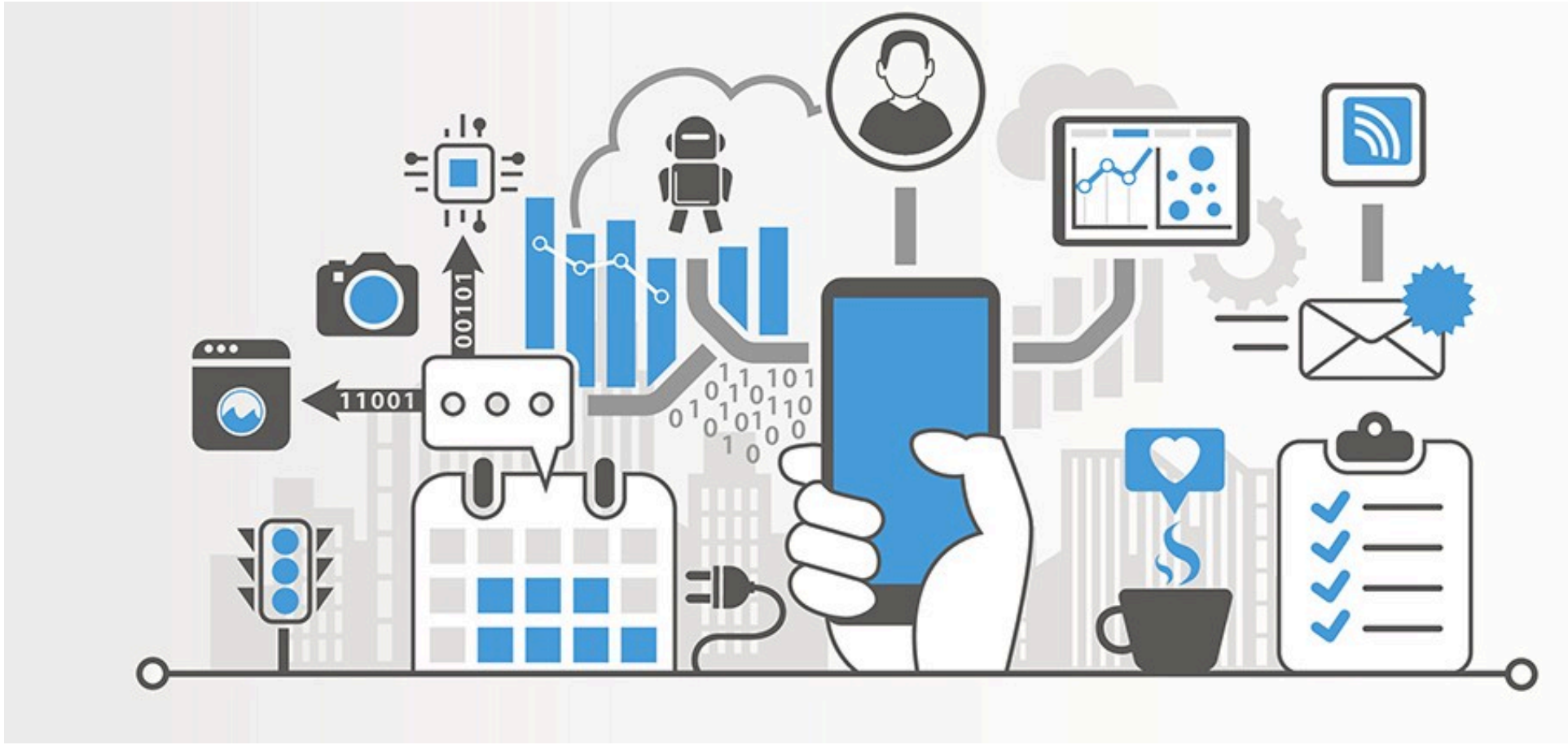
Programming for IoT Applications

Edoardo Patti

Lecture 3



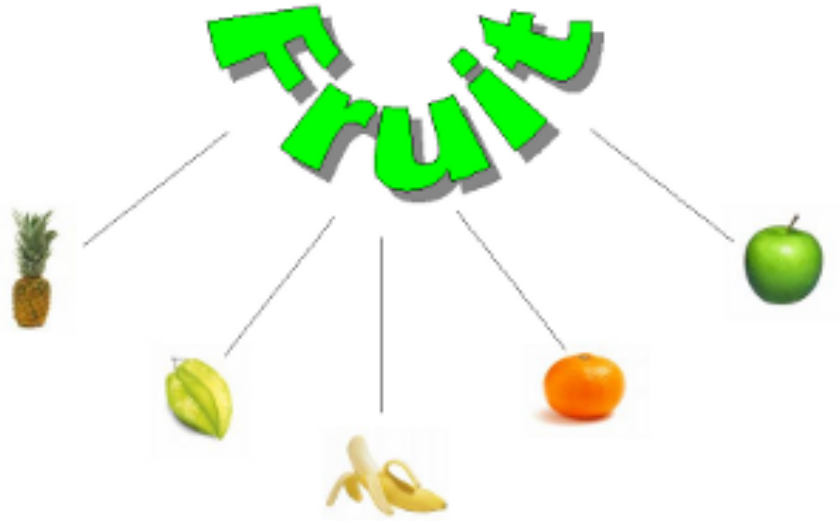
Python programming

OBJECT ORIENTED PROGRAMMING

Object Oriented Programming

- An **object** is a software item that contains **variables** and **methods**

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.

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 simply 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
