The Kyoto College of Graduate Studies for Informatics

kcg.edu

コンピュータプログラミング概論

秋期第８回　eラーニング資料

安　平勲

h_an@kcg.ac.jp

オブジェクト指向プログラミングとは？

Wikipediaより

- **オブジェクト指向**プログラミング（object-oriented programming，略語:OOP）は，オブジェクト指向に基づいたコンピュータプログラミング手法である
- **オブジェクト**は概ね**データ（変数，プロパティ）**と**コード（関数，メソッド）**の複合体を指す見解で一致しているが，その詳細の解釈は様々である
- オブジェクト指向のプログラムはこのオブジェクトの集合として組み立てられる事になるが，その実装スタイルもまた様々である

オブジェクト指向とは？

IT用語辞典より

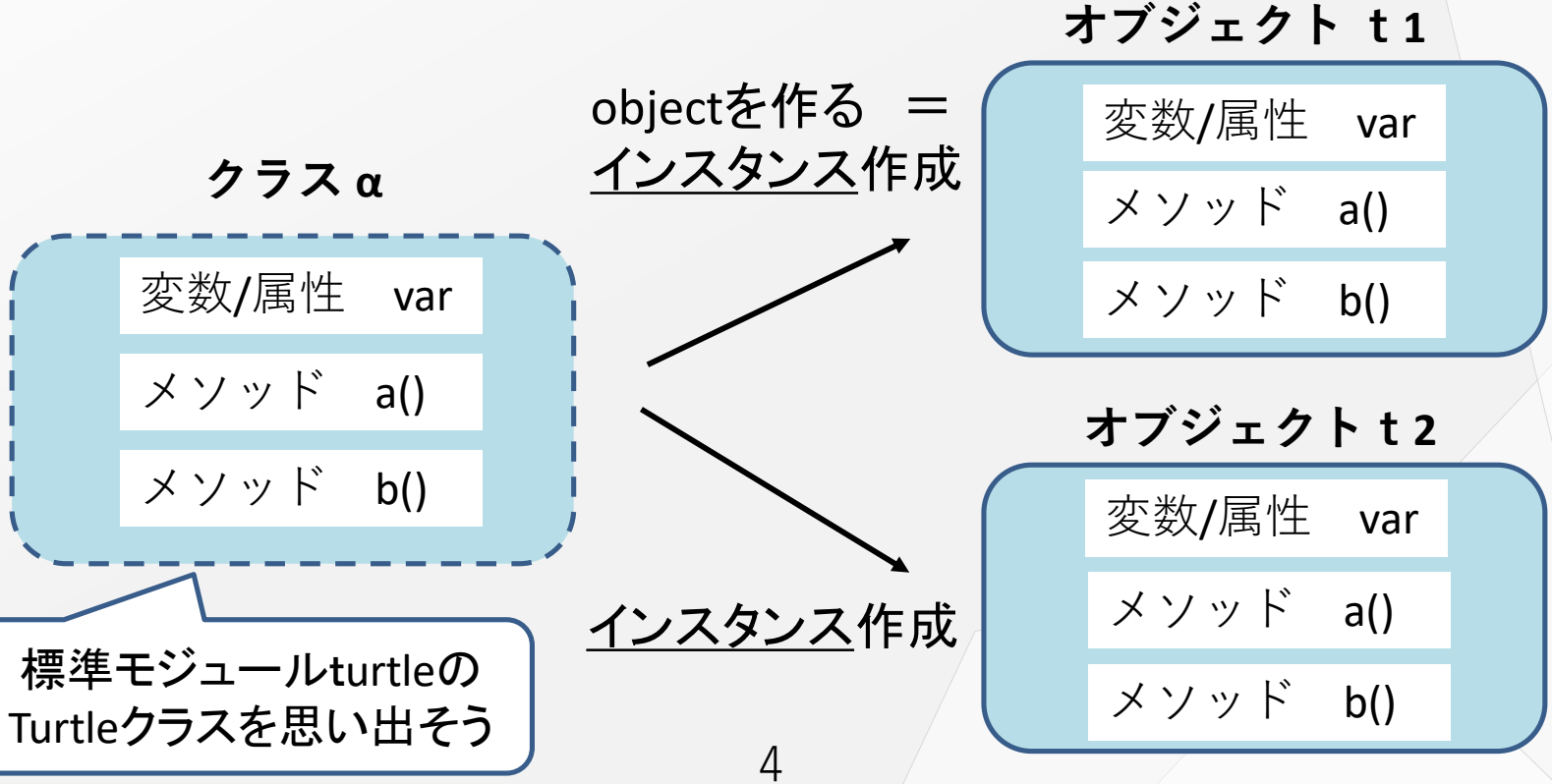
- システム全体を，現実世界の物理的なモノ（object）に見立てた「**オブジェクト**」と呼ばれる構成単位の組み合わせとして捉え，その振る舞いをオブジェクト間の相互作用として記述していく
- **オブジェクト**にはそれぞれ固有のデータ（**属性/プロパティ**）と手続き（**メソッド**）があり，外部からのメッセージを受けて**メソッド**を実行し，**データ**を操作する。オブジェクトに付随するデータの操作は原則としてすべてオブジェクト中のメソッドによって行われる

