プログラミング言語論

オブジェクト指向プログラミング言語

水野嘉明

目次

2

1. オブジェクト

3

再掲

1.1 オブジェクトとは

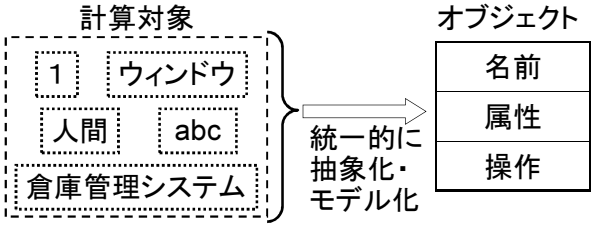
- *オブジェクト*（object)とは
  - 実際の「もの」を指す概念
  - プログラム上の計算対象を、抽象化・モデル化したもの
  - 自分自身の名前、属性（データ）、操作（メソッド）を持つもの

4

再掲

1.1 オブジェクトとは

- オブジェクトの概念



いろいろな「もの」がある

5

1.2 値型と参照型

- 原始的なデータ（例えば整数型int）をオブジェクトとすると、コストがかかる

- *値型*（value type）（プリミティブ型とも）
  - 非オブジェクト
    （整数、実数、文字、論理型等）
- *参照型*（reference type）
  - オブジェクト、参照により実装

6

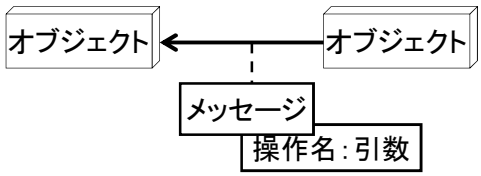
1.2 値型と参照型

- ボックス化機能
  - JavaやC#には、値型と参照型の自動変換機能がある
- ラッパークラス
  - 値型のデータに対応したオブジェクトが必要なとき、それを表すもの
    （例）int 型 → Integer クラス

7

1.3 オブジェクトの動作

- *メッセージ*（message）
  - オブジェクトの通信手段



8

1.3 オブジェクトの動作

- オブジェクトは、メッセージに対応した操作（＝*メソッド* method）を実行する
  - オブジェクトの別定義
    ＝「メッセージに反応するもの」

9

1.3 オブジェクトの動作

- Java でのメソッドの例

```
boolean order(Book book, int number)
  if (stock.isExists(book,number)) {
    stock.get(book,number);
    log.write(book,number)
    return true;
  } else {
    log.write(book, number, false);
    return  false;
  }
}
```

メッセージorderを受け取ったときの動作を定義

object.method(args) という形式

10

1.3 オブジェクトの動作

- 「書店」オブジェクト(?)が、order(注文)メッセージを受け取ると、上記 orderメソッドが実行される
- orderでは、最初に stock(在庫)オブジェクトに isExistsメッセージを送り、注文された本の在庫を調べる
- 在庫があれば・・・ (以下略)

11

2. オブジェクト指向とは

2.1 抽象データ型
2.2 オブジェクト指向言語の本質

12

2.1 抽象データ型

- *抽象データ型* とは
  - データと操作の両方の性質を併せ持つもの
  - データ構造の実装を隠蔽し、操作を抽象化する
  - オブジェクト指向の概念のベースの一つ

13

2.1 抽象データ型

- 例 「スタック」
  - スタックのデータ表現
  - データを格納する push操作
  - データを取出す pop操作

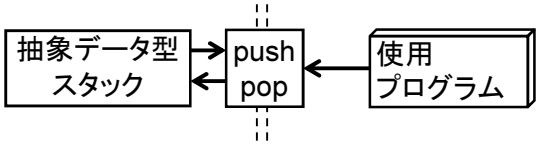
⇩

これらの実装を公開しない

14

2.1 抽象データ型

- 使用する側は、実装を気にせず、操作 push/pop を使用するだけ

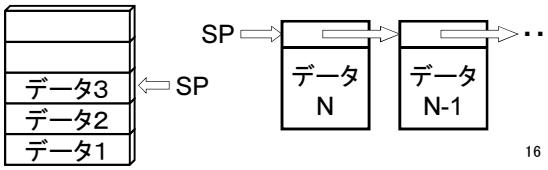


- ◆実装の変更がしやすい
- ◆誤操作を防ぎ、信頼性が高い

15

2.1 抽象データ型

- スタックの実装には、配列にする方法、リストを用いる方法等がある
- 使用するプログラム側では、push/pop 操作さえ出来れば問題ない



16

2.2 オブジェクト指向言語の本質

- 命令型言語からの、パラダイムシフト（考え方の転回）
  - システムの大規模化
  - 命令型言語での問題分割に限界

17

2.2 オブジェクト指向言語の本質

- 命令型言語では
  操作（機能・命令）を中心に問題を分割
  - 例：
    ユーザのログイン、データの新規作成、データベースへの登録、データベースの参照・変更 等