- OOP allows:
  - programmers to (re)use large blocks of code
  - without knowing all the picture
- OOP makes code reuse a real possibility
- OO simplifies maintenance and evolution

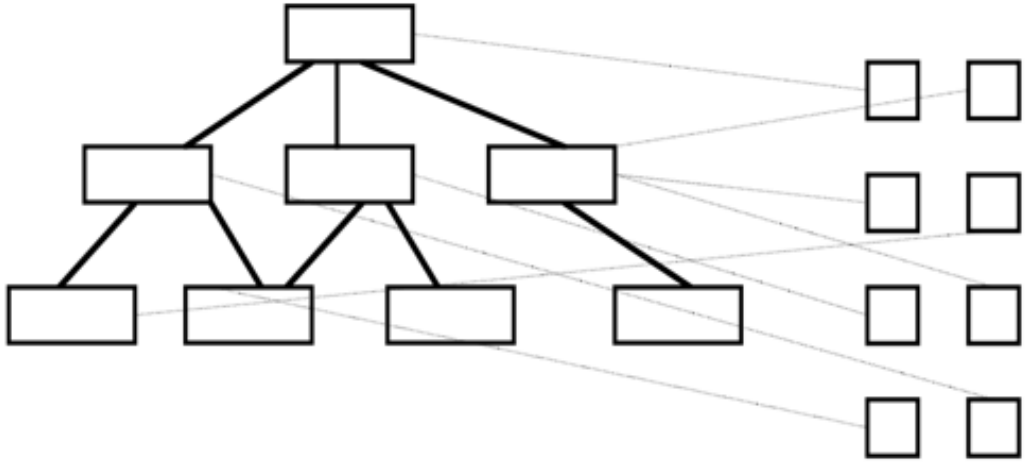
An Engineering Approach

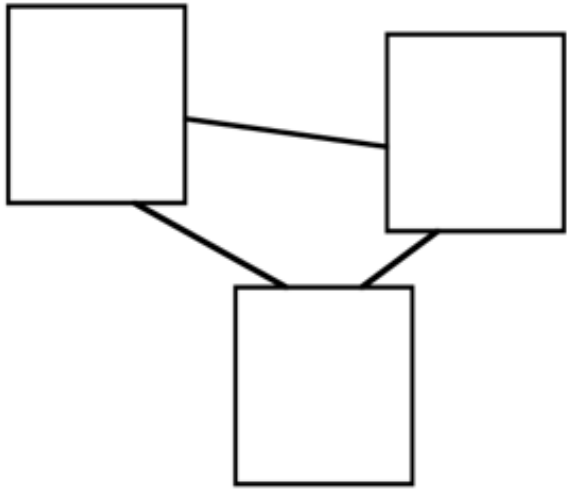
Given a system, with components and relationships among them, we have to:

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

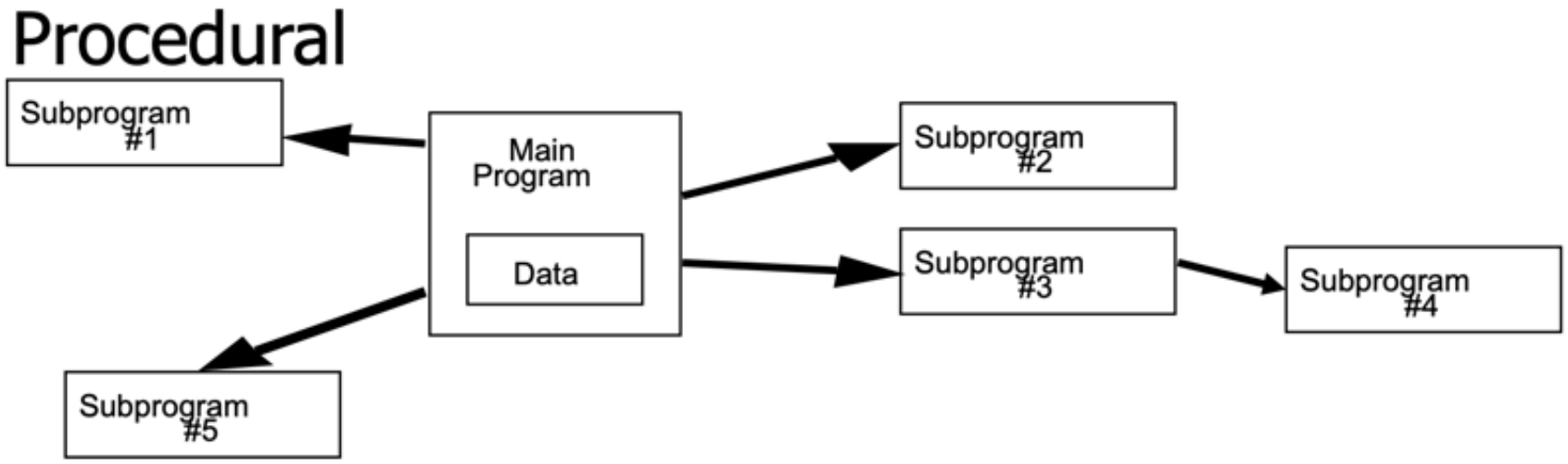
Object Oriented Design

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

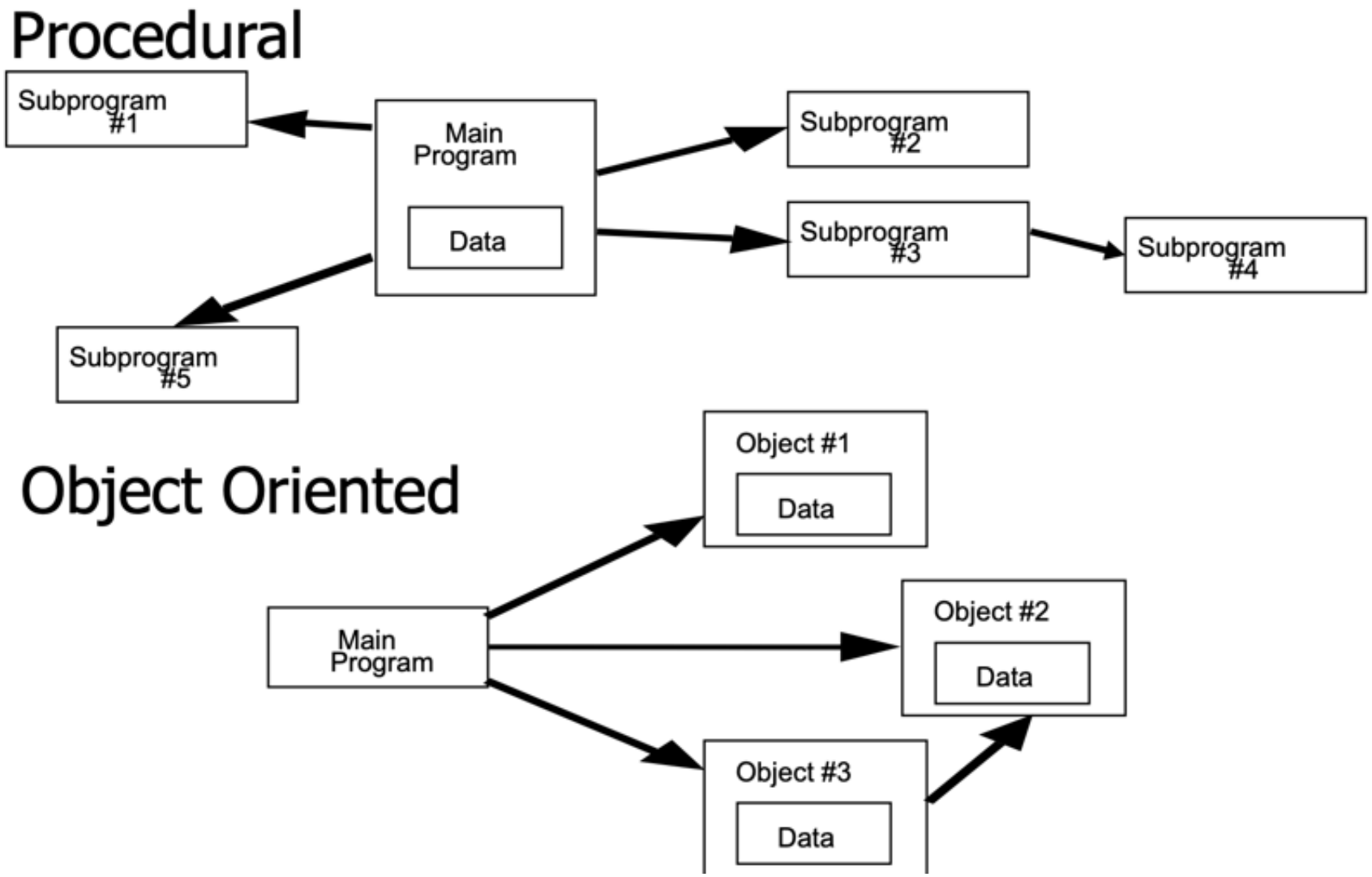




Procedural vs Object Oriented



