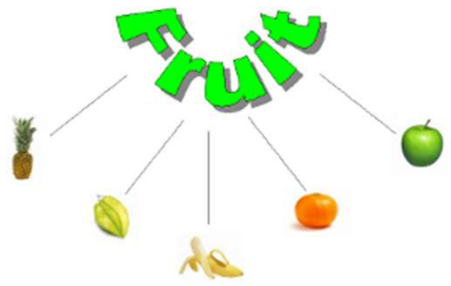
Machine Learning for IoT

Part-I: Object Oriented Programming (Basics)

Object Oriented Programming



- An **object** is a software item that contains **variables** and **methods**
- An object oriented program is based on classes and there exists a **collection of interacting objects**, as opposed to the procedural programming, in which a program consists of functions and routines.
- In Object Oriented Programming (OOP), each object can receive messages, process data and send messages to other objects.

Object Oriented Programming

Object Oriented Programming (OOP) is a new way of organizing a program

Why OOP?

- Programs are getting too large to be fully comprehensible by any person
- There is a need for a way of managing very-large projects
- Key advantages:
  - Allow programmers to (re)use large blocks of code without knowing all the picture
  - Make code reuse a real possibility
  - Simplify maintenance and evolution

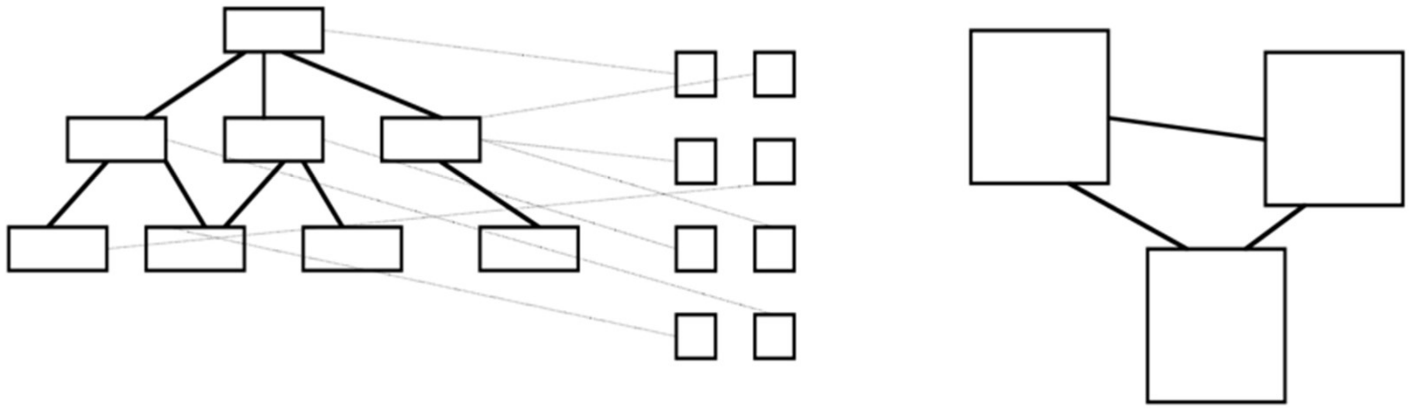
An Engineering Approach

Given a system, with components and relationships among them, we have to:

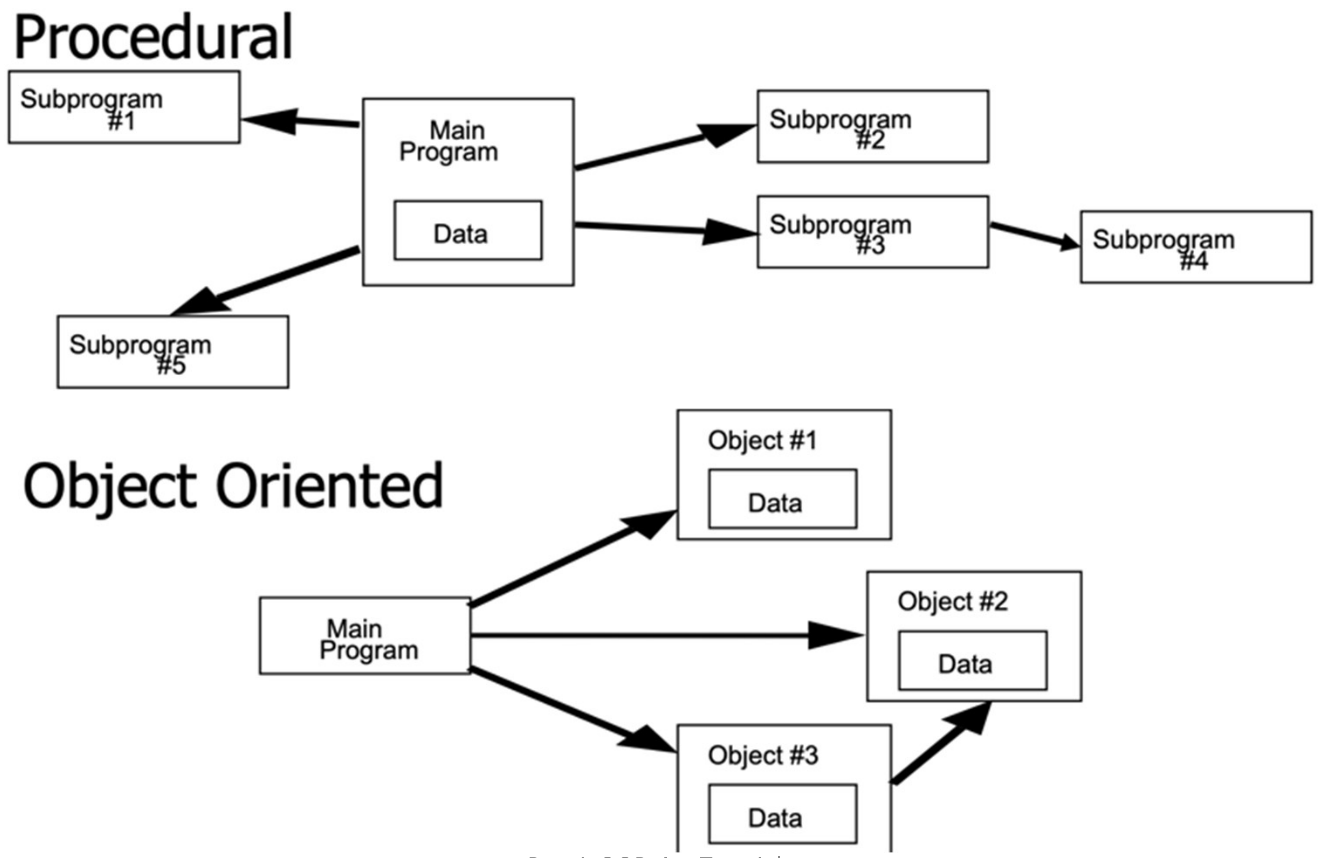
- Identify the components
- Define component interfaces
- Define how components interact with each other through their interfaces
- Minimize relationships among components

Object Oriented Design

- Objects introduce an additional aggregation construct
- More complex system can be built