18

2.2 オブジェクト指向言語の本質

- オブジェクト指向言語では
  オブジェクトを中心に、クラスにより問題を分割
  - 例:
    ユーザクラス、データクラス、データベースクラスに分割し、各クラスの操作としてログイン、新規作成、登録などを用意

19

2.2 オブジェクト指向言語の本質

- オブジェクト指向の利点
  - 抽象データ型の考え方が、自然に導入できる
    - 命令型言語では、意識しなければできない
  - 名前解決が動的なため、弾力的で疎結合なプログラム構造となる

20

2.2 オブジェクト指向言語の本質

- オブジェクト指向言語の、本質的な機能
  ① オブジェクト定義(クラス定義)
  ② クラス間の関係記述
  ③ 動的メソッド検索とメッセージ通信

(オブジェクト指向言語ならば、必ず持っている機能)

21

2.2 オブジェクト指向言語の本質

① オブジェクト定義（クラス定義）
- 言語仕様に、クラスを定義する機能がある
- クラスには属性と操作がある
- クラス定義に従って、オブジェクトを生成する

22

2.2 オブジェクト指向言語の本質

② クラス間の関係記述
- 継承やインタフェースなどの関係を記述できる
- クラス間の関係記述は、差分プログラミング（後述）に必要な機能
  - 大規模なプログラム開発に必要

23

2.2 オブジェクト指向言語の本質

③動的メソッド検索とメッセージ通信
- メッセージ通信はオブジェクトの基本操作
- メッセージを受け取ったオブジェクトは、*動的メソッド検索* 機能で実行するメソッドを検索する（実行時にメソッドを検索する）

24

3. オブジェクト指向言語の詳細

3.1 カプセル化と情報隠蔽
3.2 クラス
3.3 差分プログラミング
3.4 継承
3.5 ポリモーフィズム
3.6 インタフェース

25

3.1 カプセル化と情報隠蔽

- *カプセル化* (encapsulation)
  - ➢データ構造と操作を一まとめにし、
  - ➢特定のインタフェースを通してのみ外部と通信できる

  ◆抽象データ型を実現
  ◆モジュール間の独立性を保証
    - 他のソースコードに依存しない

26

3.1 カプセル化と情報隠蔽

- *情報隠蔽* (information hiding)
  - ➢データ構造を使用者から隠し、必要な情報のみを公開する

  使用者側のプログラムは、提供側の実装に依存しない

情報隠蔽は、カプセル化が本質的に持つ性質である

27

3.1 カプセル化と情報隠蔽

- 情報隠蔽の利点
  - 利用者側プログラムは、提供者側プログラムの公開部分にだけ注目すればよい
  - 提供者は、非公開部分のプログラムは自由に変更できる

相互に、依存性が減少

28

3.1 カプセル化と情報隠蔽

- データや操作の公開範囲
  - 公開インタフェースと、プライベートインタフェースに二分される
  - 公開範囲を多段階に制御することも多い
    - 例：Javaは4種類のアクセス属性を持つ（public, protected, private, 指定なし）

29

3.1 カプセル化と情報隠蔽

- スタックの例（Java）

ここは情報隠蔽

```
class ArrayStack {
  private Object[] stack = new Object[100];
  private int      stackPointer = 0;
  public void push(Object element) {
    stck[stackPointer++] = element;
  }
  public Object pop() {
    return stack[--stackPointer];
  }
}
```

ここは情報公開

30

3.2 クラス

- クラス（class）とは
  - オブジェクトの「型」に相当
    ＝ ユーザ定義型
  - オブジェクト（インスタンス）を作成するための設計図、あるいはテンプレート（雛形）

31

3.2 クラス

- クラスは
  - オブジェクトの属性を抽象化した情報
    - 属性のタイプと初期値
    - アクセス制御情報　　など
  - インスタンス共通の属性
  - 操作（メソッド）