Procedural vs Object Oriented



Object Oriented Approach

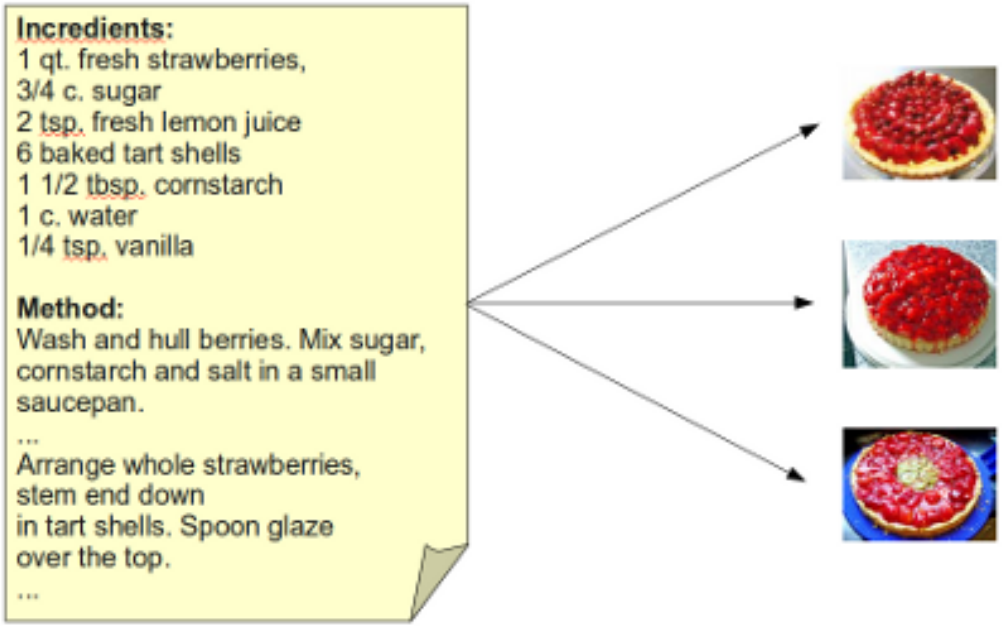
- Defines a new component type
  - Object (and class)
  - Both data and functions accessing it are within the same module
  - Allows defining a more precise interface