Procedural vs Object Oriented



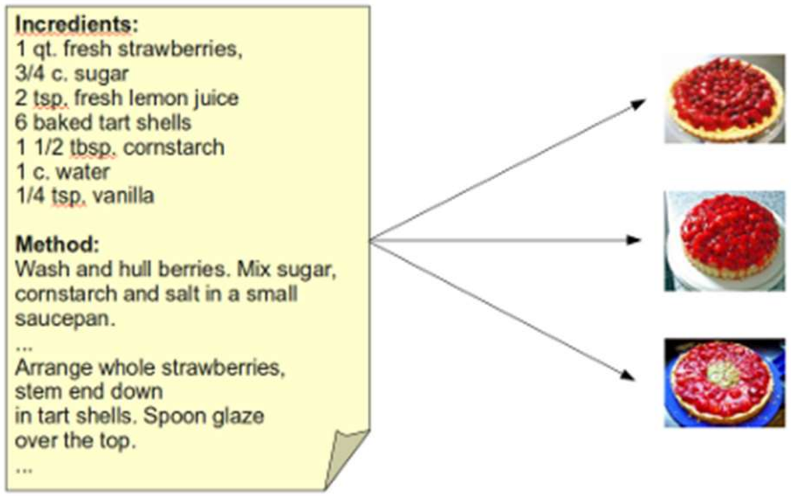
Object Oriented Approach

- Defines a new component type
    - Object (and class)
    - Both data and functions accessing it are within the same module
    - Allows defining a more precise interface

- Defines a new kind of relationship
    - Message passing
    - Read/write operations are limited to the same object scope

Classes

- Like baking a cake, an OOP program constructs objects according to the class definitions
- A class contains variables and methods - if you bake a cake you need ingredients and instructions to bake the cake.



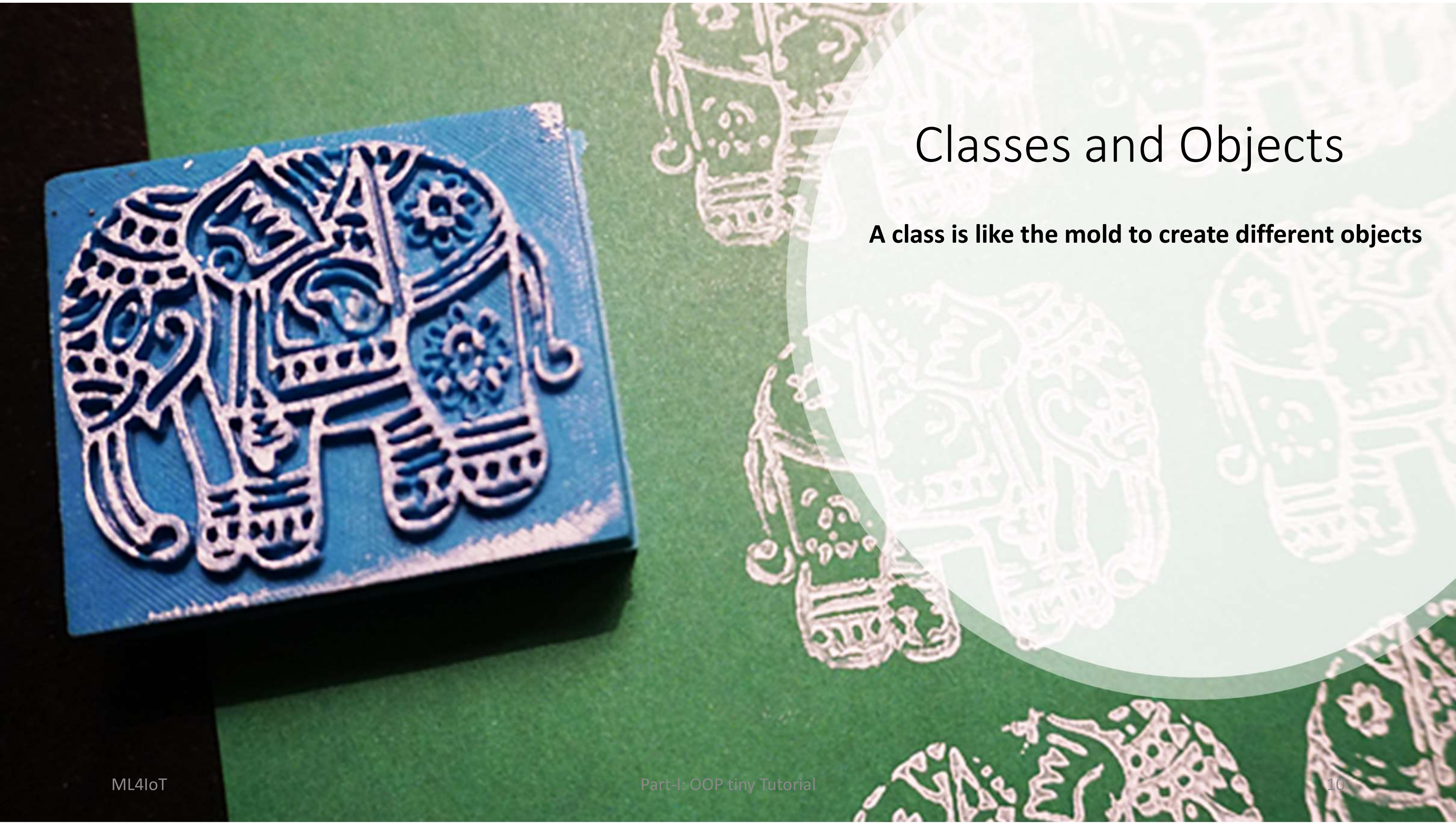
Incredients:
1 qt. fresh strawberries,
3/4 c. sugar
2 tsp. fresh lemon juice
6 baked tart shells
1 1/2 tbsp. cornstarch
1 c. water
1/4 tsp. vanilla

Method:
Wash and hull berries. Mix sugar, cornstarch and salt in a small saucepan.
...
Arrange whole strawberries, stem end down
in tart shells. Spoon glaze over the top.
...

- A class needs variables and methods - there are <u>class variables</u>, which have the same value in all objects and there are <u>instance variables</u>, which have normally different values for different objects
- A class also defines all the necessary methods needed to access the data.

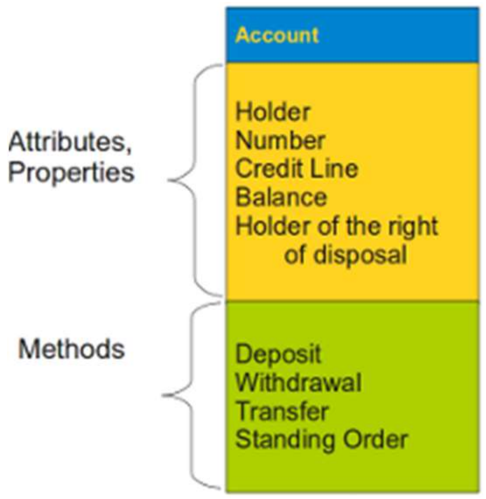
Classes and Objects

A class is like the mold to create different objects



Classes

- A class defines a data type, which contains variables, properties and methods - a class describes the abstract characteristics of a real-life thing
- An instance is an object of a class created at run-time
- The set of values of the attributes of a particular object is called its state.
- The object consists of state and the behavior that is defined in the object's classes
- The terms object and instance are normally used synonymously



Objects

- Model of a physical or logical item
  - e.g. a student, an exam, a window
- Characterized by
  - identity
  - attributes (or state)
  - operations it can perform (behavior)
  - messages it can receive