  などを持つ

32

3.2 クラス

- クラス定義の例（Java）

```
class TV {
  String power = "off";
  int    channel = 1;
  void  powerOn() {
     power = "on";
  }
  void  selectChannel (int  ch) {
     if (power.equals("on")) channel = ch;
  }
}
```

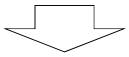
属性

操作

33

3.2 クラス

- オブジェクトは、動的な存在である



プログラムの実行時に、オブジェクトを生成・消去する

34

3.2 クラス

- *インスタンス* (instance) とは
  - クラスから生成されるオブジェクト
  - クラスの抽象部分である属性を具象化したもの

35

3.2 クラス

- インスタンス生成の例 (Java)

```
class  Client  {
  // メインメソッドの定義
  public static void main(String[ ] args) {
      TV  tv = new TV();
      tv.powerOn();
      tv.selectChannel(3);
           :
```

インスタンス生成

36

3.2 クラス

- *メソッド*（method）とは
  - オブジェクトに対する操作
  - メッセージをオブジェクトが受信すると、そのオブジェクトのクラスから動的メソッド検索を行って、実行するメソッドを見つける

（注：静的なメソッド呼び出しもある）

37

3.2 クラス

- インスタンス生成・開放時のメソッド
  - オブジェクトの宣言時に、初期値を設定するための初期化用関数
    ＝ *コンストラクタ*（constructor）
  - 生成したオブジェクトが不要になった時自動的に呼出されれ、後始末するための関数
    ＝ *デストラクタ*（destructor）

38

3.2 クラス

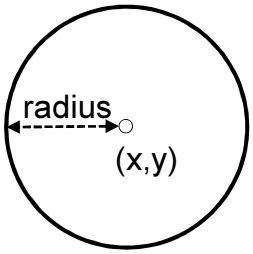
- どのようにクラスを設計するか
  - クラスとするか、値型や値型の配列のままで用いるか
  - どのような属性やメソッドを持たせるか

  - プログラムの効率、読解性、開発やメンテナンスのコスト、など

39

3.2 クラス

- 例: 円を表すクラス Circle



40

3.2 クラス

```
class  Circle {
   int  x;  int  y;   int  radius;  ・・・ }
```

```
class  Position {
   int  x;  int  y;   ・・・ }
class  Circle {
   Position  pos;  int  radius;  ・・・ }
```

41

演習 6.1

- 以下の動物を分類するためのクラスを作成せよ
  - イルカ、　ウシ、　　コウモリ、コブラ、　ダチョウ、ハト、ペンギン、ミツバチ、ライオン、ワニ

注: クラス分けの方法は、一通りではない。色々なクラスを考えることができる。

42

3.3 差分プログラミング

- *差分プログラミング*とは
  - ➢既存のプログラムに対する*差分のみを記述*するプログラミング
    - ●ベースとなる既存プログラムに対し、オブジェクトの属性や操作の 追加・削除・変更のみを記述することでプログラミングする

43

3.3 差分プログラミング

- 差分プログラミングのメリット
  - ➢差分のみの記述なので、記述する量が減少

    ⇩
    - ◆生産性の向上
    - ◆信頼性の向上

44

3.3 差分プログラミング

※「コピー＆ペースト手法」によるプログラミングは、差分プログラミングとは似て非なるもの

- ●バグの修正、機能の追加・変更に追従できない

「コピー＆ペースト手法」は悪い手段であり、使用しないこと

45

3.3 差分プログラミング

- オブジェクト指向言語では
  - 差分化するための機能が明示的に用意されている
    ⇒ 大規模システムを多人数で開発するのに便利
- 命令型言語では
  - ほとんど 不可能

46

3.3 差分プログラミング

- 以降は、差分プログラミングの具体的な手法をみていく

47

3.4 継承

- *継承* (inheritance) とは
  - あるクラスが別のクラスをもとにして作られ、その特性を引継いでいる時、その関係を*継承関係* と言う
  - 親のクラスを*スーパークラス* という
  - 子のクラスを*サブクラス* という

48

3.4 継承

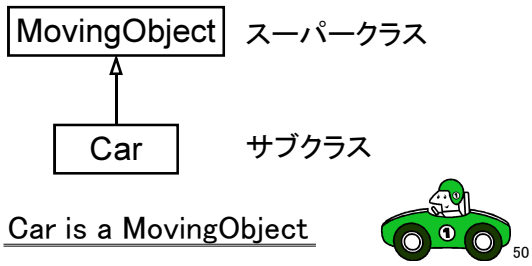
- 抽象／具象、汎化／特化、一般／特殊 の関係である
- 「 A is a B」（AはBである）の関係である