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

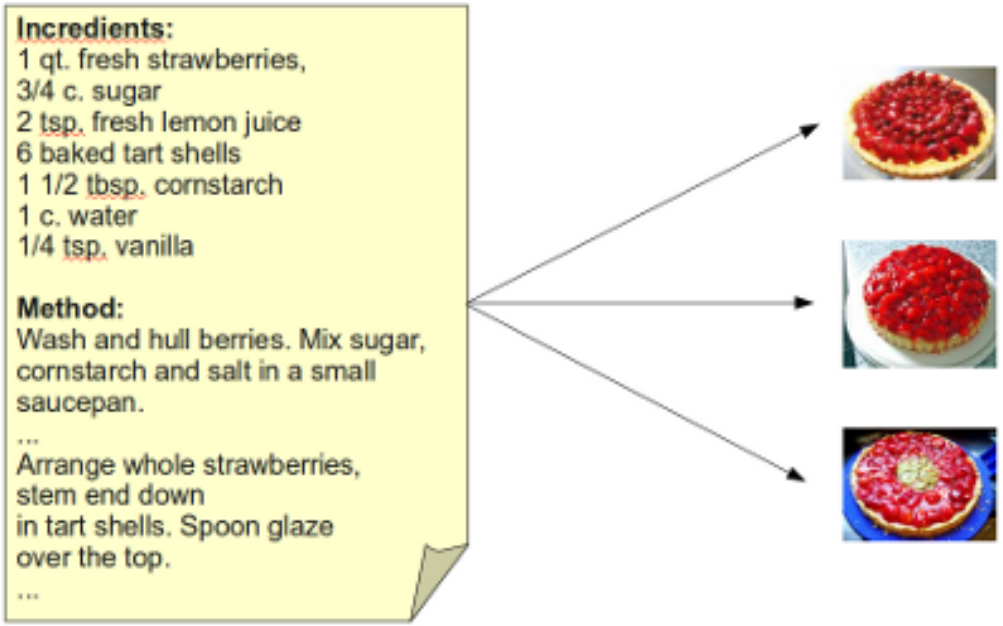
Classes

- Like baking a cake, an OOP program constructs objects according to the class definitions



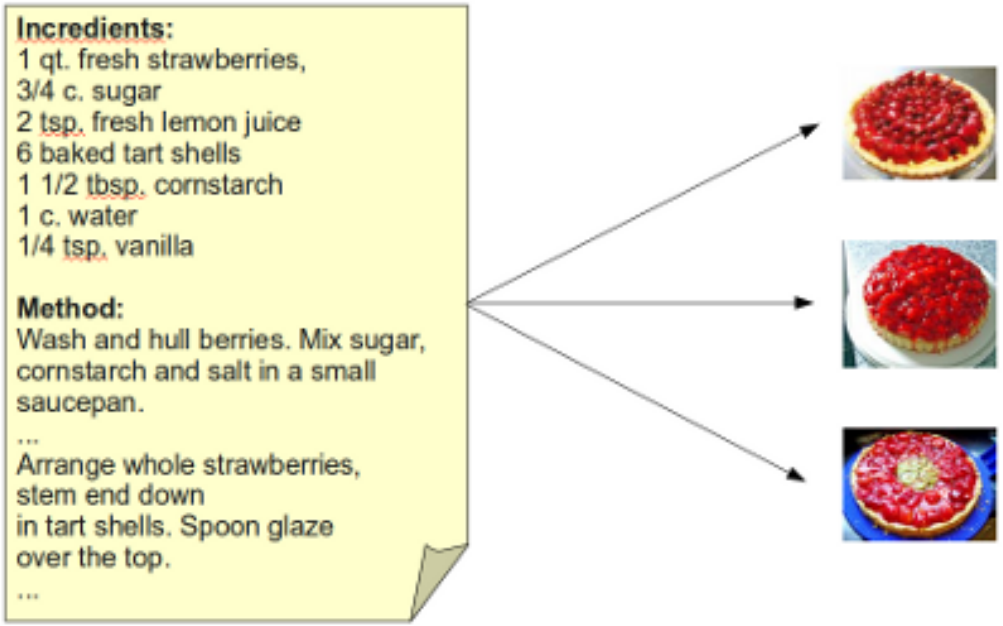
Classes

- 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.



