bianca.phelan@tudublin.ie

O Object
O Oriented
P Programming

S1 = Python with Bianca

S2 = Java with Bryan

DT228(TU856)/DT282(TU858) - 2

COMPUTER SCIENCE

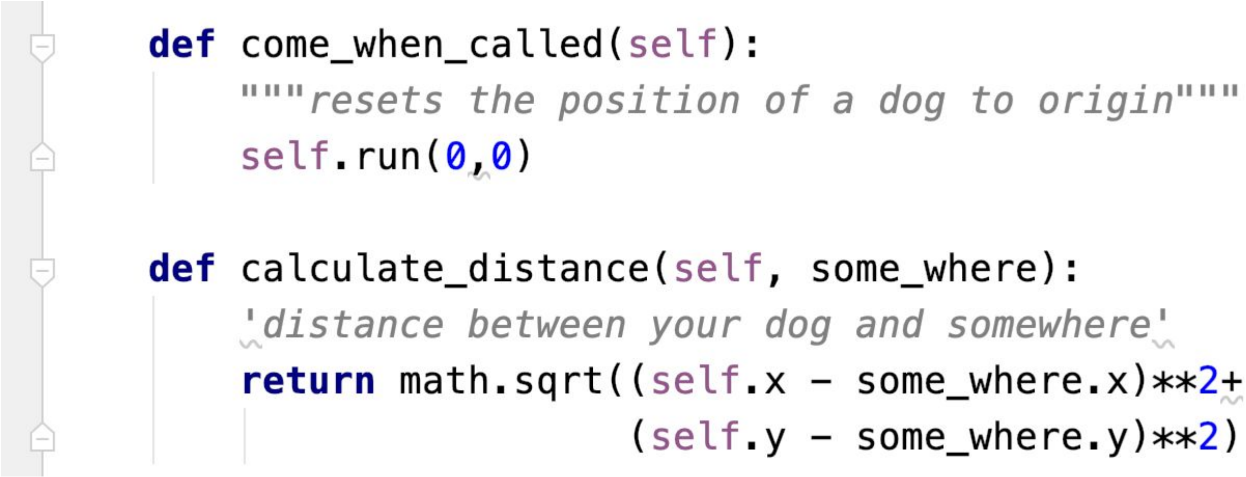
Objectives

- Documenting code
- Understand objects and OOP principles
- Design objects and classes in Python
- First design steps with UML

Documenting Code

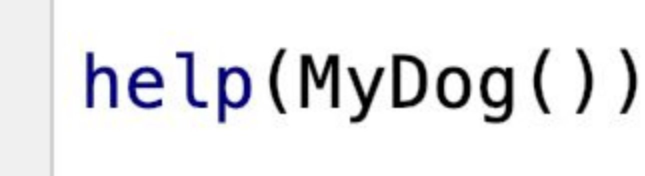
Writing API Documentation

- Explain what you are doing
- Python uses something called docstrings
- Not separate but a mechanism right in your code
- Can be lengthy, style guide recommends that a line width should not extend 80 characters

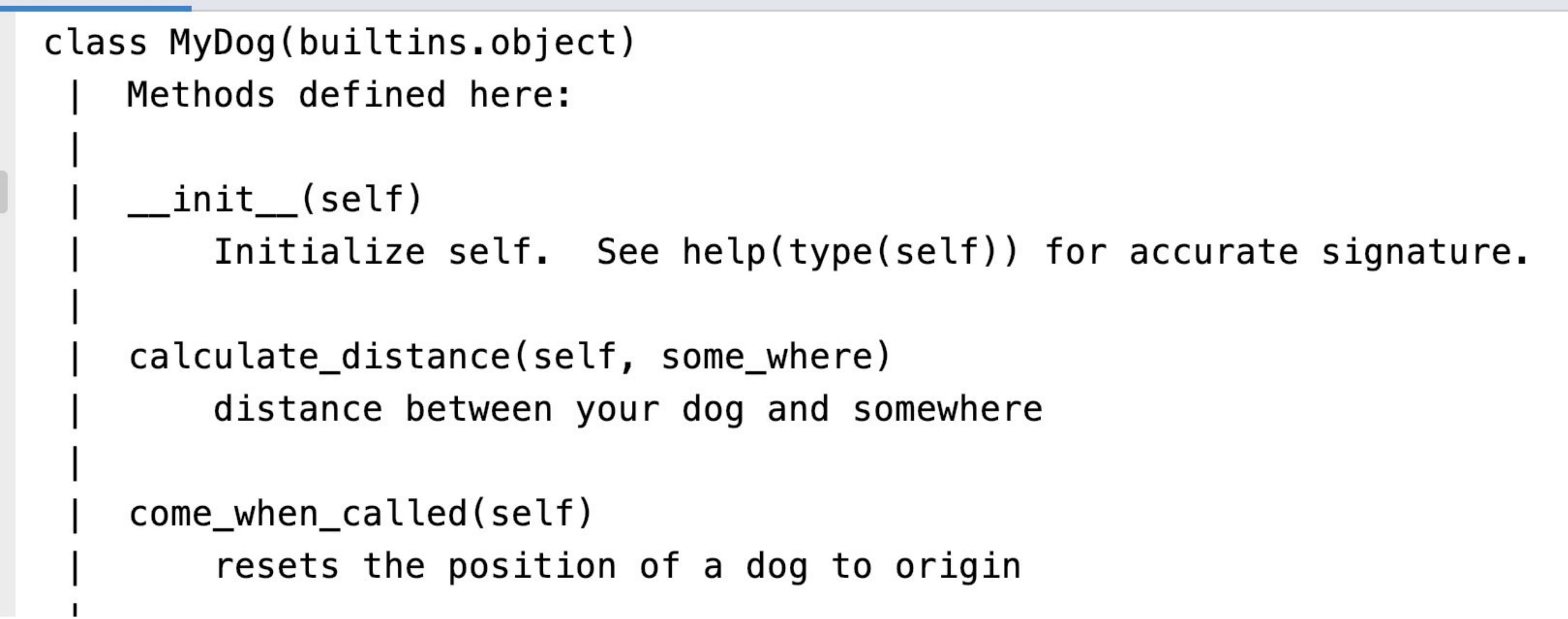


```
def come_when_called(self):
    """resets the position of a dog to origin"""
    self.run(0,0)

def calculate_distance(self, some_where):
    'distance between your dog and somewhere'
    return math.sqrt((self.x - some_where.x)**2+
                     (self.y - some_where.y)**2)
```

API doc



```
help(MyDog())
```



```
class MyDog(builtins.object)
 |  Methods defined here:
 |
 |  __init__(self)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  calculate_distance(self, some_where)
 |      distance between your dog and somewhere
 |
 |  come_when_called(self)
 |      resets the position of a dog to origin
 |
```

OOP Principles and Design

Object Basic Principle

- Look at the real world:
  - Your dog
  - Your desk lamp
  - Your tv
- All have a **state** and a **behaviour**
  - Example dog:
  - State: breed, size, colour, name
  - Behaviour: bark, play fetch, go walkies

- Some objects are more complex than other objects
- Some objects contain other objects.

[1]

A Software Object is Similar

- We also have a state and a behavior
- State is stored in variables
- Behaviour is exposed by methods/functions
  - Methods operate on an object’s internal state
  - Primary way of communication among objects

Encapsulation: Hides the object’s state and requires that all interactions with the object are performed via messages.

[1]

Example: Bicycle

- **State**: current speed, current gear
- **Behaviour**: methods to change speed, change gear
- For example, if the bicycle object only has 6 values for gear, the method would reject a value outside this range

The object remains in control of how the outside world is allowed to use it.

[1]

What is a Class

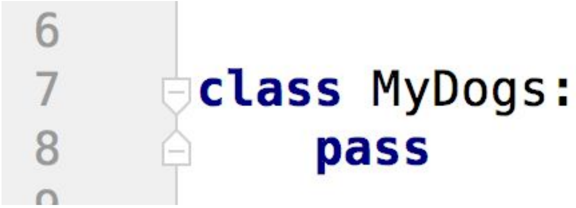
- A class is a blueprint from which individual objects (or instances of the class of object) can be created
- In the real world we often find many individual objects of the same kind, for example many poodles, many bicycles of the same make
    - All share same building blocks
    - Your particular poodle is an instance of the class of objects known as poodle

[1]

Code Examples

- Actually, although not explicitly named like this, everything in Python is a class
- Class consists of 2 parts: a header and a body
    - Class name can be followed by other class names, this means it inherits from those.
- Body is indented list of statements
- Class name must start with a letter or an underscore,
    - the name can only contain letters, underscores and numbers

Indentation uses 4 spaces (tab).

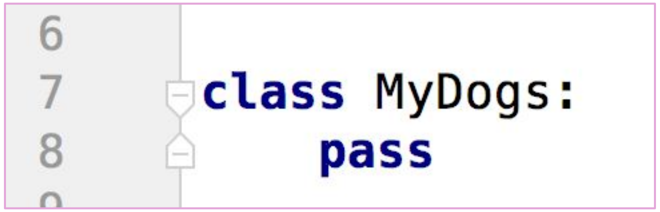


```
class MyDogs:
    pass
```

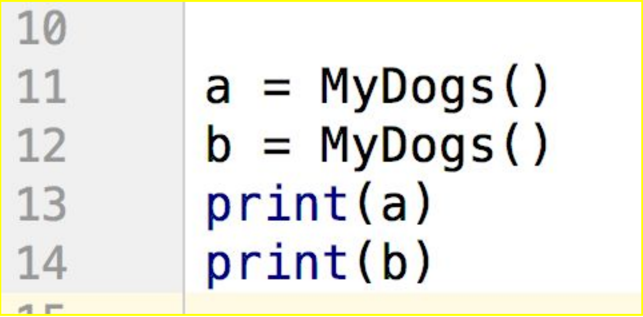
Python style guide: search online for PEP8, recommends camel case naming for classes and _ for methods.