Class and Object

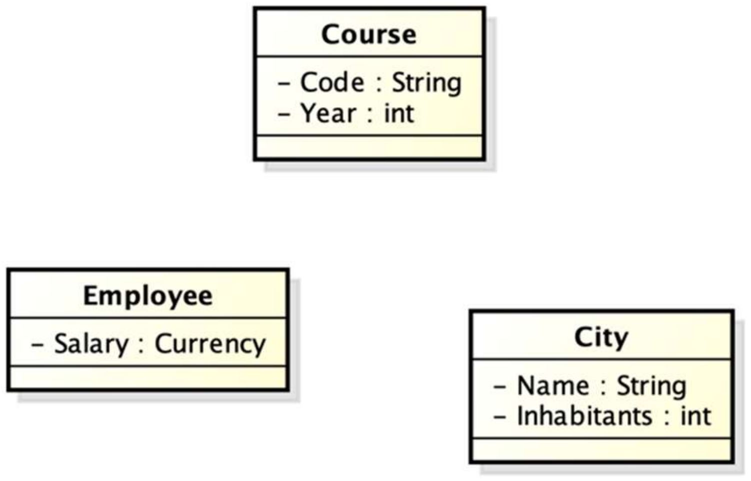
- Class (the description of object structure, i.e. *type*):
  - Data (ATTRIBUTES or FIELDS)
  - Functions (METHODS or OPERATIONS)
  - Creation methods (CONSTRUCTORS)
- Object (class instance)
  - State and identity

Class and Object

- A class is a type definition
  - Typically no memory is allocated until an object is created from the class
- The creation of an object is called instantiation. The created object is often called an instance
- There is no limit to the number of objects that can be created from a class
- Each object is independent. Interacting with one object does not affect the others

Attribute

- Elementary property of classes
  - Name
  - Type
- An attribute associates to each object (instance of a class) a value of the corresponding type
  - Name: String
  - ID: Numeric
  - Salary: Currency

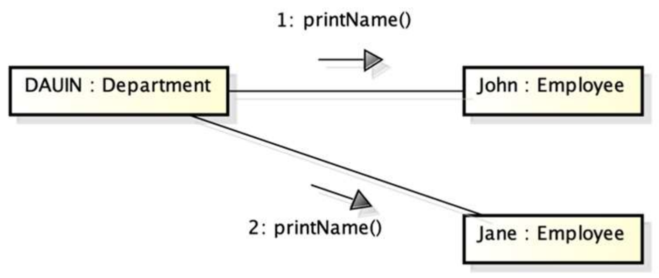


Methods

- Classes usually contain attributes and properties and methods for these attributes and properties
- Essentially, a method is a special function belonging to a class, i.e. it is defined within a class and works on the instance and class data of this class.
- A method describes an operation that can be performed on an object
- Similar to functions in procedural languages
- A method represents the means to operate on or access to the attributes
- Methods can only be called through instances of a class or a subclass, i.e. the instance name followed by a dot and the method name

Message passing

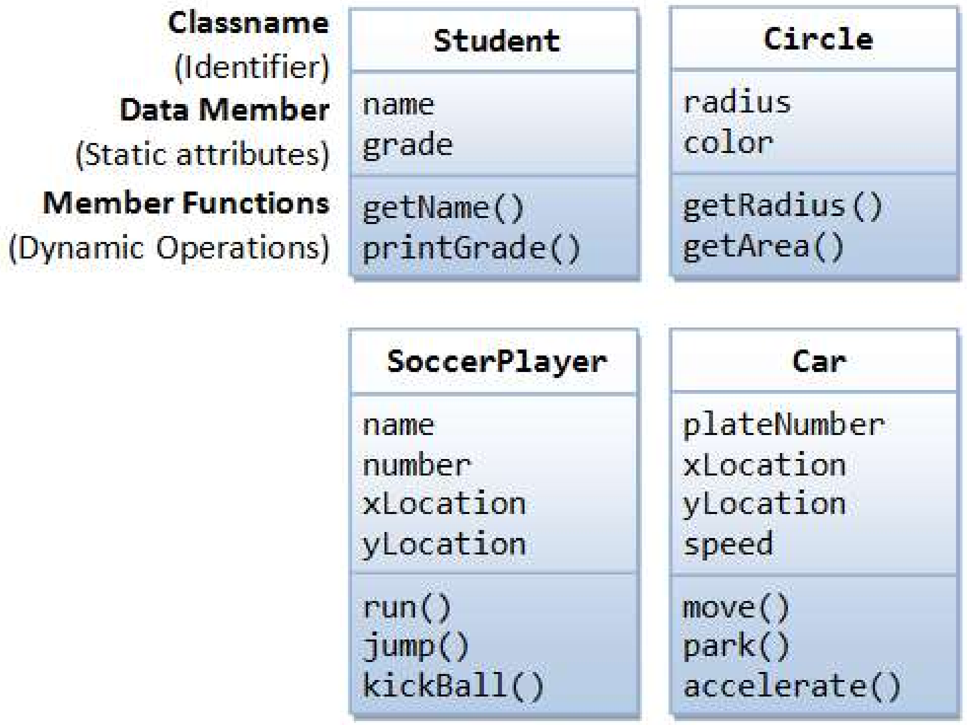
- Objects communicate by message passing
  - Not by direct access to object's local data
- A message is a service request



<u>Note:</u>

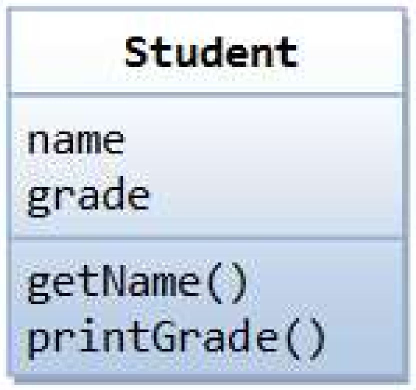
this is an abstract view that is independent from specific programming languages.

Some Examples

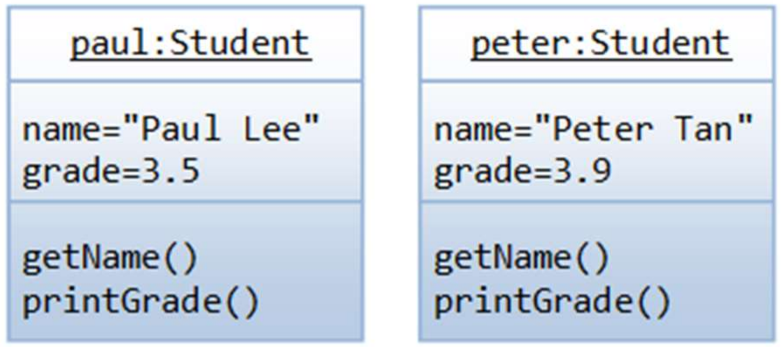


Some Examples

Class Definition

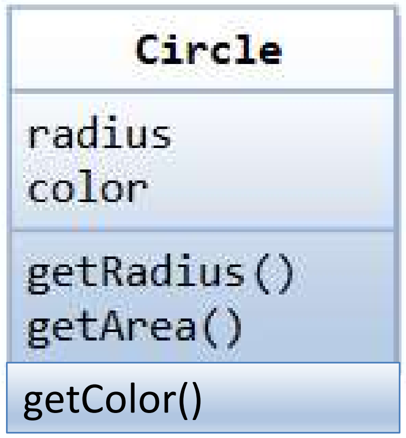


Instances

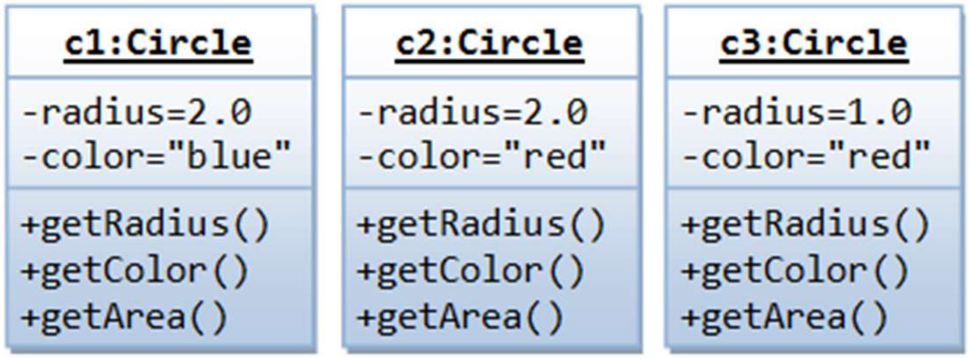


Some Examples

Class Definition



Instances



Definition of Classes

```
class name:
    statements
    ...
-or-
class name(base1, base2, ...):
  def __init__(self, arg1, arg2, ...):
                self.x = arg1
                self.y = arg2
      ...
  def name(self, arg1, arg2, ...):
                ...
```