Classes

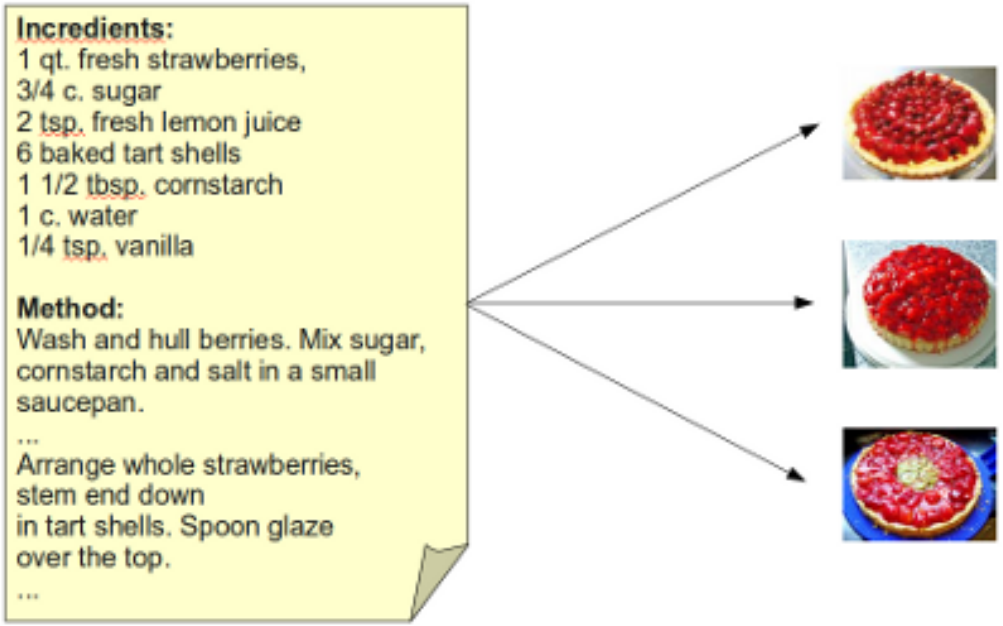
- Like baking a cake, an OOP program constructs objects according to the class definitions
- A class contains variables and methods - if you bake a cake you need ingredients and instructions to bake the cake.



- A class needs variables and methods - there are class variables, which have the same value in all methods and there are instance variables, which have normally different values for different objects

Classes

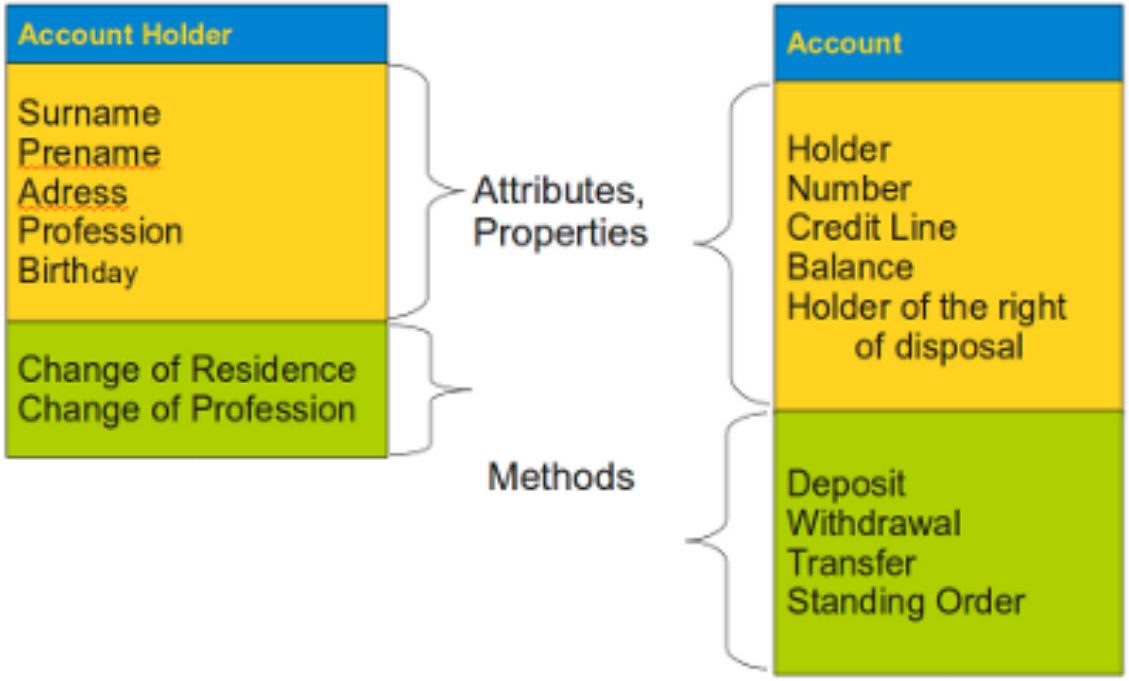
- Like baking a cake, an OOP program constructs objects according to the class definitions
- A class contains variables and methods - if you bake a cake you need ingredients and instructions to bake the cake.