[5]

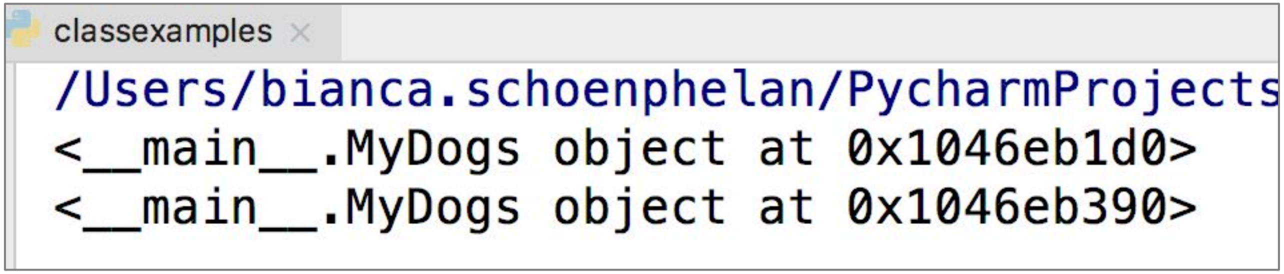
Using the Example Class



```
class MyDogs:
    pass
```



```
a = MyDogs()
b = MyDogs()
print(a)
print(b)
```

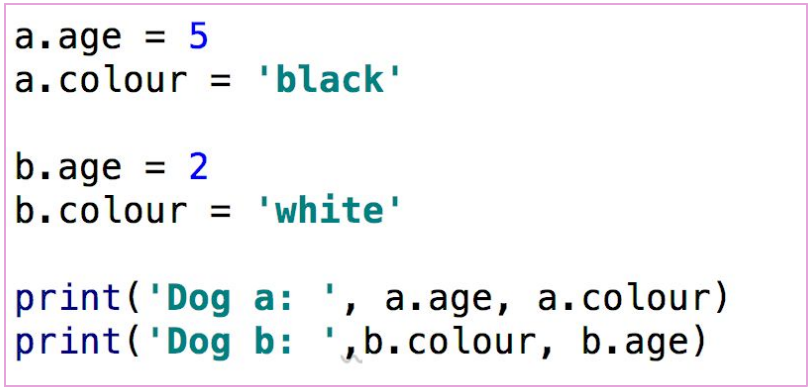


```
/Users/bianca.schoenphelan/PycharmProjects
<__main__.MyDogs object at 0x1046eb1d0>
<__main__.MyDogs object at 0x1046eb390>
```

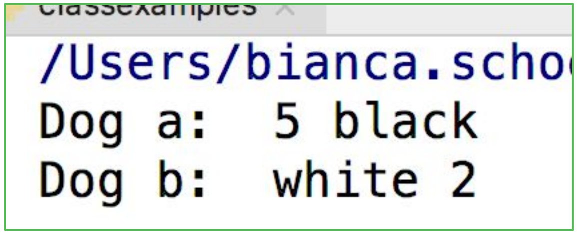
- Two new objects a and b have been instantiated from the class MyDogs()
- Looks like a function call, Python knows what to do

[5]

Now with Attributes



```
a.age = 5
a.colour = 'black'

b.age = 2
b.colour = 'white'

print('Dog a: ', a.age, a.colour)
print('Dog b: ',b.colour, b.age)
```

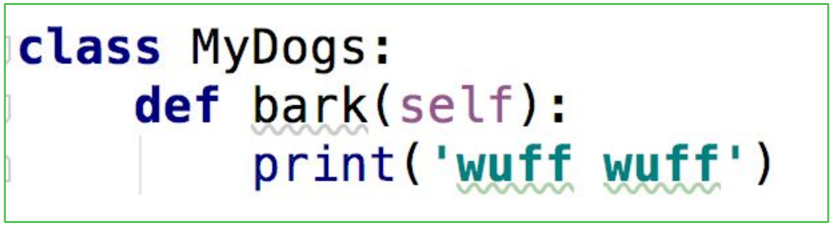


```
/Users/bianca.scho
Dog a:  5 black
Dog b:  white 2
```

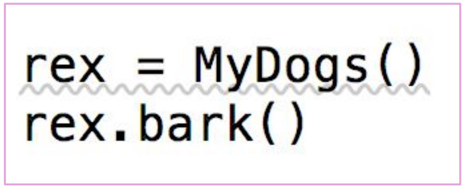
- First empty class
- Then class with attributes
- Dot notation:
  - <object>.<attribute>=<value>
- Value can be many things, a python primitive, a built in data type, another object, even a function or another class

[5]

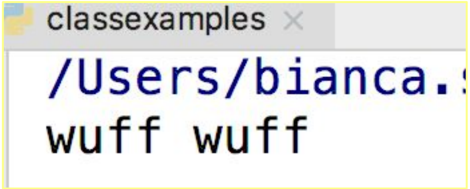
Behaviours



```
class MyDogs:
    def bark(self):
        print('wuff wuff')
```



```
rex = MyDogs()
rex.bark()
```



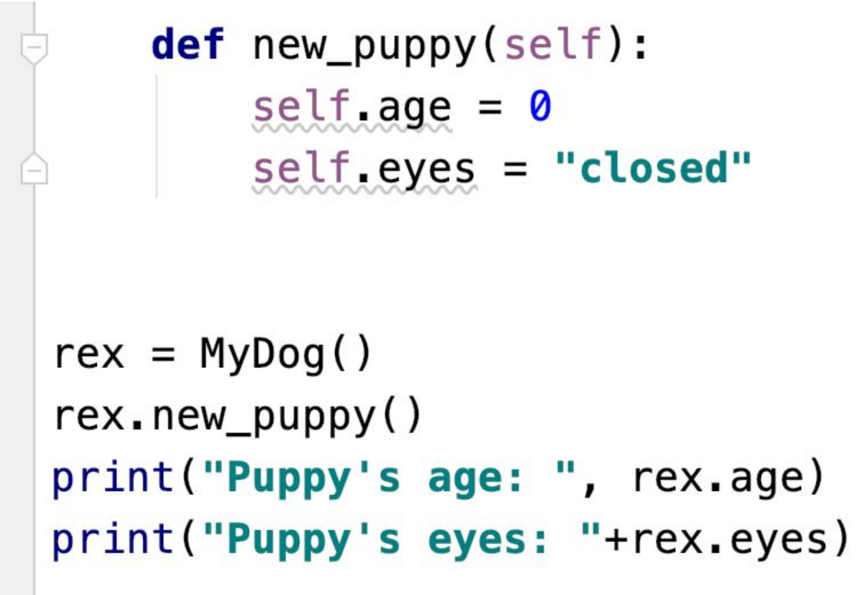
```
/Users/bianca.
wuff wuff
```

- Starts with keyword def followed by space and the the name of the function
- Parentheses for parameter list
- Terminated by colon :
- Next line is indented to contain the statements of the method

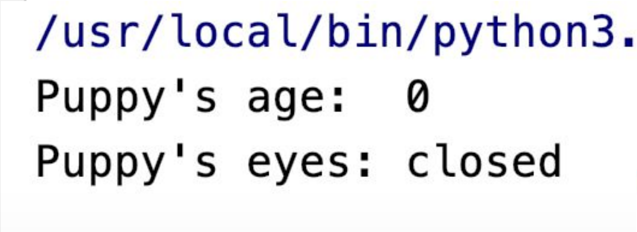
All methods have one required argument: `self`

- This is a convention, but never seen anyone not use it
- It’s a reference to the object that the method is being invoked on

Behaviours cont’d

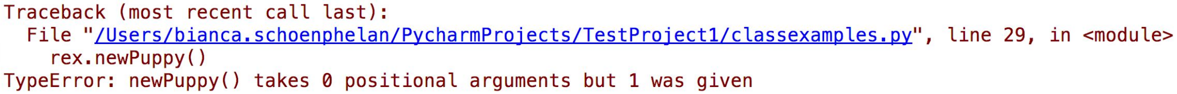


```
    def new_puppy(self):
        self.age = 0
        self.eyes = "closed"


rex = MyDog()
rex.new_puppy()
print("Puppy's age: ", rex.age)
print("Puppy's eyes: "+rex.eyes)
```



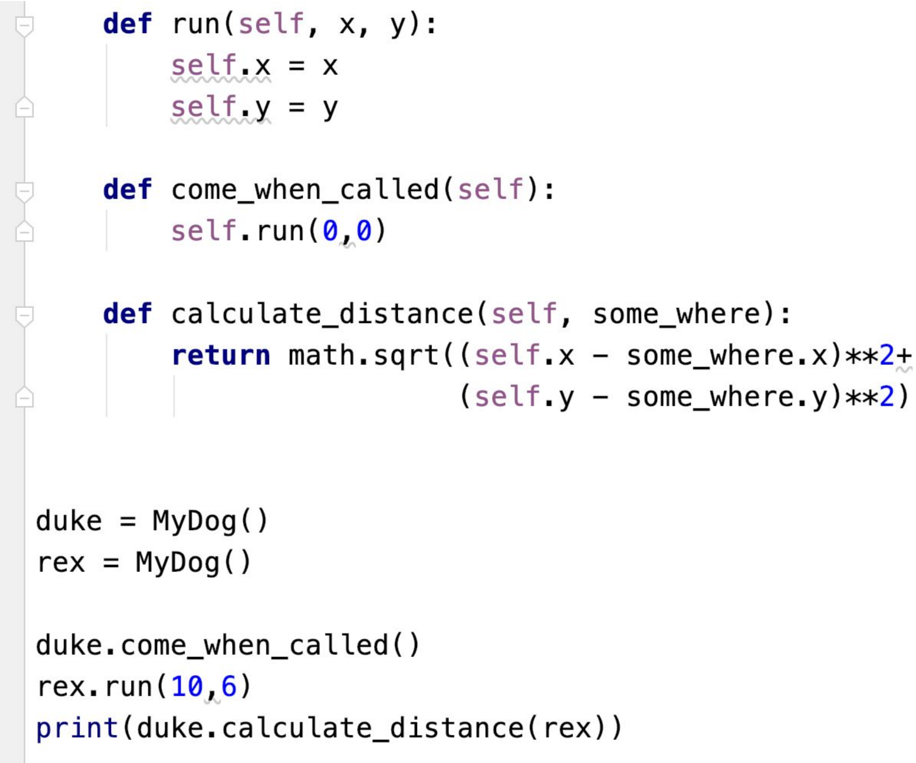
```
/usr/local/bin/python3.
Puppy's age:  0
Puppy's eyes: closed
```