⇒ *"is_a"関係*（「である」関係）と呼ばれる

49

3.4 継承

- 継承関係の例



Car is a MovingObject

50

3.4 継承

- 継承による差分プログラミング
  - スーパークラスの属性やメソッドはサブクラスに引き継がれる（「親の物は子の物」）
  - サブクラスでは、スーパークラスに対する差分、属性やメソッドの追加・変更などを記述

51

3.4 継承

➢例 (スーパークラス)

```
class  MovingObject {
  int       velocity;    //速度
  Position  position;    //位置

  void  move(Position position) {
    this.position = position;
  }
  void  speedUp(int speed) {
    velocity = velocity + speed;
  }
  . . .
```

52

3.4 継承

➢例 (サブクラス)

スーパークラスを指定

```
class Car extends MovingObject {
  String  name;    //名前
  Engine  engine;  //エンジン

  void  startEngine() {
    engine.start();
  }
  void  accell() {
    this.speedUp(10);
  }
  . . .
```

属性やメソッドを追加

継承したメソッド

53

3.4 継承

- 継承の利点
  - ➢自然に差分プログラミングが可能
    ⇒ 生産性・信頼性が向上
- 継承の欠点
  - ➢カプセル化を弱める
    - スーパークラスの変更が、サブクラスに影響を与える

54

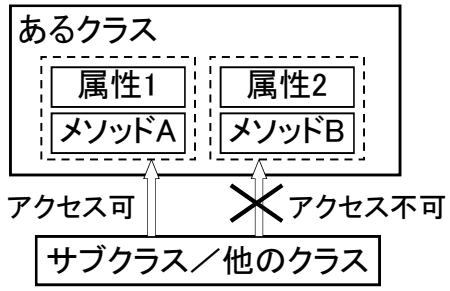
3.4 継承

- アクセス制御
  - どの属性やメソッドを、サブクラスや他のクラスからアクセス可能とし、どれをアクセス不可とするか
  - 言語により、アクセス制御は異なる

55

3.4 継承

- アクセス制御 概念図



56

3.4 継承

- Javaにおけるアクセス制御の例

アクセス修飾子	自ファイル			他ファイル	
	自クラス	サブクラス	他クラス	サブクラス	他クラス
public	○	○	○	○	○
protected	○	○	○	○	×
なし	○	○	○	×	×
private	○	×	×	×	×

57

3.4 継承

➢アクセス修飾子の使い方（再掲）

```
class ArrayStack {
  private Object[] stack = new Object[100];
  private int      stackPointer = 0;
  public void push(Object element) {
    stck[stackPointer++] = element;
  }
  public Object pop() {
    return stack[--stackPointer];
  }
}
```

58

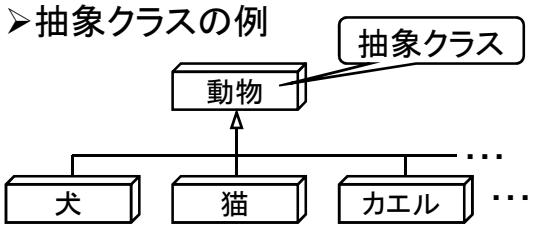
3.4 継承

- *抽象クラス*（abstract class）
  - ➢それ自身がインスタンスを作ることはなく、他のクラスに継承されるためだけに存在するクラス
  - ➢抽象クラスは、必ず継承してサブクラス化してやらなければならない
  - ➢インスタンスは生成できない

59

3.4 継承

➢抽象クラスの例



犬でも、猫でも、カエルでも・・・でもない抽象的な「動物」というインスタンスはありえない

60

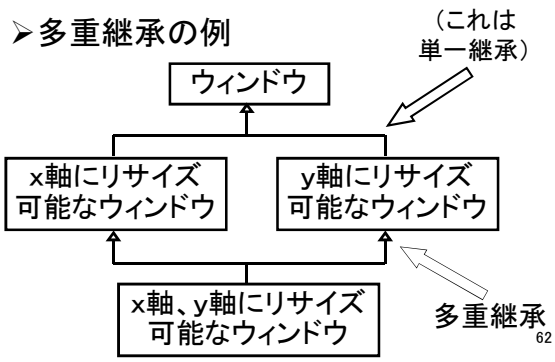
3.4 継承

- 単一継承と多重継承
  - ➢ *単一継承* (single inheritance)
    - 一つのスーパークラスだけを継承する
  - ➢ *多重継承* (multiple inheritance)
    - 二つ以上のスーパークラスを継承する