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

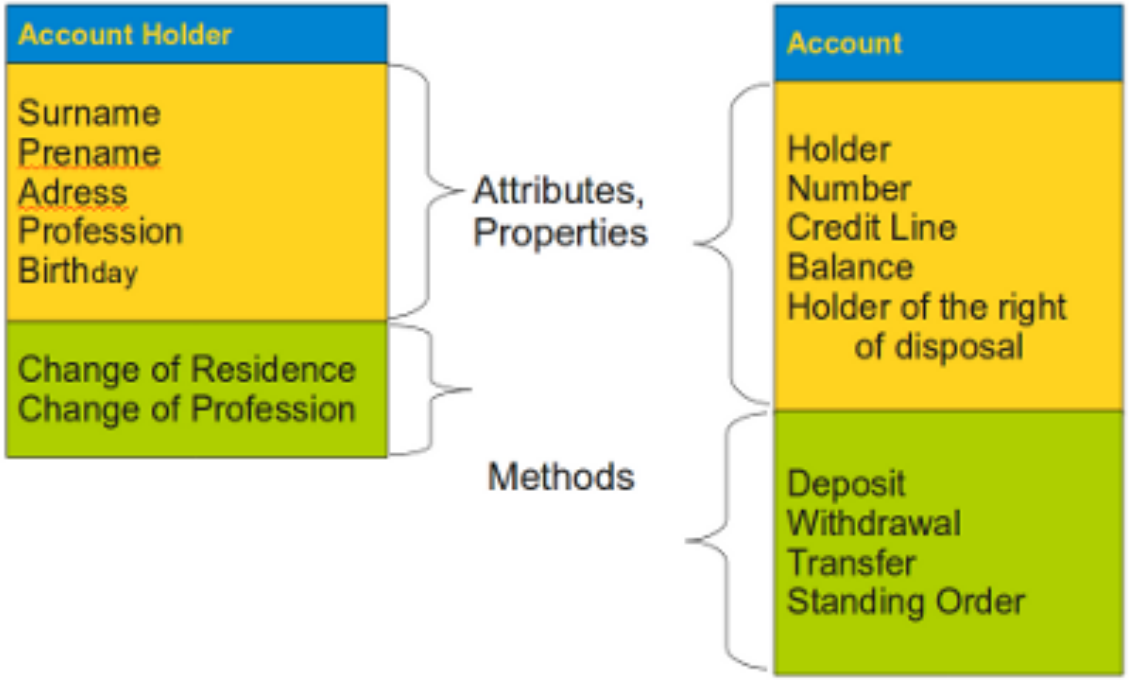

A class is like the
mold to create
different objects

Classes



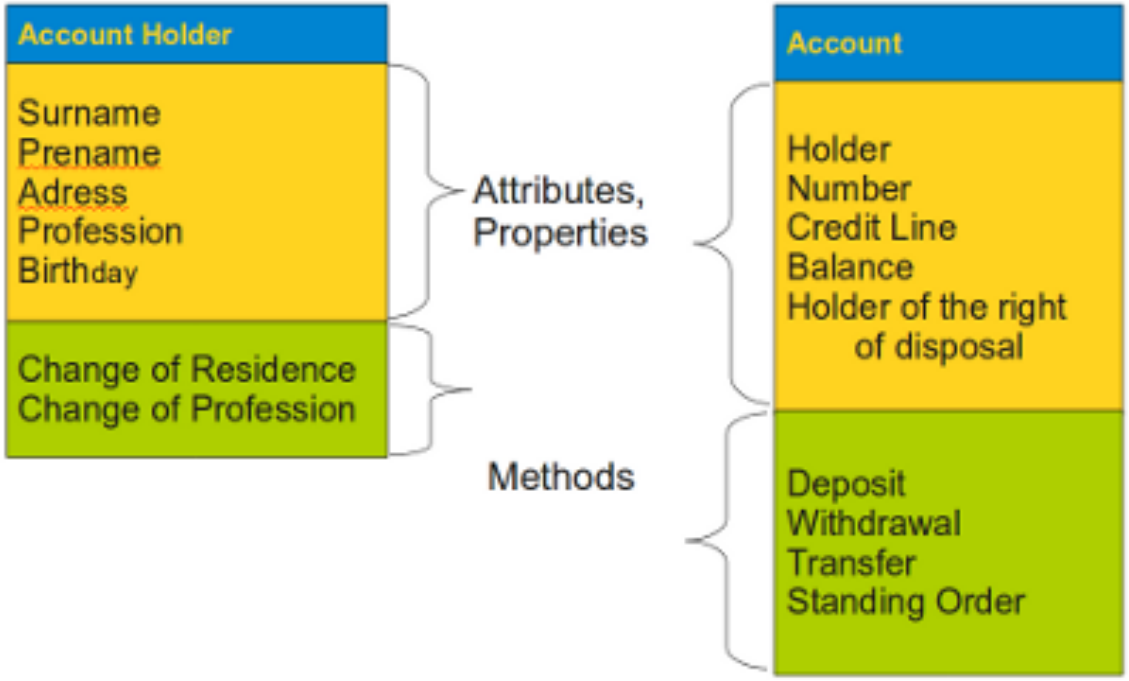
- A class defines a data type, which contains variables, properties and methods - a class describes the abstract characteristics of a real-life thing