- Usage of self
- Access to attributes and methods
- Notice that when we call the method newPuppy we do not have to pass an argument!
    - Python knows what we want to do
- If you forget about self in the method declaration you get an error message

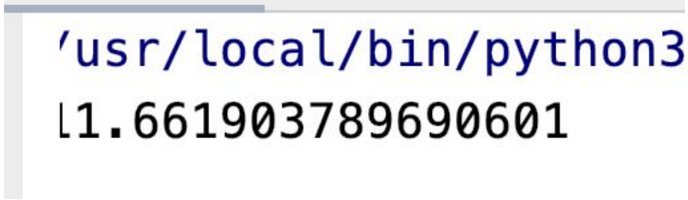


```
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/PycharmProjects/TestProject1/classexamples.py", line 29, in <module>
    rex.newPuppy()
TypeError: newPuppy() takes 0 positional arguments but 1 was given
```

More Arguments



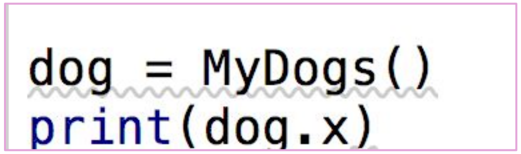
```python
    def run(self, x, y):
        self.x = x
        self.y = y

    def come_when_called(self):
        self.run(0,0)

    def calculate_distance(self, some_where):
        return math.sqrt((self.x - some_where.x)**2+
                         (self.y - some_where.y)**2)


duke = MyDog()
rex = MyDog()

duke.come_when_called()
rex.run(10,6)
print(duke.calculate_distance(rex))
```



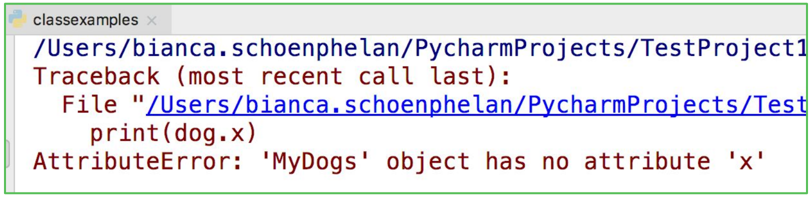
```
/usr/local/bin/python3
11.661903789690601
```

- **run** method accepts two arguments (plus self)
- **come_when_called** calls run and puts location points to origin
- **distance** calculation Pythagorean theorem for distance between two points
- To call use dot notation and place arguments into parentheses

Initialising Objects



```
dog = MyDogs()
print(dog.x)
```



```
/Users/bianca.schoenphelan/PycharmProjects/TestProject1
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/PycharmProjects/Test
    print(dog.x)
AttributeError: 'MyDogs' object has no attribute 'x'
```

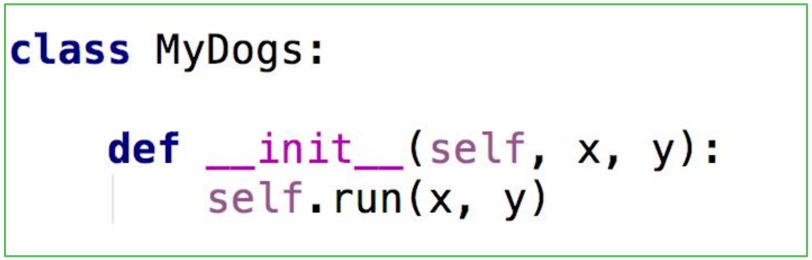
- What if we never set the position anywhere but try to use it?
- Try it out!
  - Great method of teaching yourself programming stuff
- Attribute error: useful!

Creating attributes in this way can get very messy, very fast!

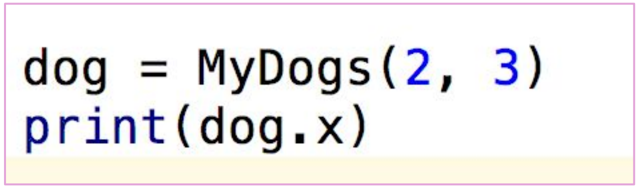
Initialising Objects cont’d

- Most OOP languages have a constructor
- Python has a constructor and an initializer
    - Constructor is rarely used
    - Initializer: __init__
        - Leading and trailing double underscore means that the Python interpreter will treat this in a special way

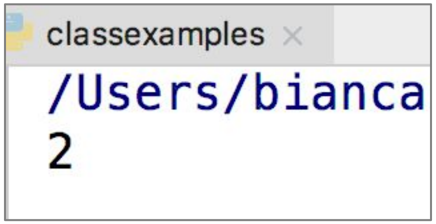
How to Initialise



```
class MyDogs:

    def __init__(self, x, y):
        self.run(x, y)
```

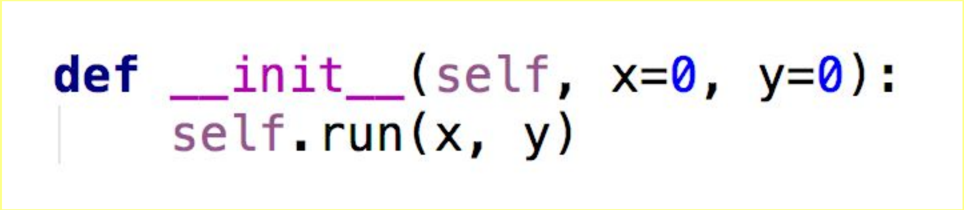


```
dog = MyDogs(2, 3)
print(dog.x)
```



```
/Users/bianca
2
```

- If you try to construct an object of MyDogs without an argument now, you can an argument error, similar to before
- Solution: use default value which gives you a choice



```
def __init__(self, x=0, y=0):
    self.run(x, y)
```

Advantages of OOP

- Modularity
  - Independently write source code for several objects
  - Once an object has been created it’s easy to pass it around the system
- Information-hiding
- Code re-use
- Plugability
- Ease of debugging

[1]

Advantages of OOP

- Modularity
- Information-hiding
  - Interaction happens strictly through methods (well, in most OOP, Python wants you to be a responsible programmer instead)
  - In Python we often "hide" the name by using __ but you can still find it if you are determined (name mangling, we'll get to it)
  - Details of an object remain hidden
- Code re-use
- Plugability
- Ease of debugging

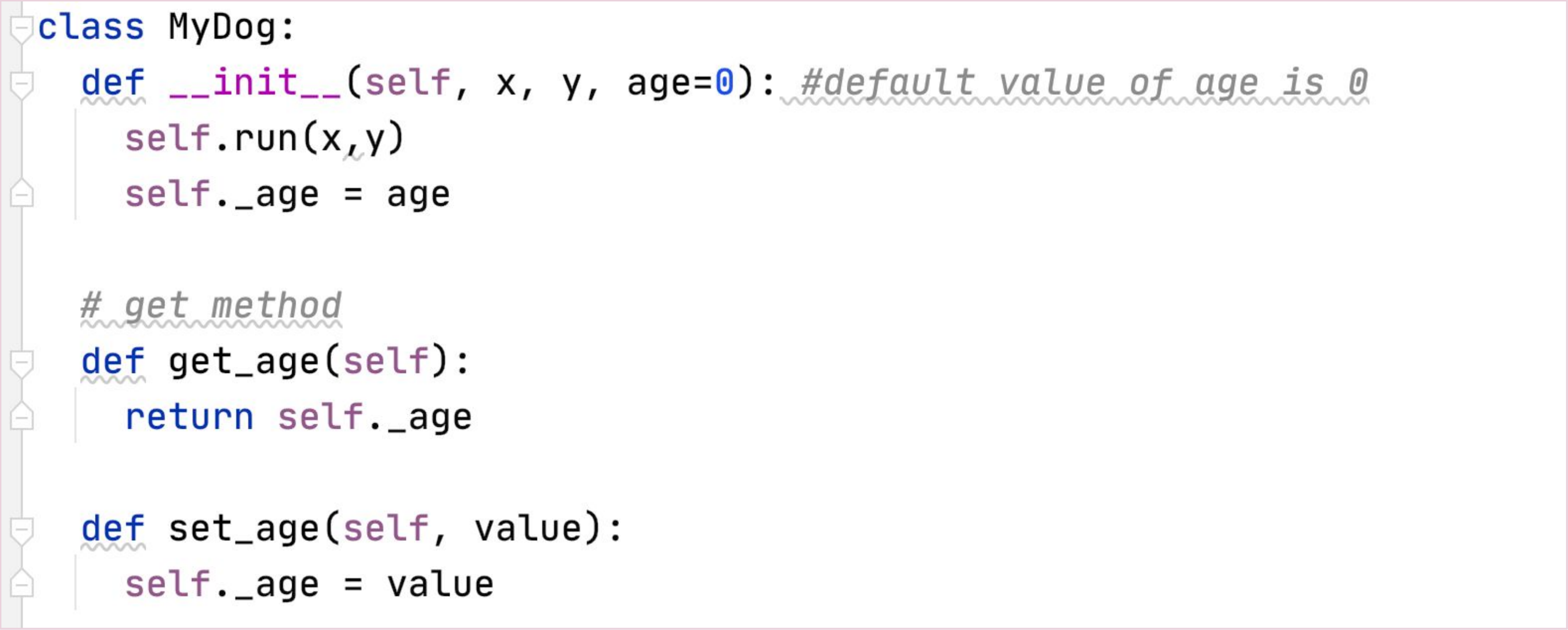
[1]