オブジェクト指向

- 1980年代に，それまでの**手続き型**言語（COBOL，Cなど）で書かれた**プログラムの保守性の低さ**に対する問題解決策の一つとして**オブジェクト指向**が提唱された
- 書籍『オブジェクト指向における**再利用**のためのデザインパターン』において，GoF (Gang of Four; 四人組) と呼ばれる共著者は，デザインパターンという用語を初めてソフトウェア開発に導入した
- C++やJavaの成功，GUIとの相性などから現在は主流
- キーワードは継承，カプセル化，ポリモーフィズム

classとインスタンスobject

- **クラス**classは**オブジェクト**objectを作る設計図



Pythonでのclass定義（自作）

- 『クラス名』と『メソッド定義』を書いて，クラスclassを定義する。
  このクラス定義がオブジェクトの設計図

```
class クラス名():
    def __init__(self, 仮引数):
        # self（=オブジェクト）の初期化など

    def メソッド名(self, 仮引数):
        # self（=オブジェクト）に対する処理
```

字下げ

コンストラクタ（クラスがオブジェクトを生成する際に必ず呼ばれる）

複数書ける

Pythonでのclass定義（自作）

- クラス名は大文字で始めるのがPython推奨
- **全てのメソッド定義**の第１引数に「メソッドの実行対象となるオブジェクト」を表す変数が**必須**

  ※ self という名前にするのが慣習
- 変数（属性）は __init__（コンストタクタ）の中で self （=オブジェクト）に代入して作る → self.*属性名 = 値/変数*
- 関数と同様に，クラス定義は分離できる。つまり，クラス定義だけを単独ファイル（モジュール）にして，import文で読み込んで使うことができる　※turtle.pyのTurtleクラス

自作のclass（1）

```
# Personクラスの定義
class Person():
    # __init__はインスタンス変数の初期化メソッド
    def __init__(self, name_arg):
        # 初期化メソッドの第１引数はインスタンスの名前を
        # 初期化メソッドの第２引数をオブジェクトのインス
        self.name = name_arg

teacher = Person('安　平勲')      # インスタンスを作成
student = Person('京都　太郎')    # インスタンスを作成
print(teacher.name, student.name) # 各インスタンスのname
print(teacher, student)
```

```
安　平勲 京都　太郎
<__main__.Person object at 0x000001B12BCB9588> <__main__.Person object at 0x000001B12BCB95C8>
```

- **class *クラス名*()**：で**クラス**定義
  ※クラス名の頭は大文字を推奨
- **def** で**メソッド**定義　※関数と同じ
- **def _ _init_ _(self, *引数*)** が初期化メソッド
- **`self. *変数*`**がインスタンスの変数

- ***Object名*＝*クラス*（*引数*）**でインスタンスObject作成。この時　**_ _init_ _*メソッド***が必ず呼ばれる
- ***Object名. 変数***でインスタンス変数にアクセスできる

自作のclass （2）

- インスタンス**変数**を追加

```
# Personクラスの定義
class Person():
    def __init__(self, name_arg, position_arg='生徒'):
        self.name = name_arg
        # インスタンス変数を追加
        self.position =  position_arg

teacher = Person('安　平勲','先生')  # positionを指定
student = Person('京都　太郎')       # positionを省略
print(teacher.name, teacher.position)
print(student.name, student.position)
```

```
安　平勲 先生
京都　太郎 生徒
```

- ➢ インスタンス変数にposition追加
- ➢ メソッドの引数に省略値を指定できる

- ✓ 同じクラスから異なる属性（変数）のインスタンスオブジェクトを作成（turtleのt1とt2）

自作のclass （3）

```
# Personクラスの定義
class Person():
    def __init__(self, name_arg, position_arg='生徒'):
        self.name = name_arg
        self.position =  position_arg
        self.total = 0          # インスタンス変数を追加
    def practice(self, score): # メソッドを定義
        self.total += score

student = Person('京都　太郎')
print(student.name, student.position)
student.practice(10)            # メソッド呼出し
print(student.total)
student.practice(9)
print(student.total)
student.practice(1)
print(student.total)
```

```
京都　太郎 生徒
10
19
20
```

