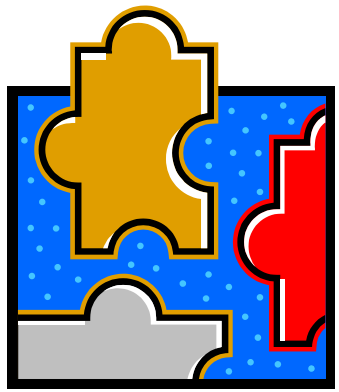
※言葉から受けるイメージは牛尾の独断と偏見だ。

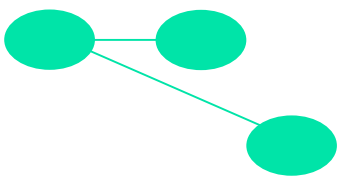
異なるメンバーがうまく協調して全体を構成する

独断と偏見な付録1　日本語と英語のインピーダンスミスマッチ

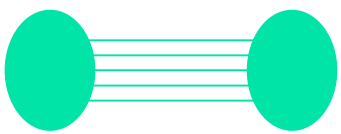
●Low Couplingパターン

結合は、1つのクラスが他のクラスに対し、どの程度の強さで
接続するか、把握するか、あるいは依存するかを表す尺度です。

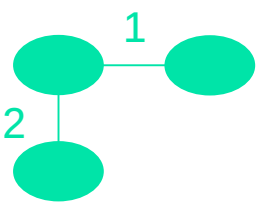




本来の意味のイメージ



日本語のイメージ（結合度）



英語のイメージ

結合
いくつかの物が結びついて1つになること。また、その結びつき。

If one piece of equipment is coupled to another, it is joined to it so that the two pieces of equipment work together.

Couplingの数が多いか少ないか？

独断と偏見な付録1　日本語と英語のインピーダンスミスマッチ

インターフェイスの継承は-ableが多い

●Responsiblity

某コーチング本によると
もともとRespons-ablityなので、反応できるという意味合いが強い。日本語だと責務･責任だが、これは失敗したら責められるニュアンスが強い

A responsiblitiy = an obligation to perform a task or know information

#obligation
#If you have an obligation to do something, it is your duty do that thing.
#duty
#If you say that something is your duty, you believe that you ought to do it because it is your responsiblitiy.

responsibility
If you have responsibility for something or someone, or if they are
your responsibility, it is your job or duty to deal with them and to
take decisions relating to them.

同じ意味でもニュアンスが自発的でポジティブ

同じ意味でもニュアンスが反応的でネガティブ

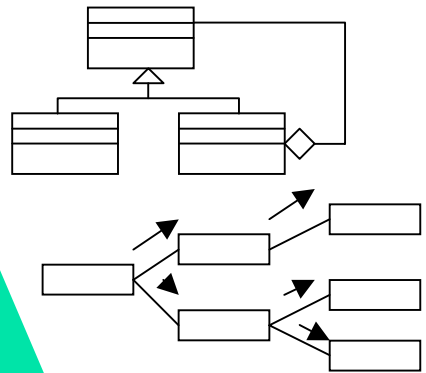
責務
自分の責任として果たさなければいけない事柄。つとめ
責任
自分が引き受けて行わなければならない任務。義務。
「ーを果たす。」「保護者としてのー」
自分がかかわった事柄や行為から生じた結果に対して
王義務や償い
「ーをとって辞任する」「だれのーでもない」
「ーの所在」

独断と偏見な付録2：クラス指向とオブジェクト指向

●今のオブジェクト指向のやり方はクラスのことばっかり考えているので、クラス指向じゃないか？重要なのはオブジェクト（インスタンス）では？という意見がある。

●なぜクラス指向という言葉がでてきたのか？（適当予想）

・Javaなどは結局クラスしか意識しない／できない
・Smalltalkなどはクラスとオブジェクトが明確だし、型もないのでオブジェクト感覚が強かった
・クラス中心の分析／設計スタイルが多かった。



●なぜクラス指向ではなくオブジェクト指向が重要か？

・結局動くのはインスタンスであるので。
・クラス指向はわかりくい。オブジェクト指向のほうがわかりやすい。
・オブジェクトとメッセージですべて話が済んで説明が簡単

●オブジェクト指向ではなく、クラス指向なのか？

・言語では結局クラスを書く。
・オブジェクトがあってもクラスへのマップ（抽象度の具合）が必要。

・教育資料はクラス指向色が強すぎるのが問題かな？
・オブジェクト指向ももっと活用すべし
・ほぼクラス指向派の人もいる
・どちらの観点も必要で使い分けるべし。

参考文献

- 「オブジェクト脳のつくり方」 翔泳社 著者 牛尾剛、監修長瀬嘉秀
- 「オブジェクト脳オンライン」 http://www.geocities.jp/objectbrain/
- 「コーチング・センスが身につくスキル」 著者 岸 栄光
- 「実践UML」 著者 クレーグ・ラーマン 訳 依田光江 監訳 今野睦＋依田智夫
- 「大人のための勉強法」 著者 和田秀樹
- 日立システムアンドサービス 英殿のプレゼンテーション（逆輸入）
- Collins COBUILD CD-ROM版
- Object Design Rebecca Wirfs-Brock and Alan McKean