Getter and Setter Methods

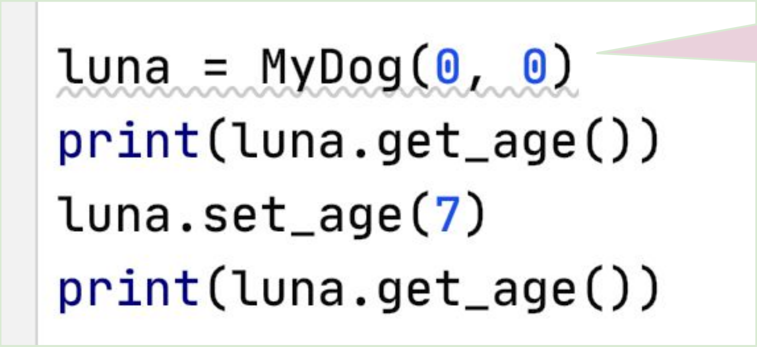
- Proper OOP design
  - Ensures data encapsulation
- Getter method to get a variable
- Setter method to set a variable (other than at initialisation)
- In other languages variables can be hidden from outside access, not so in Python
  - We often implement them anyway, for example for validation when variables are set (for example: right type? Right value range?)
  - To indicate that we would like to avoid direct access to variable fields
  - In Python implemented by putting an underscore _ in front of the name

Not the same in Python as in other OOP languages.

Example Get() and Set() Methods in Python



```python
class MyDog:
  def __init__(self, x, y, age=0): #default value of age is 0
    self.run(x,y)
    self._age = age

  # get method
  def get_age(self):
    return self._age

  def set_age(self, value):
    self._age = value
```

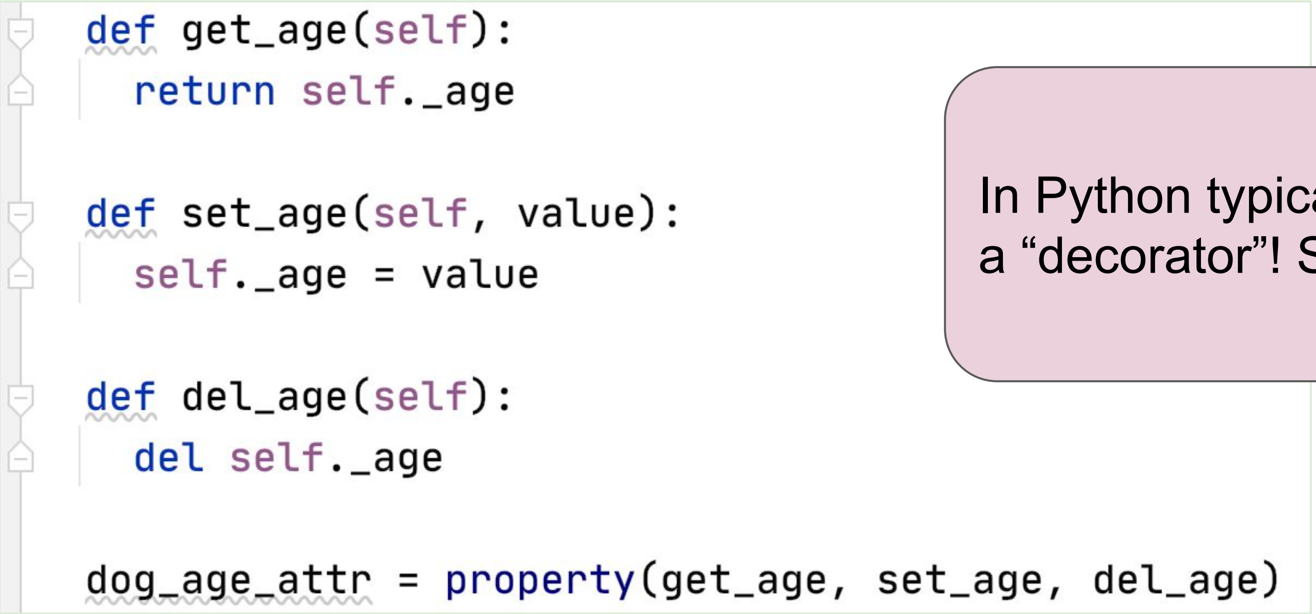


```python
luna = MyDog(0, 0)
print(luna.get_age())
luna.set_age(7)
print(luna.get_age())
```

Notice how we don’t need to pass the age because of the default value.

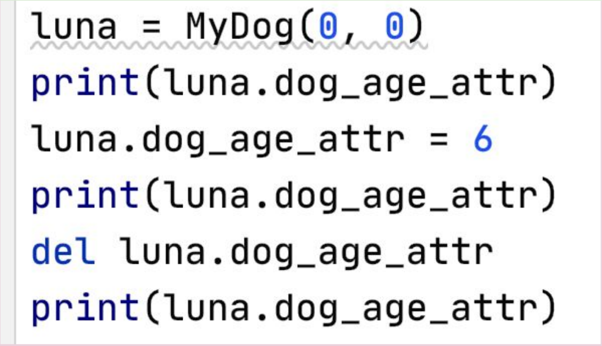
`_age` does not appear as an option when you type `luna.`

Use property() instead of get() and set()

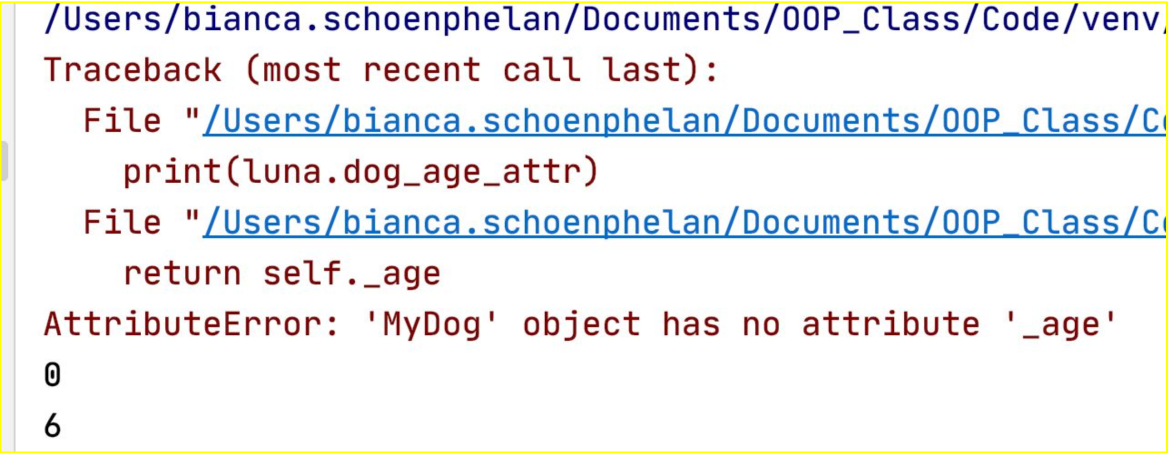


```
def get_age(self):
  return self._age

def set_age(self, value):
  self._age = value

def del_age(self):
  del self._age

dog_age_attr = property(get_age, set_age, del_age)
```

In Python typically handled via a “decorator”! See tutorial.



```
luna = MyDog(0, 0)
print(luna.dog_age_attr)
luna.dog_age_attr = 6
print(luna.dog_age_attr)
del luna.dog_age_attr
print(luna.dog_age_attr)
```



```
/Users/bianca.schoenphelan/Documents/OOP_Class/Code/venv/
Traceback (most recent call last):
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/C
    print(luna.dog_age_attr)
  File "/Users/bianca.schoenphelan/Documents/OOP_Class/C
    return self._age
AttributeError: 'MyDog' object has no attribute '_age'
0
6
```

Advantages of OOP

- Modularity
- Information-hiding
- Code re-use
  - An existing object from one project can easily be integrated into a different project
  - Knowledge of quality of an object, trust to use it in your code
- Plugability
- Ease of debugging

[1]

Advantages of OOP

- Modularity
- Information-hiding
- Code re-use
- Plugability
  - Problematic objects can be removed quite easily
  - Like a bolt that breaks in mechanics
- Ease of debugging
  - Limited scope for error searches
  - Task separation

[1]

Main OOP Principles

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

[1]

OOP Principle: Inheritance

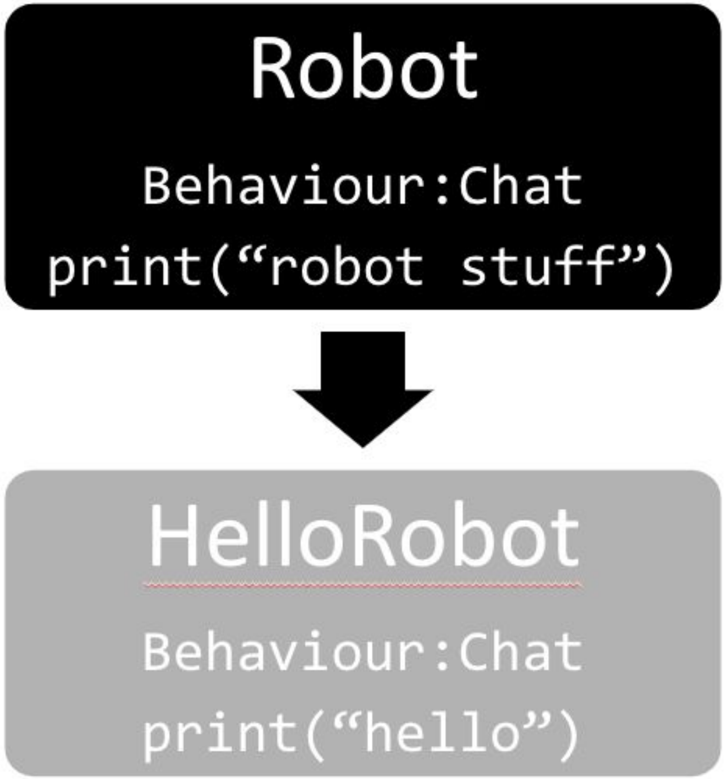
- Different kinds of objects have certain things in common
  - Example: trucks, cars, motorbikes are all motorised vehicles
  - States: current speed, current gear, cc value…
  - Behaviours: drive, stop, change gear
    - Some are executed slightly differently, depending on vehicle → **method overriding**

OOP allows classes to inherit certain traits, aka common states and behaviours, and implement their own versions if required.