Class definition

statements can be **method definitions**

__init__ is the **constructor** method called to create an object. It prepares the new object for use, often accepting arguments that the constructor uses to set required member variables.

statements can be **class variable assignments**

Object Methods

`def` **name**`(self,` **parameter**`,` **...**`,` **parameter**`):`

  statements

- `self` *must* be the first parameter to any object method
- *must* access the object's fields through the `self` reference

```
class Point():
    def __init__(self):
        self.x = 0
        self.y = 0
    def translate(self, dx, dy):
        self.x += dx
        self.y += dy
    ...
```

Calling Methods

- A client can call the methods of an object in two ways:
  - the value of `self` can be an implicit or explicit parameter

1) **object.method(parameters)**

or

2) **Class.method(object, parameters)**

- Example:

```
p = Point()
p.translate(1, 5)
or
Point.translate(p, 1, 5)
```

Example

```
class Stack():

    def __init__(self):         # constructor
        self.items = []
    def push(self, x):
        self.items.append(x)   # the sky is the limit
    def pop(self):
        x = self.items[-1]           # what happens if it is empty?
        del self.items[-1]
        return x
    def isEmpty(self):
        return len(self.items) == 0 # Boolean result
```

Example

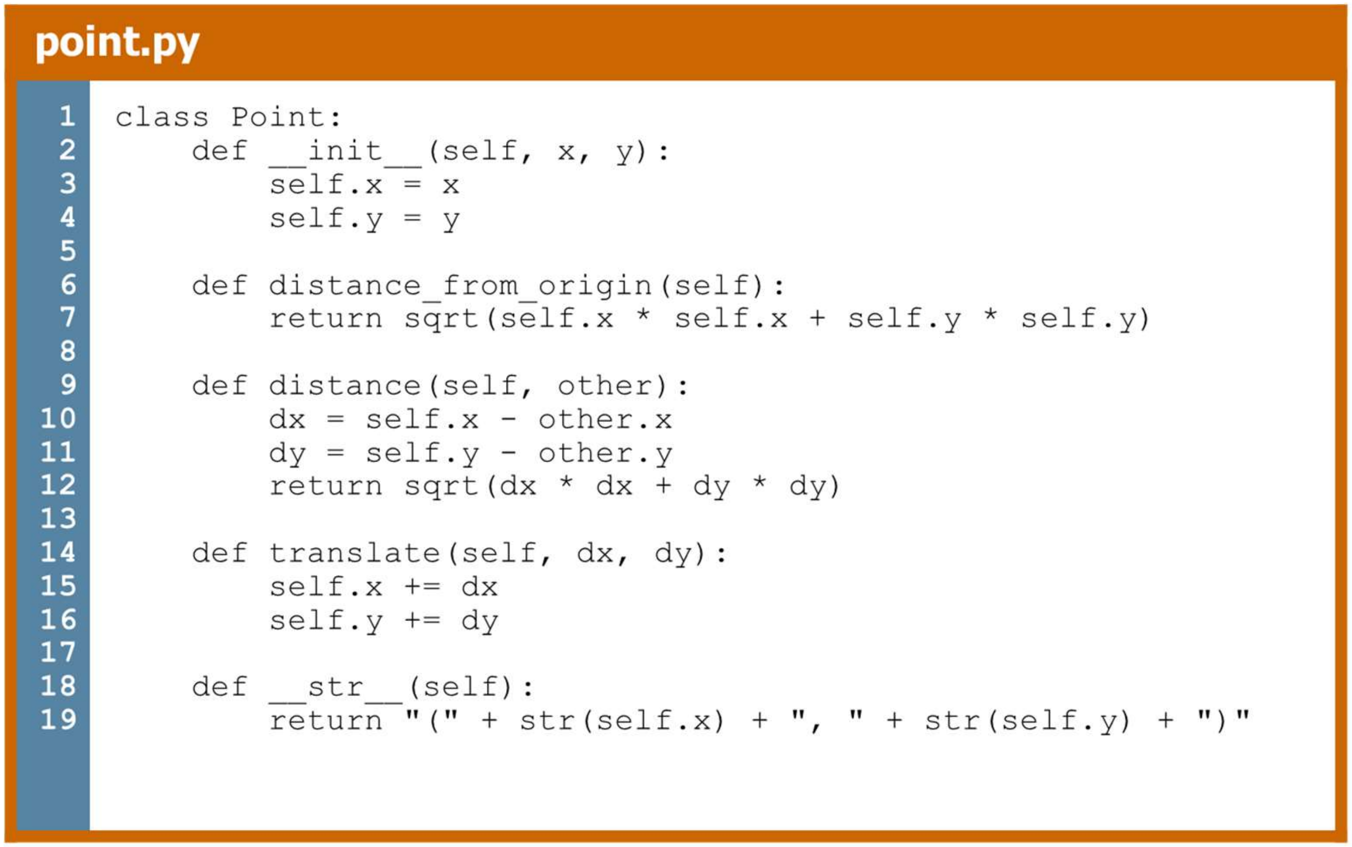
```
x = Stack()                # x is a new object instance
                           # of the class Stack()

x.isEmpty()                # -> 1
x.push(1)                  # [1]
x.isEmpty()                # -> 0
x.push("hello")            # [1, "hello"]
x.pop()                    # ["hello"]

x.items                    # -> [1]
```

To create an instance, simply call the class object

To use methods of the instance, call using dot notation

To inspect instance variables, use dot notation

Example Class


point.py

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)