61

3.4 継承

➢多重継承の例



62

3.4 継承

- ➢単一継承では、
  - クラス設計がしにくい場合がある
    - 例：前記のウィンドウを、どのように設計するか？
- ➢多重継承では、
  - メソッド検索が複雑になる
    - 複数の親クラスに同一名のメソッドや属性があったらどうするか？

63

3.4 継承

- Java、C# は、単一継承のみ
  - 多重継承に代わる機能（インタフェースなど）がある
- C++ は、多重継承が可能

64

演習 6.2

- 次のプログラムの実行結果を答えよ

```
// Exercise 6.2
abstract class  A {
    public abstract void print();
}
class B extends  A {
    public void print() {
        System.out.println("Bメソッドの呼出し");
    }
}
```

（続く）

65

（続き）

```
class C extends  A {
    public void print() {
        System.out.println("Cメソッドの呼出し");
    }
}
public class  Ex6_2 {
    public static void main( String args[ ] ) {
        A  a = new B();
        a.print();
        a = new C();
        a.print();
    }
}
```

66

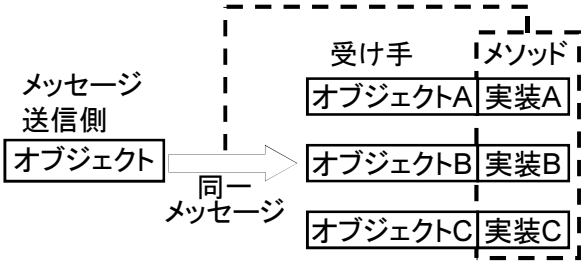
3.5 ポリモーフィズム

- *ポリモーフィズム* (polymorphism)とは
  - 異なるクラスの複数のオブジェクトに対し、同一のインタフェース(入出力規約)を与える
  - 同じメッセージを受け取っても、オブジェクトにより動作が異なる
  - 日本語では、「多態性」「多様性」「多相性」など

67

3.5 ポリモーフィズム

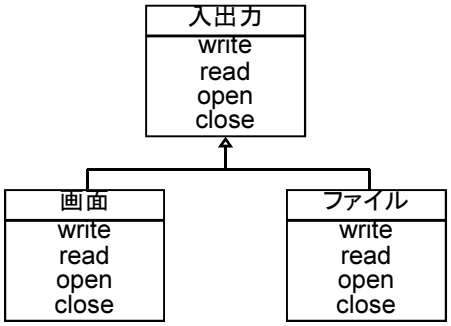
- ポリモーフィズム 概念図



68

3.5 ポリモーフィズム

- ポリモーフィズムの例 (1)



69

3.5 ポリモーフィズム

- クライアントプログラム

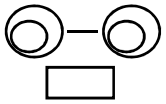
```
InOut  inout = newFile();
open(inout);
read(inout);
write(inout);
close(inout);
```

画面でもファイルでも、変更の必要がない

70

3.5 ポリモーフィズム

- 例(2) 図形
  - 線分、長方形、円などの基本的な図形を扱う問題
  - 「形状」は、一連の図形で構成



71

3.5 ポリモーフィズム

- 手続きによるプログラミング

```
void draw(figure f) {
  for (形状f中の図形a) {
    switch (a.kind) {
      case LINE:   線分用のコード;
      case RECT:   長方形用のコード;
      case CIRCLE: 円用のコード;
      default:     エラー処理;
    }
  }
}
```