[1]

Method Overriding

- Ability of a class to change the implementation of a method that is provided by one of its ancestors
- Very important concept in OOP
- Example on right:
  - Creating an instance of `HelloRobot` and calling the behavior will cause “hello” to be printed on screen
  - Printing is a capability of Robot, so `HelloRobot` can do it too, but has its own version if you want to. Otherwise it will use Robot’s print behaviour.



Robot

Behaviour:Chat

print(“robot stuff”)

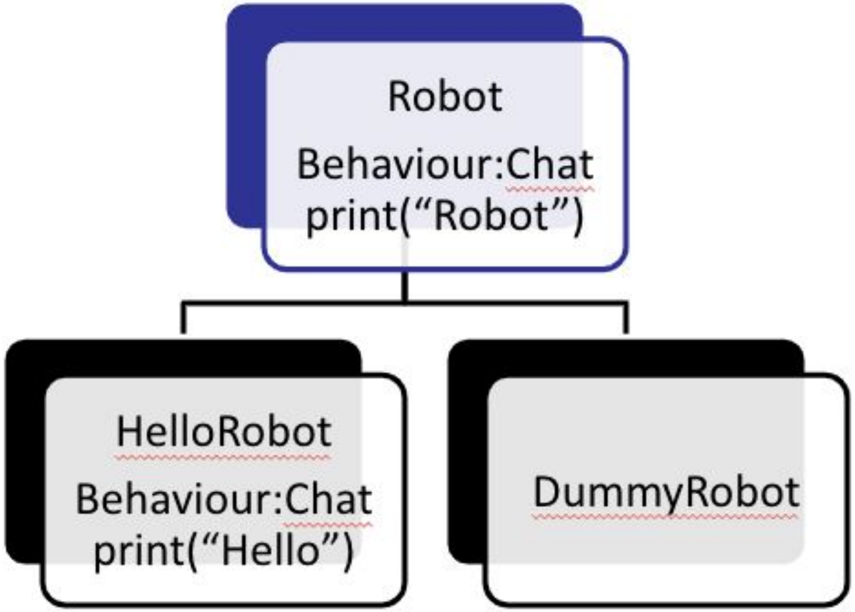
HelloRobot

Behaviour:Chat

print(“hello”)

Method Overriding

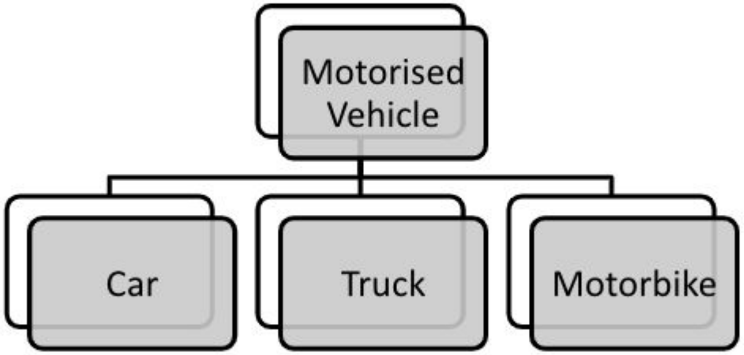
- Happens at class level, the parent level stays in tact
- All methods have the same name
- `DummyRobot` will print "Robot" if asked to Chat



[2]

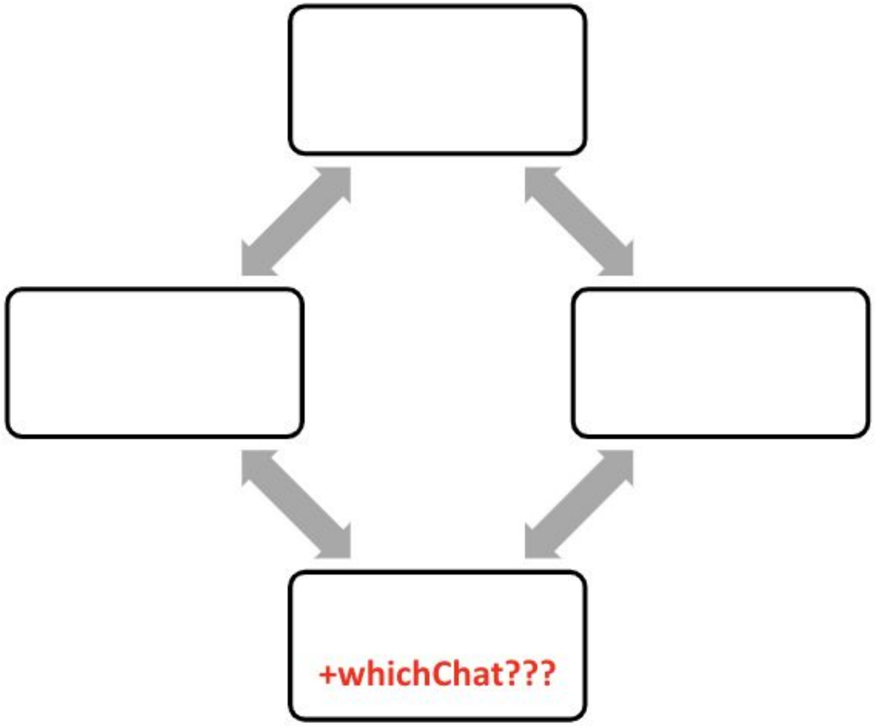
Inheritance Example Vehicles

- Motorised Vehicle: Superclass
- Some OOP allow for only one super class and many child classes
  - Java (not for classes but allows for interfaces)
- Python and c++ allow multiple inheritance
- Diamond Problem



Diamond Problem

- Multiple inheritance allows a child class to inherit from more than one parent class
- Ambiguity issue
- There are some ways around it and we will discuss them soon.



Design

1. Requirements Analysis

- What will the system do?
- Needs are often assessed by interviewing potential users; collect responses, written down
- What other systems need to be interacted with?
- Also legal requirements? Data Protection legislation!
- Requirements specification techniques
  - Object-Oriented analysis (OOA)
  - Data flow diagrams (DFDs)
  - Refinement of application goals
  - Computer-aided

Use of UML Diagrams

- Use Unified Modelling Language (UML) as a design specification standard
  - Combines commonly accepted concepts from many object-oriented (O-O) methods and methodologies
  - Includes **use case diagrams, sequence diagrams, and statechart diagrams**
- Unified modeling language
- 1997 from working group to provide the programming community with the stable and common design language for computer applications

Advantages of UML

- Resulting models can be used to design relational, object-oriented, or object-relational databases
- Brings traditional database modellers, analysts, and designers together with software application developers

Different Types of UML Diagrams

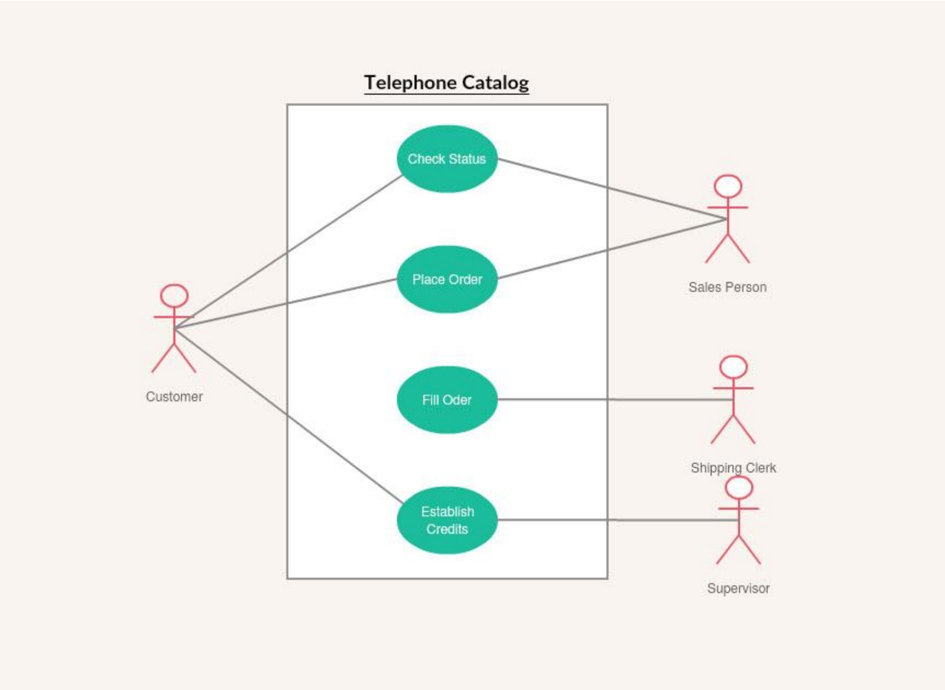
- **Structural** diagrams
  - Class diagrams and package diagrams
  - Object diagrams
  - Component diagrams
  - Deployment diagrams