- **メソッド**を追加

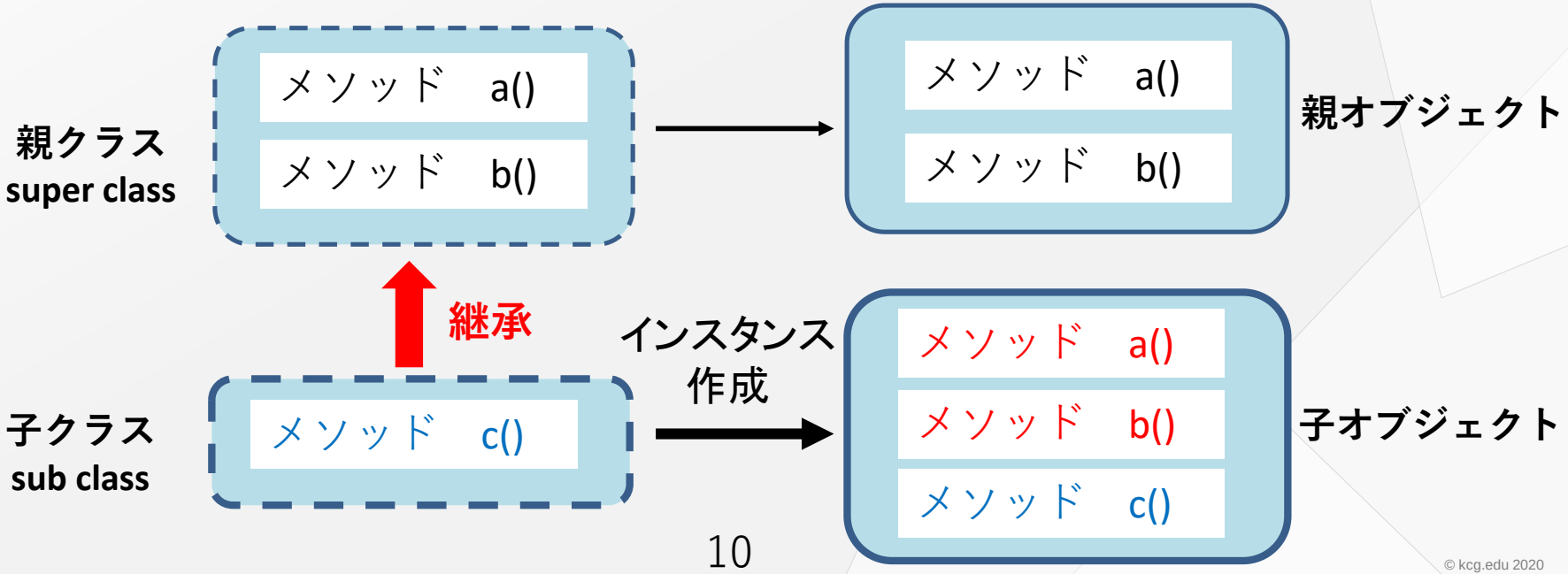
➢ **def *メソッド*（self, *仮引数*）:**

➢ ***Object*名.*メソッド*（*引数*）で呼出し**

✓ studentオブジェクトのpracticeメソッドを呼びだす度に，仮引数scoreがインスタンス変数totalに加算される
⇒オブジェクトは内部状態を持てる　※関数との違い

継承

- 親（super）の変数・メソッドを子（sub）が受け継ぐ
- プログラムを機能拡張する場合，既存（親オブジェクト）を改修することなく，新規（子オブジェクト）に機能追加できる



継承

```
# 親クラスの定義
class Parent():
    def call(self):
        print('親クラスのcall')

# Parentの子クラスの定義
class Children(Parent):
    def call2(self):
        print('子クラスのcall2')

parent = Parent()      # 親オブジェクト作成
parent.call()
children = Children() # 子オブジェクト作成
children.call2()
children.call()        # 親クラスのメソッドを継承
```

```
親クラスのcall
子クラスのcall2
親クラスのcall
```

- class *子クラス名*（*親クラス名*）：

✓ 子クラスは親の**メソッド**を**継承**

継承

```
# 親クラスの定義
class Greet():
    def hello(self):
        print("はい！")

# 子クラスの定義
class Greet2(Greet):
    # 親クラスのメソッドをオーバーライドする
    def hello(self, name = None):
        if name :
            print(name + "さん、こんにちは！")
        else :
            super().hello()     # 親クラスのhello()をそのまま使う

greeting2 = Greet2()
greeting2.hello()
greeting2.hello('安')
```

```
はい！
安さん、こんにちは！
```