    def translate(self, dx, dy):
        self.x += dx
        self.y += dy

    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Main Principles of OOP

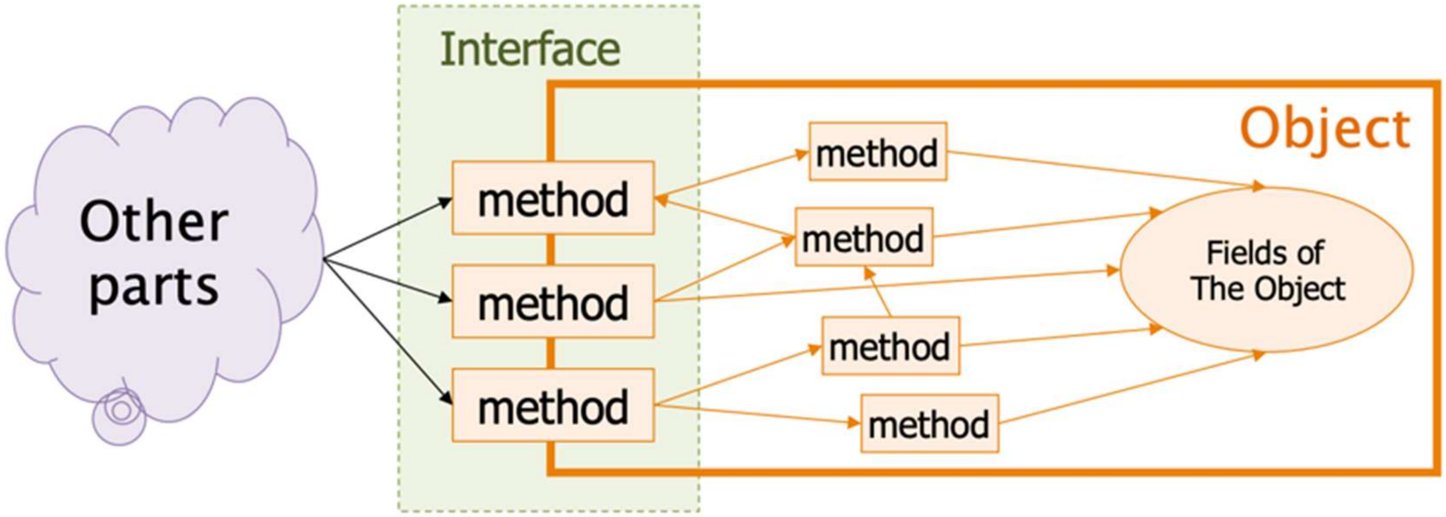
- Interface
- Encapsulation
- Inheritance
- Polymorphism

Interface

- An interface is a set of messages an object can receive
  - Each message is mapped to an internal “function” within the object
  - The object is responsible for the association (message -> function)
  - Any other message is illegal
- The interface
  - Encapsulates the internals
  - Exposes a standard boundary

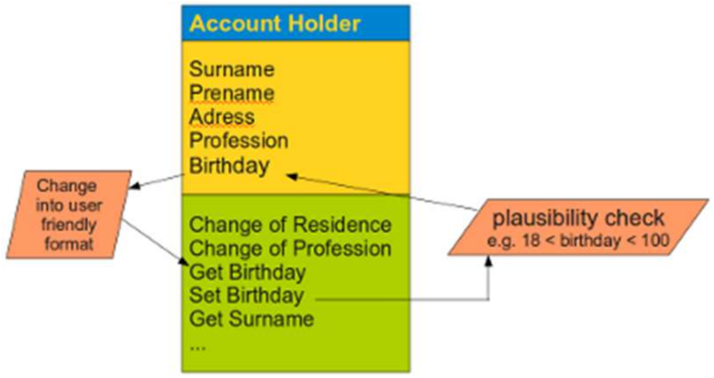
Interface

- The interface of an object is simply the subset of methods that other "program parts" are allowed to call



Encapsulation

- Dividing the code into a public interface, and a private implementation of that interface
- Encapsulation is the mechanism for restricting the access to some of object's components, this means that the internal representation of an object cannot be seen from outside of the object's definition
- Access to this data is typically only achieved through special methods: Getters and Setters - by using solely get() and set() methods, we can make sure that the internal data cannot be accidentally set into an inconsistent or invalid state

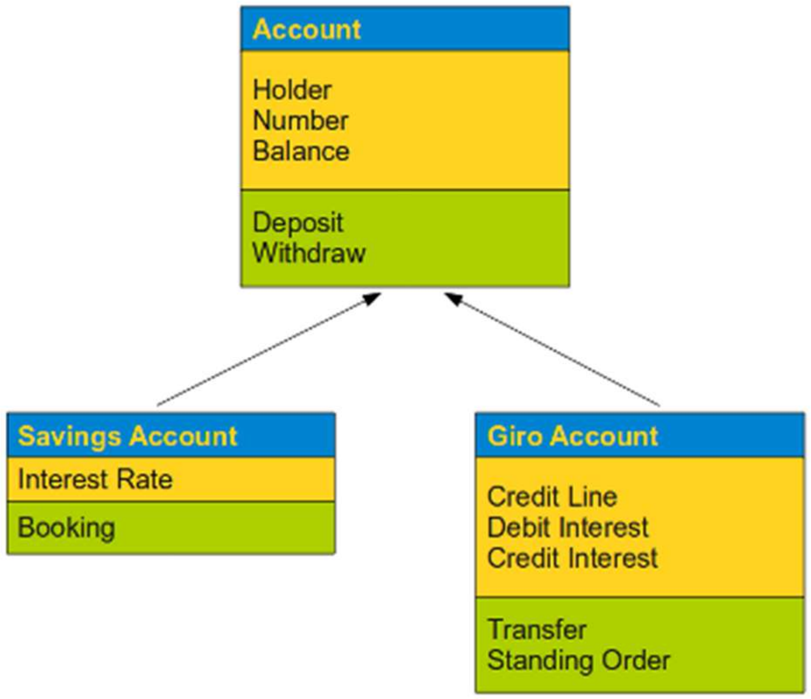


Encapsulation

- Simplified access
  - To use an object, the user need only comprehend the interface. No knowledge of the internals are necessary
- Self-contained
  - Once the interface is defined, the programmer can implement the interface (write the object) without interference of others
- Ease of evolution
  - Implementation can change later without rewriting any other part of the program (as long as the interface does not change)
- Single point of change
  - Any change in the data structure means modifying the code in one location, rather than code scattered around the program (error prone)

Inheritance

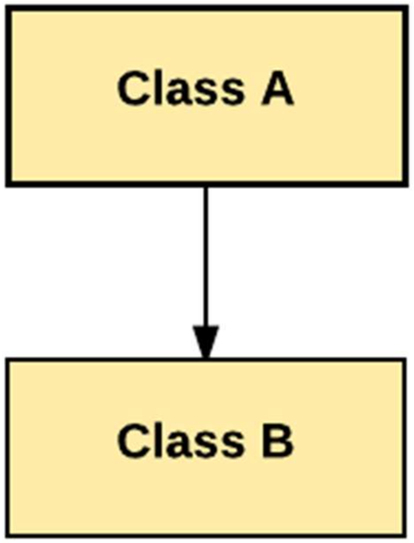
- Classes can inherit other classes - a class can inherit attributes and behaviour (methods) from other classes, called super-classes
- A class which inherits from super-classes is called a Sub-class
- There exists a hierarchy relationship between classes
- Inheritance is used to create new classes by using existing classes - new ones can both be created by extending and by restricting the existing classes



Type of Inheritance

In **Single Inheritance** one class extends another class (one class only).

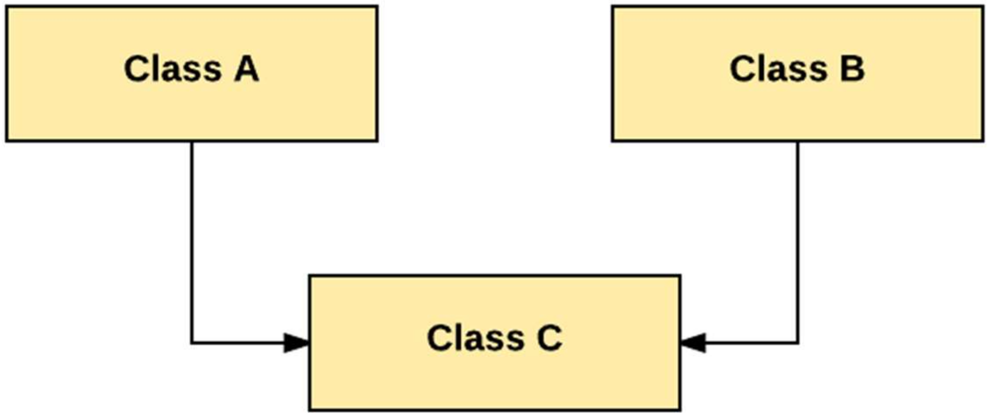
- Class B extends only Class A
- Class A is a super class and Class B is a Sub-class



Type of Inheritance

In **Multiple Inheritance**, one class extending more than one class.