Different Types of UML Diagrams cont'd

- **Behavioural** diagrams
  - Use case diagrams
  - Sequence diagrams
  - Collaboration diagrams
  - Statechart diagrams
  - Activity diagrams

Use Case Diagram (Behavioural)

- Illustrates a unit of functionality
- Visualise functional requirements
- Relationships of actors with different use cases and processes



Use Case Diagrams Examples

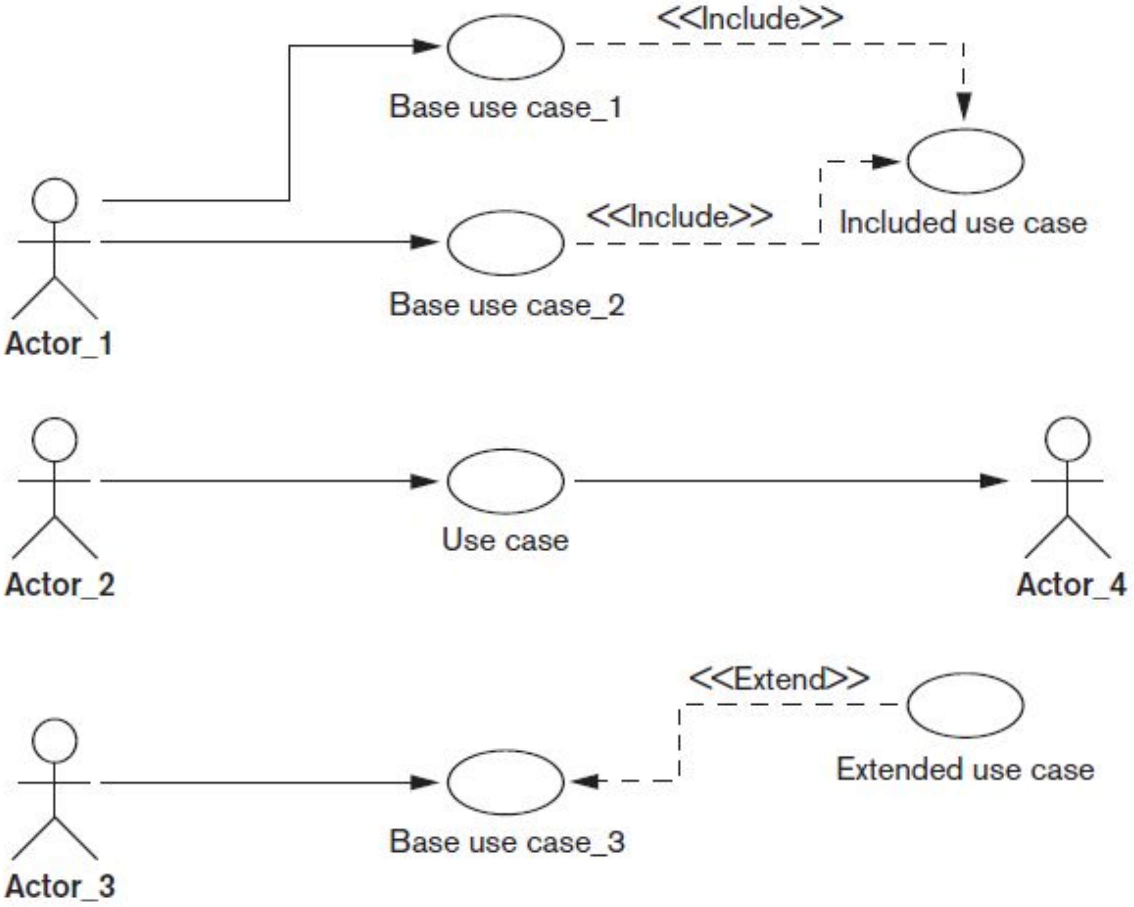
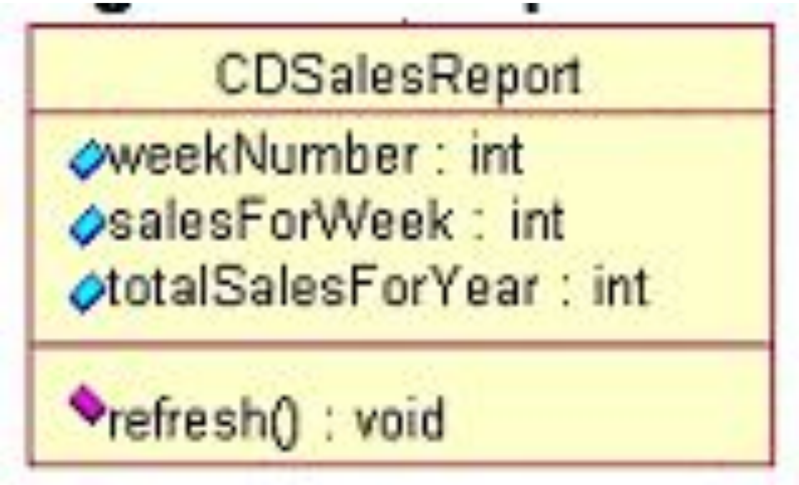


Figure 10.7
The use case diagram notation.

[8]

Class Diagram (Structural)

- Shows how different entities relate to each other; i.e. the static structures of the system
- Rectangle with three sections



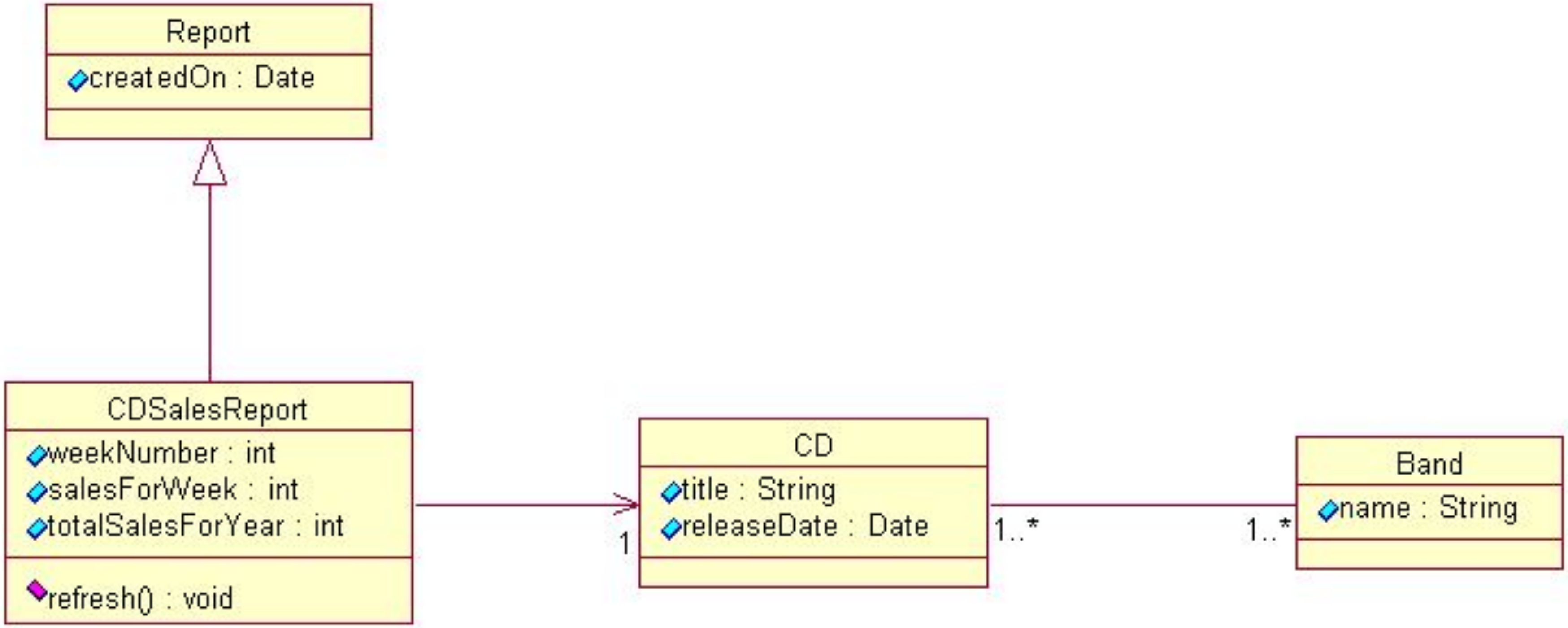
CDSalesReport

weekNumber : int
salesForWeek : int
totalSalesForYear : int

refresh() : void

[7]

Example Class Diagram



[7]

Sequence Diagram (Behavioural)

- Show detailed flow of a specific use case
- Show calls between different objects
- Vertical dimension:
  - Sequence of messages in the time order they occur in
- Horizontal dimension:
  - Object instances to which messages are sent