- 親のメソッドを子のメソッドで追記できる　※メソッド名が同じ
  ⇒オーバーライド（上書き）と言う

- ✓ **super().*method*()**で親のメソッドを呼べる

カプセル化（隠蔽）

- 『隠蔽』とも呼ばれ，オブジェクト内変数を外部から見せない仕組み
- インスタンス変数やグローバル変数など，オブジェクトの振る舞いが外部から操作される（誤動作する）ことを防ぐのが目的（保守性向上）

```
class Person():
    def __init__(self, name_arg, position_arg='生徒'):
        self.name = name_arg
        self.position =  position_arg

student = Person('京都　太郎')
print(student.position)
student.position = '先生' # positionの書き換え
print(student.position)
```

```
生徒
先生
```

✓ **Python**ではインスタンス変数の**書き換え**が***外部からできてしまう***　⇒　非公開化は次頁

カプセル化； Pythonでの非公開化

- 隠蔽したいインスタンス変数名の前に『_ _』を付ける
  ※名前のマングリング（ぐちゃぐちゃにする）という

```
class Person():
    def __init__(self, name_arg, position_arg='生徒'):
        self.__name = name_arg          #  変数名をマングリング
        self.__position =  position_arg
    def who(self):
        print(self.__name + 'です')

student = Person('京都　太郎')
student.who()
print(student.__position)          #  マングリング名は
```

```
京都　太郎です
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-10-381ac1b4e4f3> in <module>
      9 student = Person('京都　太郎')
     10 student.who()
---> 11 print(student.__position)          #  マングリング名はアクセス不可

AttributeError: 'Person' object has no attribute '__position'
```

- ➢ 変数名をマングリング；　*__名前*

- ✓ クラス内では変数にアクセス可能
- ✓ 外部からマングリング変数にアクセス不可

ポリモーフィズム

- 『ポリモーフィズム』とはオブジェクト指向プログラミングの概念の一つ。日本語では『多態性』と訳されることが多い
- Pythonでは『ダックタイピング』と呼ばれる

```
class animal():
    def __init__(self, name):
        self.name = name

class Duck(animal):
    def voice(self):
        print("ガーガー")
    def walking(self):
        print("アヒルがお尻をフリフリ歩きます。")

duck = Duck('duck1')
print(duck.name)
duck.voice()
duck.walking()
```

```
duck1
ガーガー
アヒルがお尻をフリフリ歩きます。
```

- ✓ ”If it walks like a duck and quacks like a duck, it must be a duck”
  （もしもアヒルのように歩き，アヒルのように鳴くなら，それはアヒルである）

ポリモーフィズム

```
class animal():
    def __init__(self, name):
        self.name = name

class Duck(animal):
    def voice(self):
        print("ガーガー")
    def walking(self):
        print("アヒルがお尻をフリフリ歩きます。")

class Elephant(animal):       #象クラスを追加
    def voice(self):
        print("パオーン")
    def walking(self):
        print("象がゆったり歩きます。")

def animal_ability(animal): #animalオブジェクトが引数の関数
    animal.voice()          #引数オブジェクトのvoiceメソッドを呼ぶ
    animal.walking()        #引数オブジェクトのvwalkingメソッドを呼ぶ

duck = Duck('duck1')
elephant = Elephant('elephant1')
animal_ability(duck)        # ダックのvoiceとwalkingメソッドを呼ぶ
animal_ability(elephant)    # 象のvoiceとwalkingメソッドを呼ぶ
```

ガーガー
アヒルがお尻をフリフリ歩きます。
パオーン
象がゆったり歩きます。

- animal_abilityはオブジェクトを引数にする関数。引数オブジェクトのメソッドを定義

✓ オブジェクトの持つメソッドを関数で呼べる

ポリモーフィズム

```
    def walking(self):
        print("アヒルがお尻をフリフリ歩きます。")
class Elephant(animal):
    def voice(self):
        print("パオーン")
    def walking(self):
        print("象がゆったり歩きます。")

class Person():#animalと関係ないクラス
    def voice(self):
        print("話せる")#animalとメソッドは同じ
    def walking(self):
        print("自転車に乗れる")

def animal_ability(animal): #animalオブジェクトが引数の関数
    animal.voice()          #引数オブジェクトのvoiceメソッドを呼ぶ
    animal.walking()        #引数オブジェクトのvwalkingメソッドを呼ぶ

duck = Duck('duck1')
elephant = Elephant('elephant1')
person = Person()
animal_ability(duck)
animal_ability(elephant)
animal_ability(person)    # Personのvoiceとwalkingメソッドを
```

```
ガーガー
アヒルがお尻をフリフリ歩きます。
パオーン
象がゆったり歩きます。
話せる
自転車に乗れる
```

- ➢ 同じメソッド名を持つ無関係のクラスを定義

- ✓ 関数animal_abilityで異なるオブジェクトのメソッドが呼べる＝ポリモーフィズム
  ↓
- ✓ 呼出し側はオブジェクト（クラス）の違いが判らない。メソッド名が一致すればOK
  ↓
- ✓ 機能拡張に効果←呼出し側のテスト不要