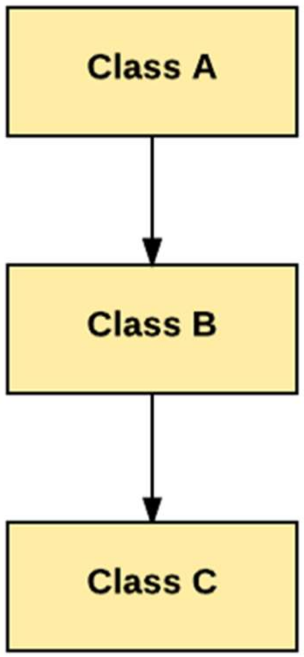
- Class C extends Class A and Class B both



Type of Inheritance

In **Multilevel Inheritance**, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

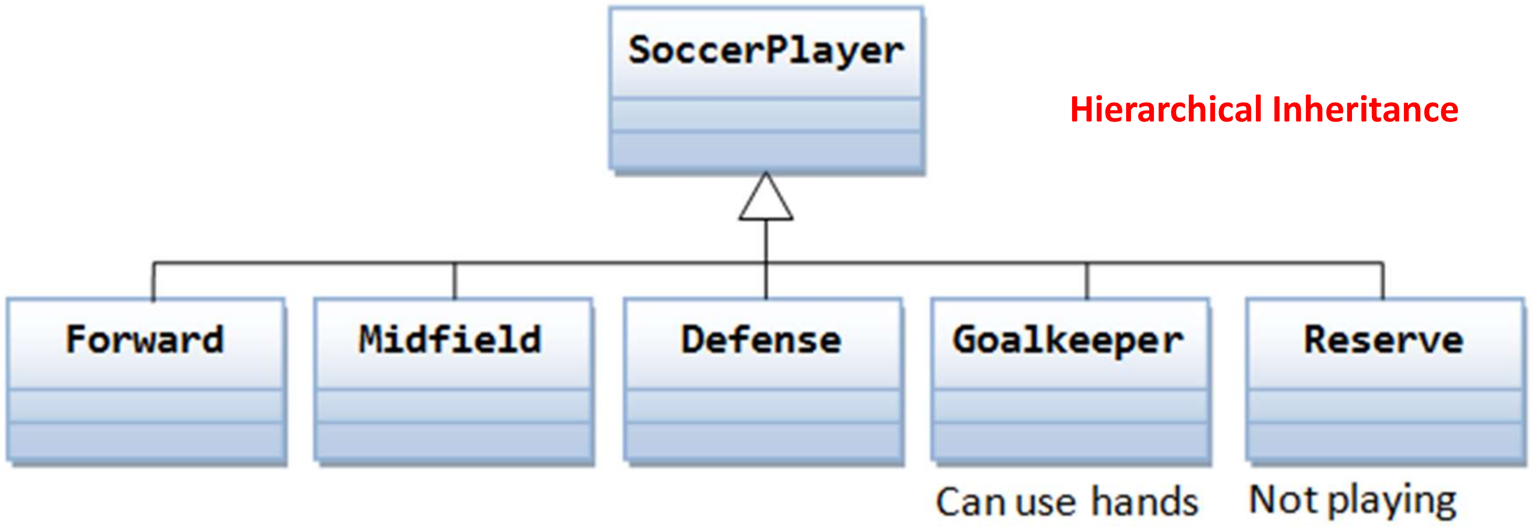
- Class C is subclass of B and B is a of subclass Class A



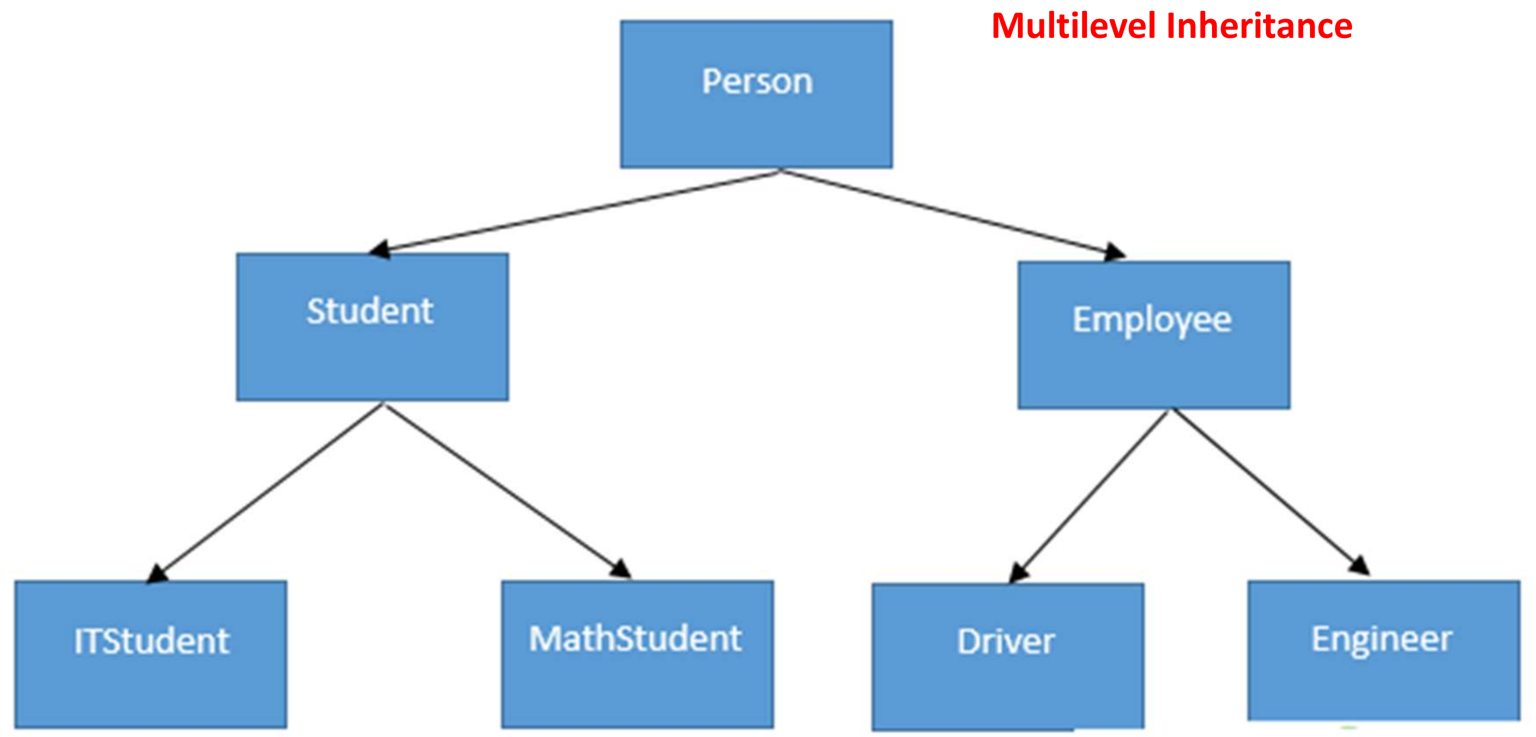
Why Inheritance?