Sequence Diagram Example

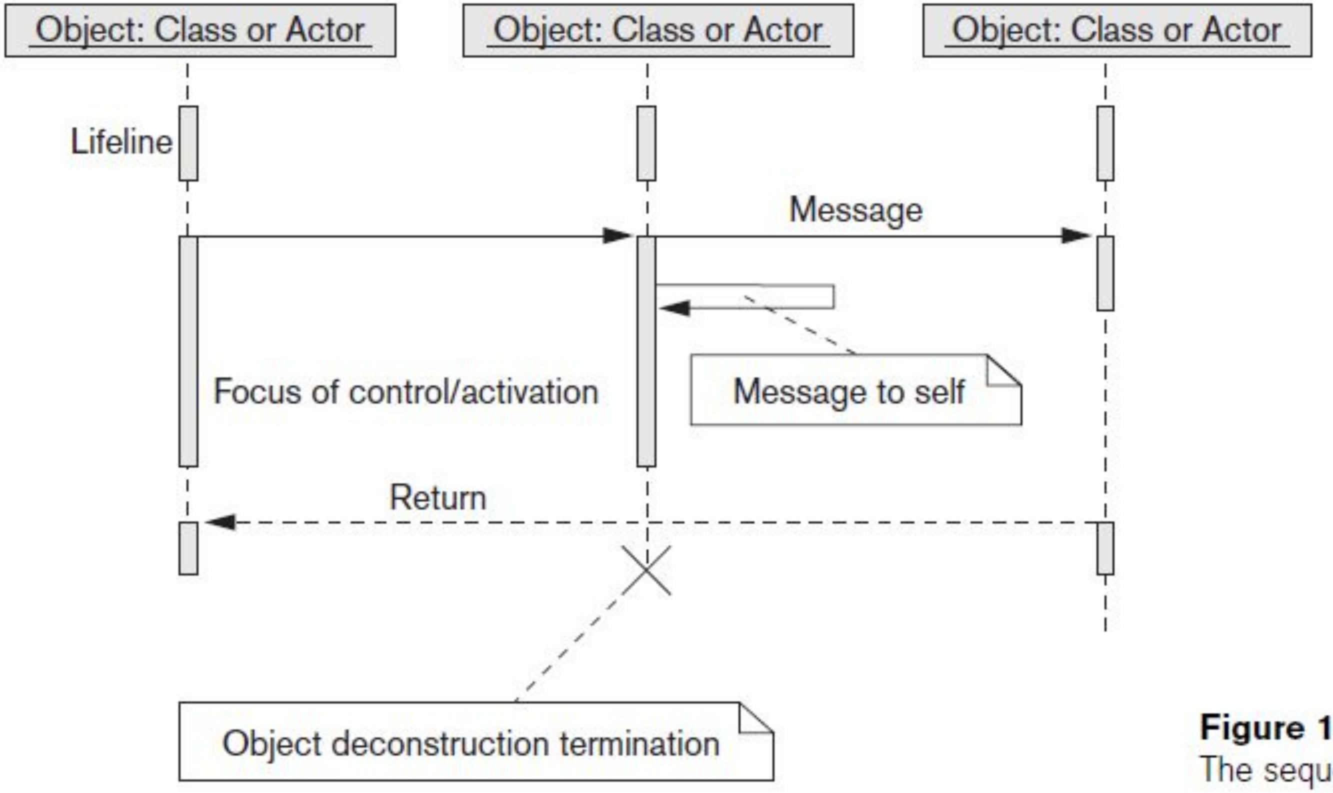


Figure 10.9
The sequence diagram notation.

[8]

Statechart Diagram (Behavioural)

- Different states that a class can be in
- Shows the transition
- Every class has a state, but not every class should have a statechart diagram
- At least 3 interesting states

Statechart Diagram Example

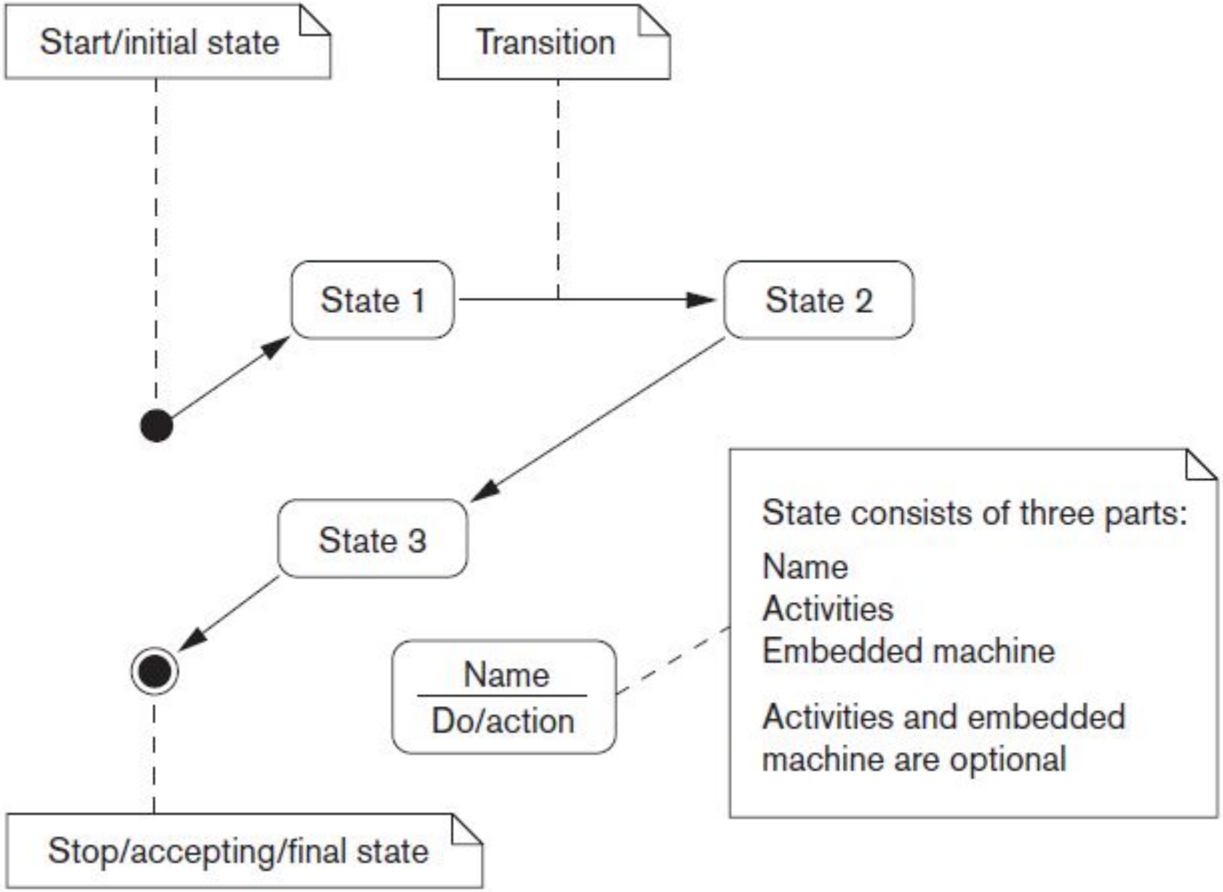

Start/initial state
Transition
State 1
State 2
State 3
Name
Do/action
State consists of three parts:
Name
Activities
Embedded machine
Activities and embedded machine are optional
Stop/accepting/final state

Figure 10.10
The statechart diagram notation.

[8]

Design Example: University

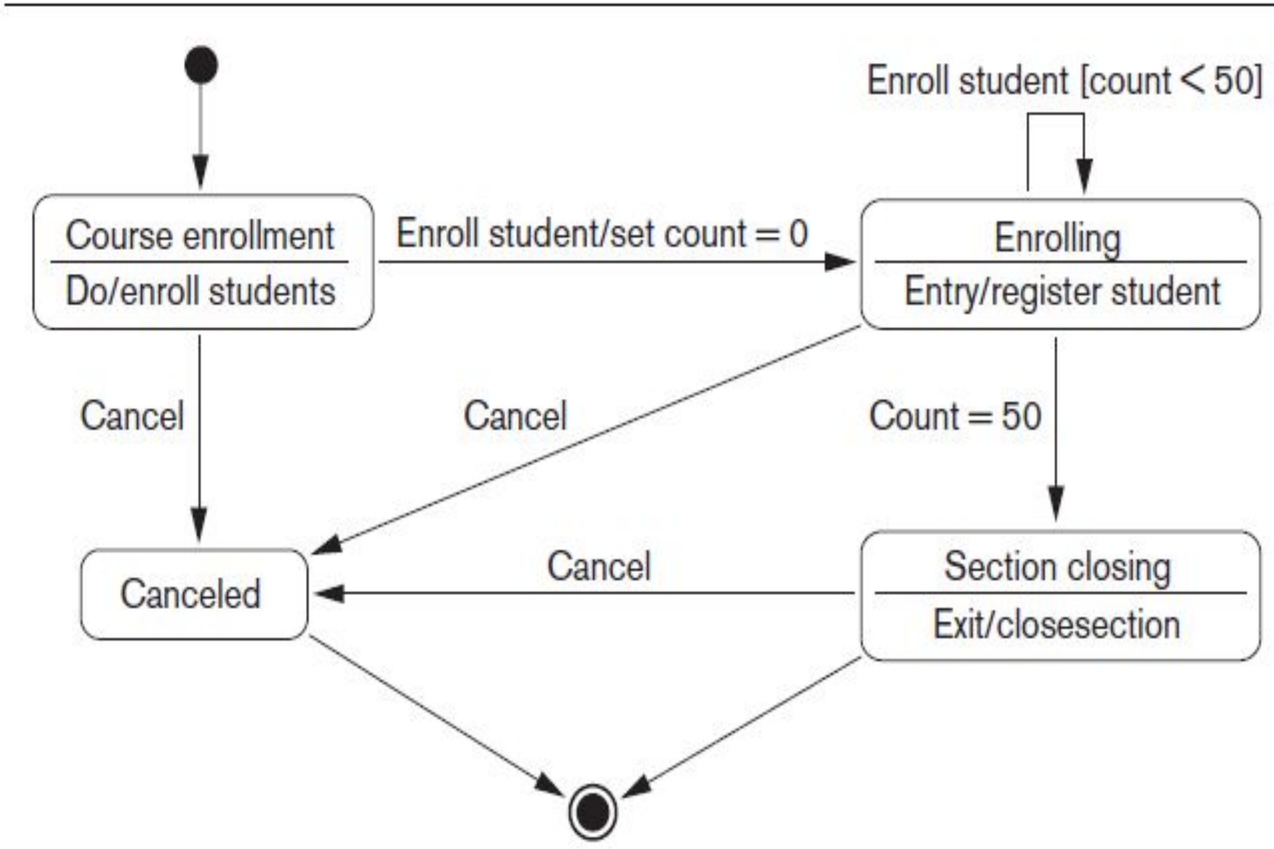


Figure 10.11
A sample statechart diagram for the UNIVERSITY database.