3.5 ポリモーフィズム

- 新しい図形「矢印」を加える

⟶

手続きdraw は、矢印も描けるように case文が追加される

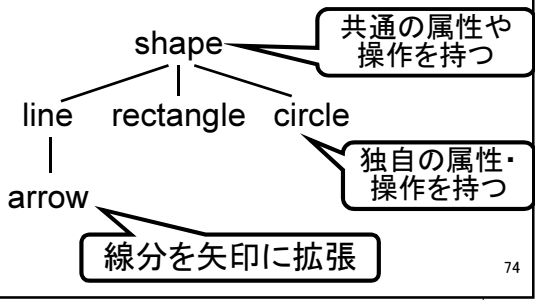
```
case ARROW:  矢印用のコード;
```

その他の図形に関わる手続きも修正される
⇒ 修正箇所が多数に分散

73

3.5 ポリモーフィズム

- 「継承」による差分プログラミング



74

3.5 ポリモーフィズム

- 各図形は、各々自分自身の draw 手続きを持つ

```
class arrow extends line {
        :
  void draw() {
     super.draw();
     矢印の頭部を描くコード;
  }
```

矢印を加えたならば、矢印のコードを作成する

75

3.5 ポリモーフィズム

- 利用する側は、形状を構成する図形をいちいち分類する必要がない

```
class figure {
      :
  void draw() {
    for (形状f中の図形a)
      a.draw();
  }
```

したがって、矢印を加えても修正する必要がない

76

3.6 インタフェース

- メソッドの設計は、規約の決定と実装という2つの段階を経る
  - 入出力規約とは、外部から見たメソッドの仕様。「何を入力したら、何を行い、何を出力するか」
  - 実装(implementation)は、メソッドを実現する手段。「どのように作るか」

77

3.6 インタフェース

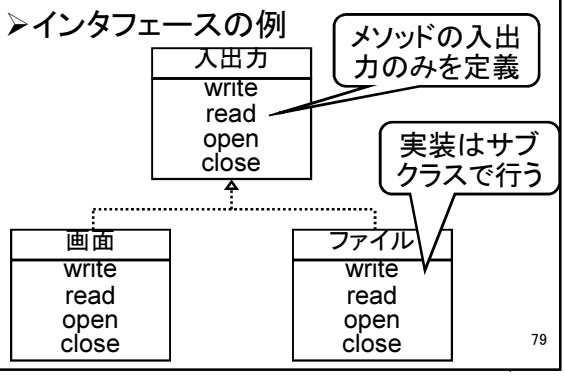
- *インタフェース* (interface) とは
  - 抽象クラスの一種
  - メソッドの入出力規約のみを取出した仕組み
  - メソッドの実装は、継承したクラスで行う

    ⇒ 様々な実装を同一のインタフェースで扱うことができる

78

3.6 インタフェース

- インタフェースの例



79

3.6 インタフェース

- インタフェースの特徴
  - （Java、C#でも）1つのクラスが複数のインタフェースを実装できる
  - Java、C#では、「interface」というキーワードで定義する
  - C++では、直接インタフェースは書けないが、純粋仮想関数という機能を使い実現できる
  - ″is implemented by″ 関係である

80

【付録】UML

- 統一モデリング言語
  （Unified Modeling Language）
  - システムのモデルを、図により表記するための記法
  - オブジェクト指向言語による開発時の設計に、よく使われる

81

【付録】UML

- OMG (Object Management Group) という組織が、仕様を作成・管理している
  - 現在の最新バージョンは、2.4.1
  - http://www.omg.org/ 参照

82

【付録】UML

- 十数種類の図(ダイアグラム)を、用途に応じて使い分ける
  - クラス図
  - ユースケース図
  - シーケンス図
  - ステートマシン図 (状態遷移図)
  - アクティビティ図　etc.

83

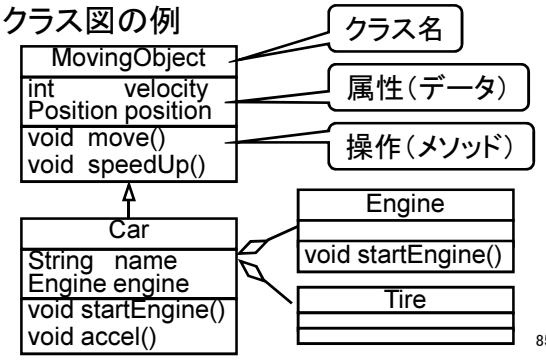
【付録】UML

- クラス図
  - クラスやクラス同士の関連を表す
    - クラス図は、UMLの中心的ダイアグラムである
  - クラスを、四角形で表わす
  - クラス同士の関係を、四角形を結ぶ各種の線分で表わす

84

【付録】UML

- クラス図の例



85

【付録】UML

- その他のダイアグラムの例を、Webサイトの
  【資料】UMLダイアグラム
  に掲載

86

お疲れさまでした