- Frequently, a class is merely a modification of another class. In this way, there is minimal repetition of the same code
- Localization of code
  - Fixing a bug in the base class automatically fixes it in the subclasses
  - Adding functionality in the base class automatically adds it in the subclasses
  - Less chances of different (and inconsistent) implementations of the same operation

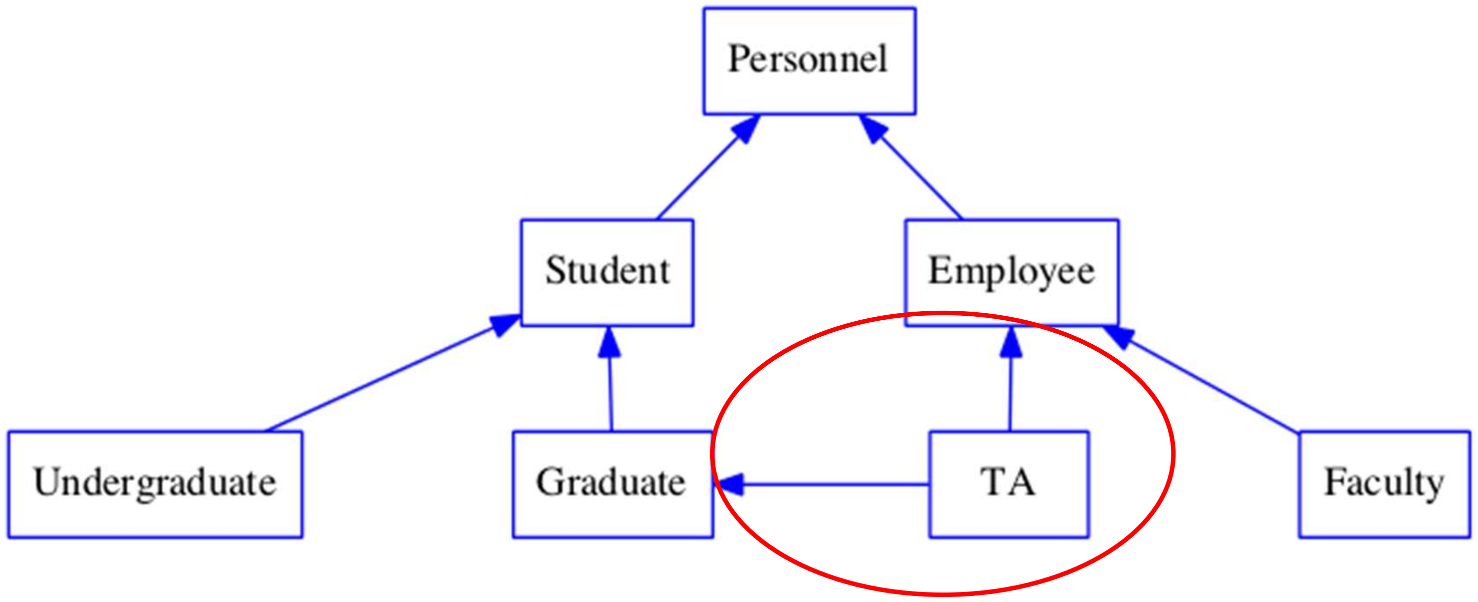
Some Examples



Some Examples



Some Examples



Multiple Inheritance

Inheritance in Python

```
class name(superclass):
    statements
```

- Example:

```
class Point3D(Point):    # Point3D extends Point
    z = 0
    ...
```

- Python supports both *multiple* and *multilevel inheritance*

```
class name(superclass, ..., superclass):
    statements
```

(if > 1 superclass has the same field/method, conflicts are resolved in left-to-right order)

Calling Superclass Methods

- methods: **super() . method** (**parameters**)
- constructors: **super() .** `__init__` (**parameters**)

```
class Point3D(Point):
    def __init__(self, x, y, z):
        super().__init__(x, y)
        self.z = z

    def translate(self, dx, dy, dz):
        super().translate(dx, dy)
        self.z += dz
```

Super Class:

```
class Point():
    def __init__(self):
        self.x = 0
        self.y = 0
    def translate(self, dx, dy):
        self.x += dx
        self.y += dy
```

Polymorphism

- The ability to **overload** standard operators so that they have appropriate behavior based on their context

- Ex. The “+” operator has a different behavior when applied to numbers or strings

```
Num1 = 12
Num2 = 34
Num3 = Num1 + Num2      # The result is 46

Str1 = “Hello ”
Str2 = “World!”
Str3 = Str1 + Str2      # The result is “Hello World!”
```

Operator Overloading

- **operator overloading**: You can define functions so that Python's built-in operators can be used with your class.
  - See also: http://docs.python.org/ref/customization.html

Operator	Class Method
-	`__neg__(self, other)`
+	`__pos__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`

Unary Operators

-	`__neg__(self)`
+	`__pos__(self)`

Operator	Class Method
==	`__eq__(self, other)`
!=	`__ne__(self, other)`
<	`__lt__(self, other)`
>	`__gt__(self, other)`
<=	`__le__(self, other)`
>=	`__ge__(self, other)`

References

- https://docs.python.org/3/tutorial/classes.html
- Fowler, M. “UML Distilled: A Brief Guide to the Standard Object Modeling Language - 3rded.”, Addison-Wesley Professional (2003)

Examples: OOP in Python

- **Notebook #1:** Encapsulation
- **Notebook #2:** Inheritance
- **Notebook #3:** Polymorphism
- **Notebook #4:** The `__call__` method
- **Notebook #5:** A DIY Training Framework for ML

A DIY Training Framework for ML

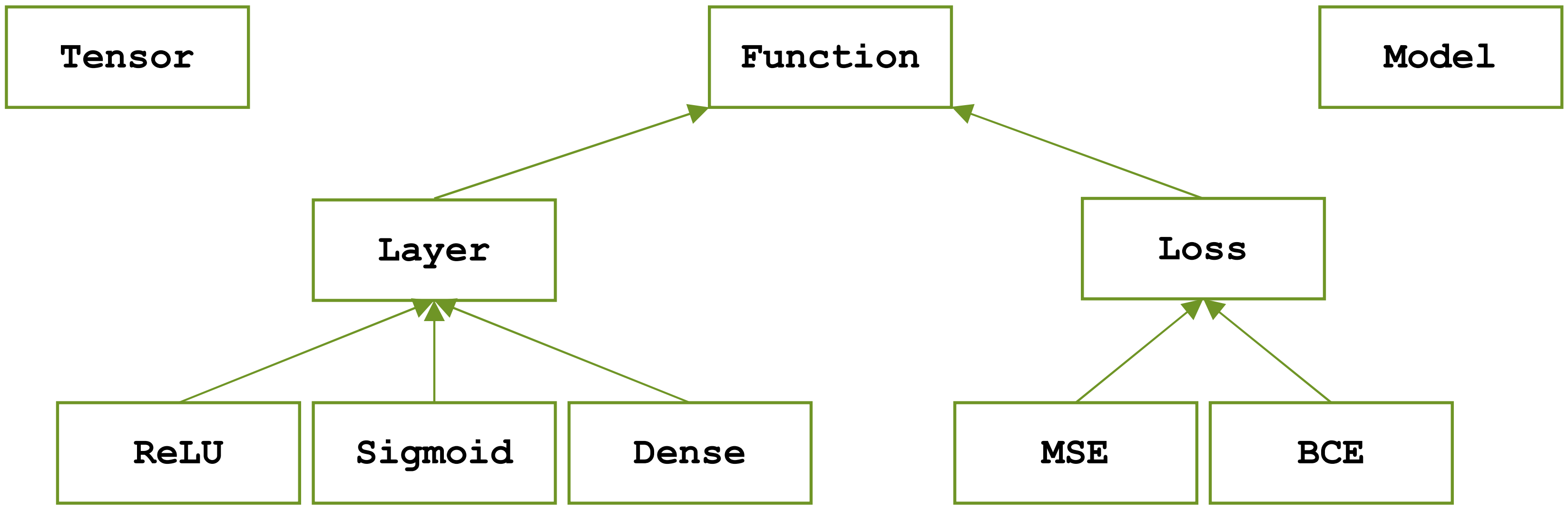
- The most common ML/DL frameworks are built using OOP (e.g., PyTorch, TensorFlow)