[8]

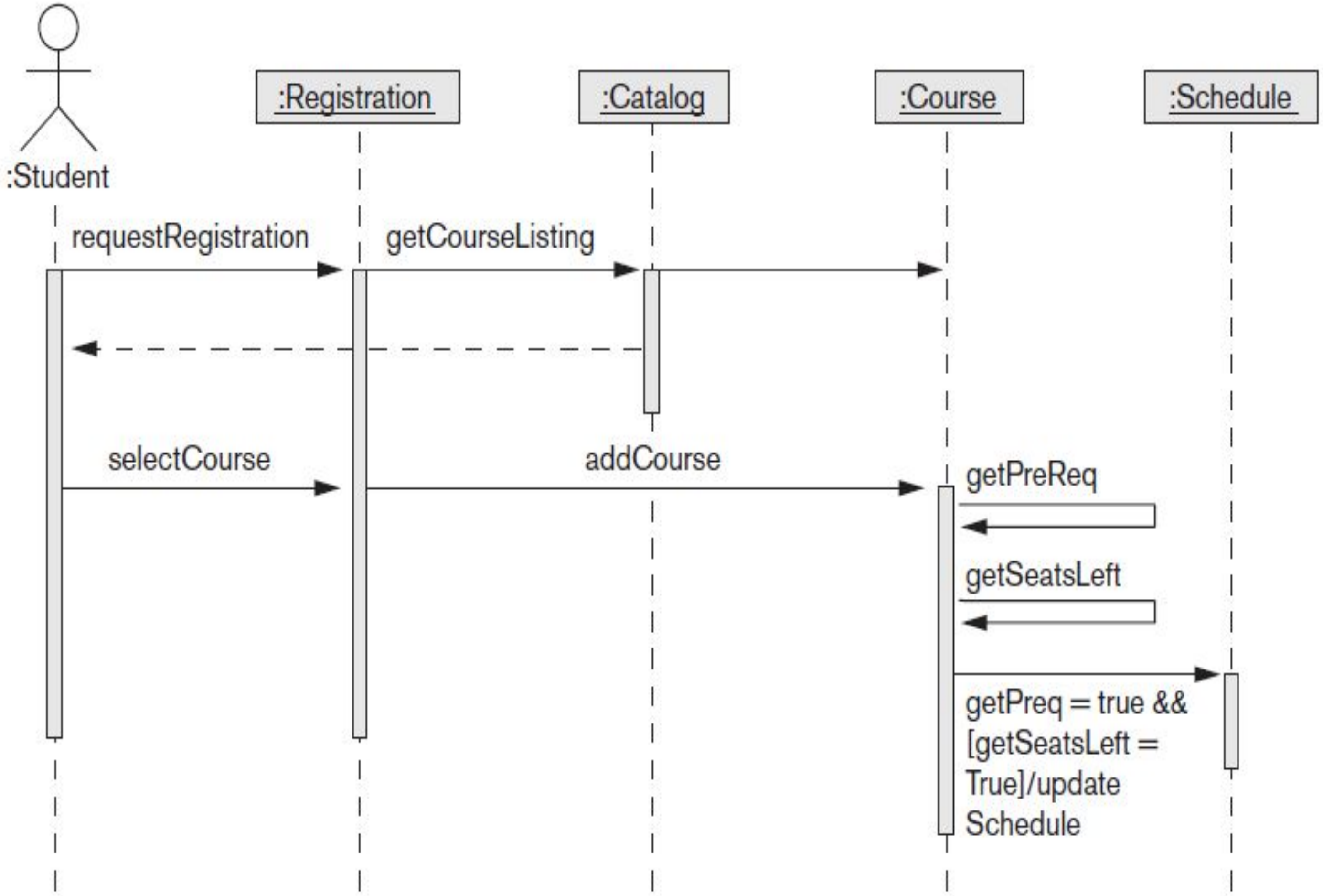


Figure 10.12
A sequence diagram for the UNIVERSITY database.

[8]

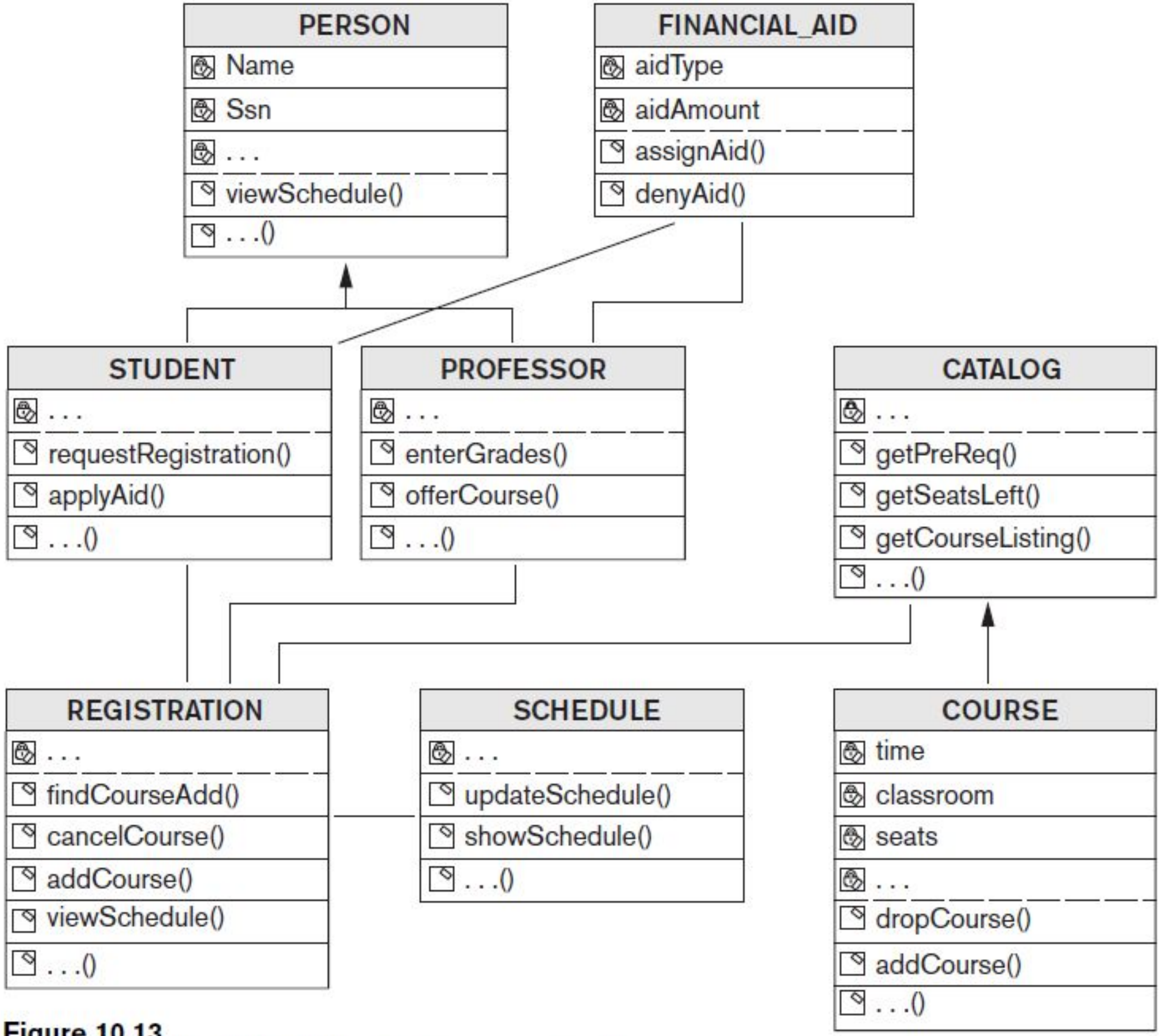


Figure 10.13
The design of the UNIVERSITY database as a class diagram.

[7]

UML Summary

- There are a lot more diagrams
- Many specifications and books written
- Not all relevant for DB and IS design
- Originates from programming perspective
- Several s/w tools support UML

Summary

- ★ **Code Documentation API in Python**
- ★ **Object Oriented Principles**
- ★ **OOP in Practice with Python**
- ★ **OOP Design using UML**

References

1. Concepts of objects, Oracle corp, https://docs.oracle.com/javase/tutorial/java/concepts/object.html, accessed Nov 2020.
2. Python – overwriting methods, https://pythonprogramminglanguage.com/overriding-methods/, accessed Nov 2020.
3. n/a
4. Python Course, Multiple Inheritance, https://www.python-course.eu/python3_multiple_inheritance.php, accessed Nov 2020.
5. Python 3: Object-oriented programming, 2nd edition, Dusty Phillips, 2015, Packt Publishing.
6. IBM UML Introduction; http://www.ibm.com/developerworks/rational/library/769.html, accessed Nov 2020
7. Sparks, G.; Database Modelling in UML; http://www.sparxsystems.com/downloads/whitepapers/Database_Modeling_In_UML.pdf; accessed Nov 2020.
8. Elmasri, R. and Navathe, S.B.; Fundamentals of Database Systems; 4th ed.; Addison-Wesley, 2003; ISBN 0-321-20448-4.
9. https://www.w3schools.com/python/python_classes.asp, accessed Nov 2020
10. n/a
11. Geeks for Geeks, Getter and Setter in Python, https://www.geeksforgeeks.org/getter-and-setter-in-python/, accessed Nov 2020