- Goal: Create a toy training framework for ML from scratch using `numpy` functions

- Disclaimer: this is just an example to understand OOP
  - do NOT use it for real ML applications but…
  - *…you can really understand something once you are able to build it on your own*

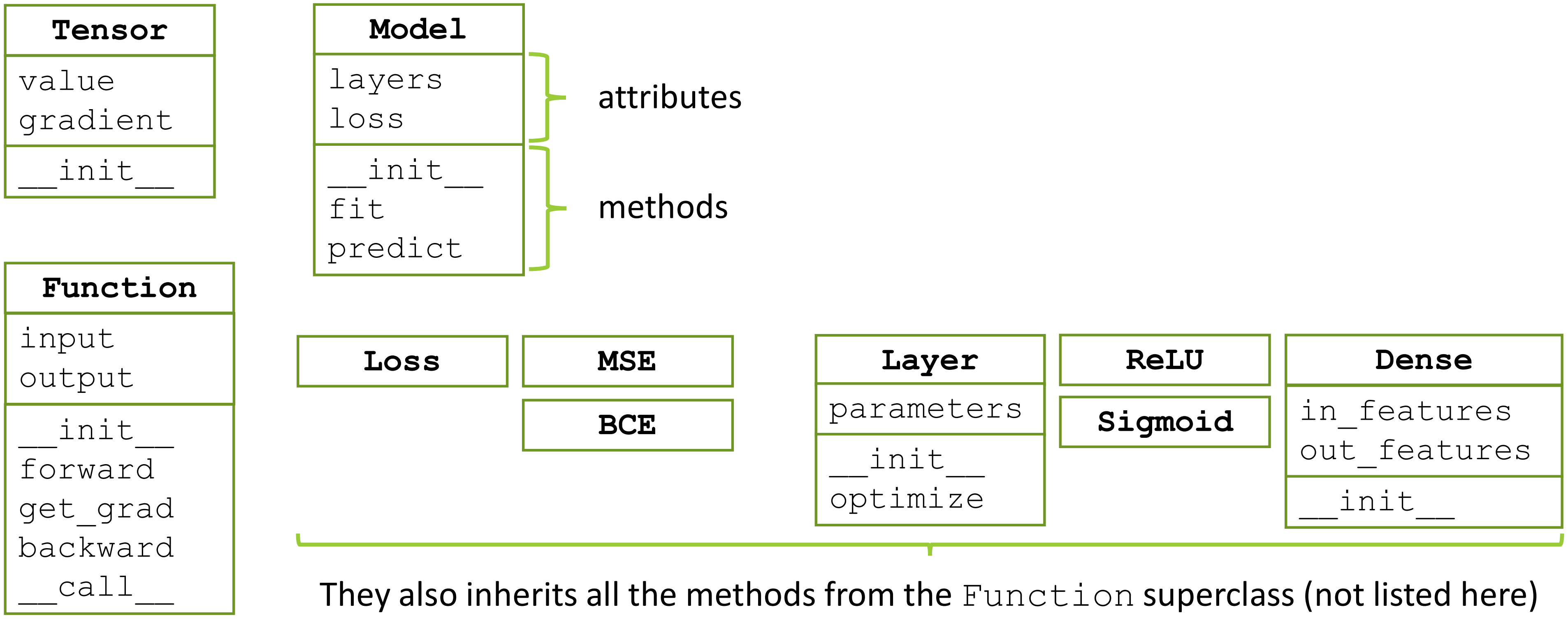
A DIY Training Framework for ML (cont’d)

- Step 1: Identify all the components (the classes)



A DIY Training Framework for ML (cont'd)

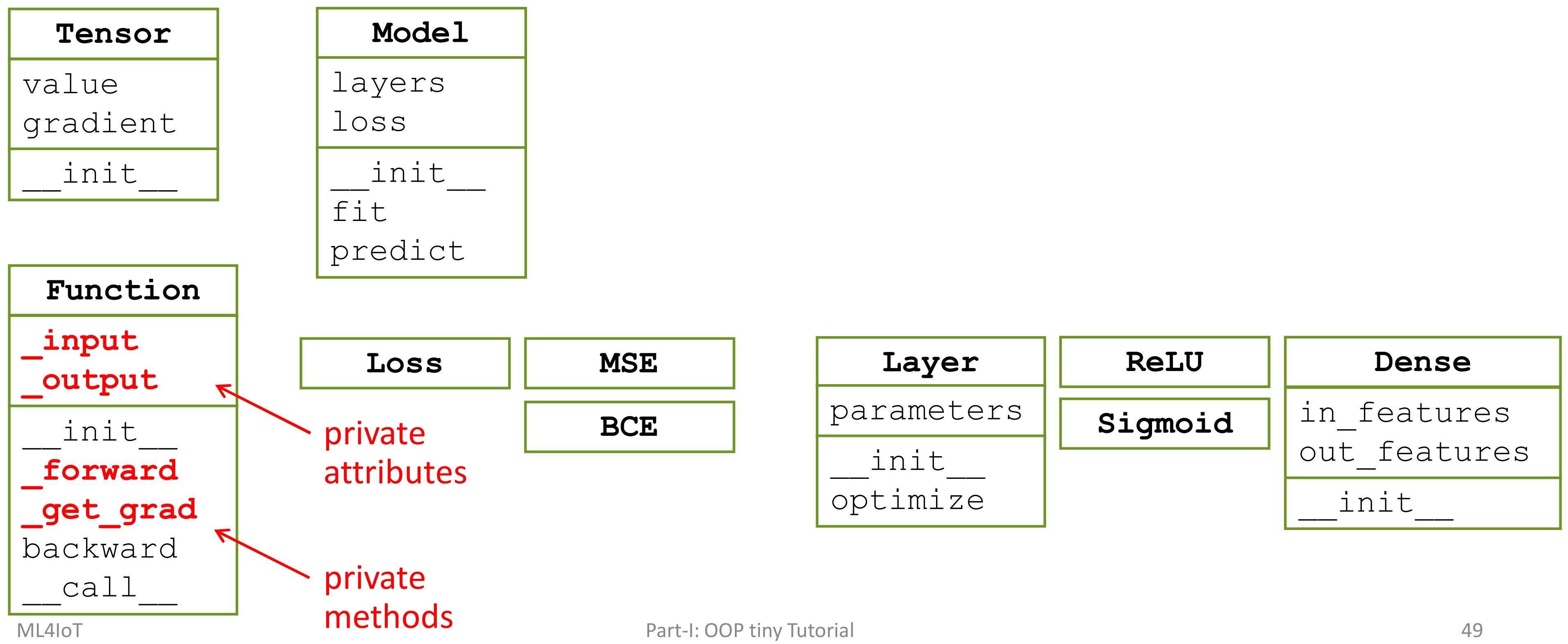
- Step 2: Identify attributes and methods for each class



They also inherits all the methods from the `Function` superclass (not listed here)

A DIY Training Framework for ML (cont'd)

- Step 3: Define components interfaces



A DIY Training Framework for ML (cont’d)

- Step 4: Implement all the methods
  - *…see the Notebook…*