Classes



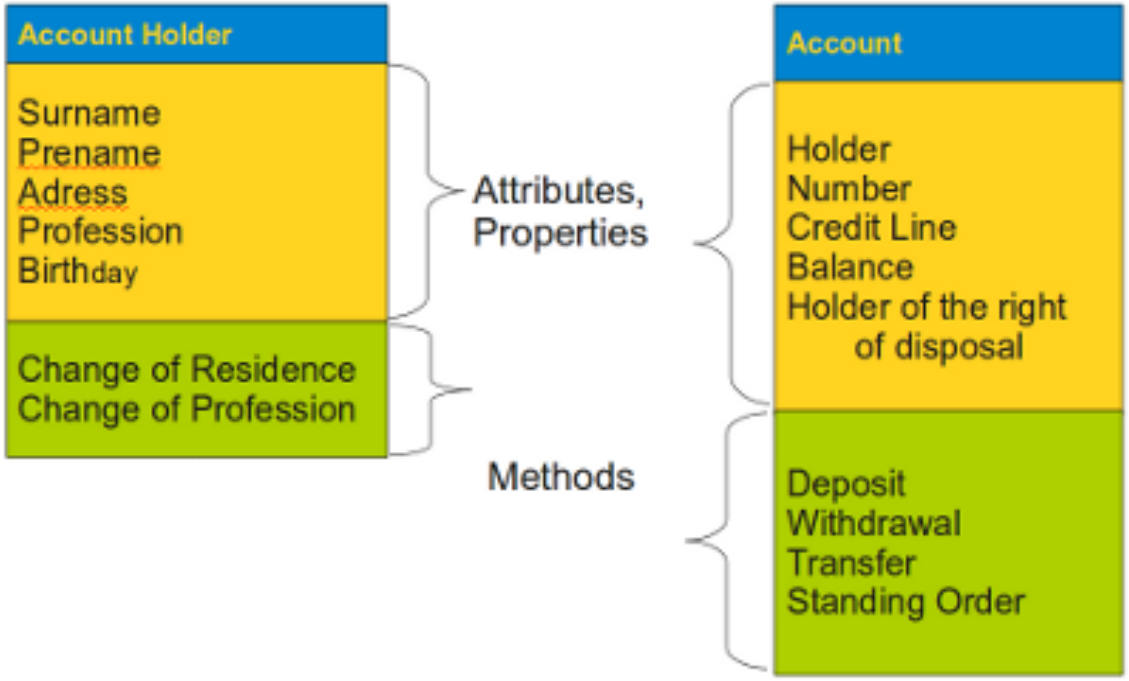
- A class defines a data type, which contains variables, properties and methods - a class describes the abstract characteristics of a real-life thing
- An instance is an object of a class created at run-time

Classes



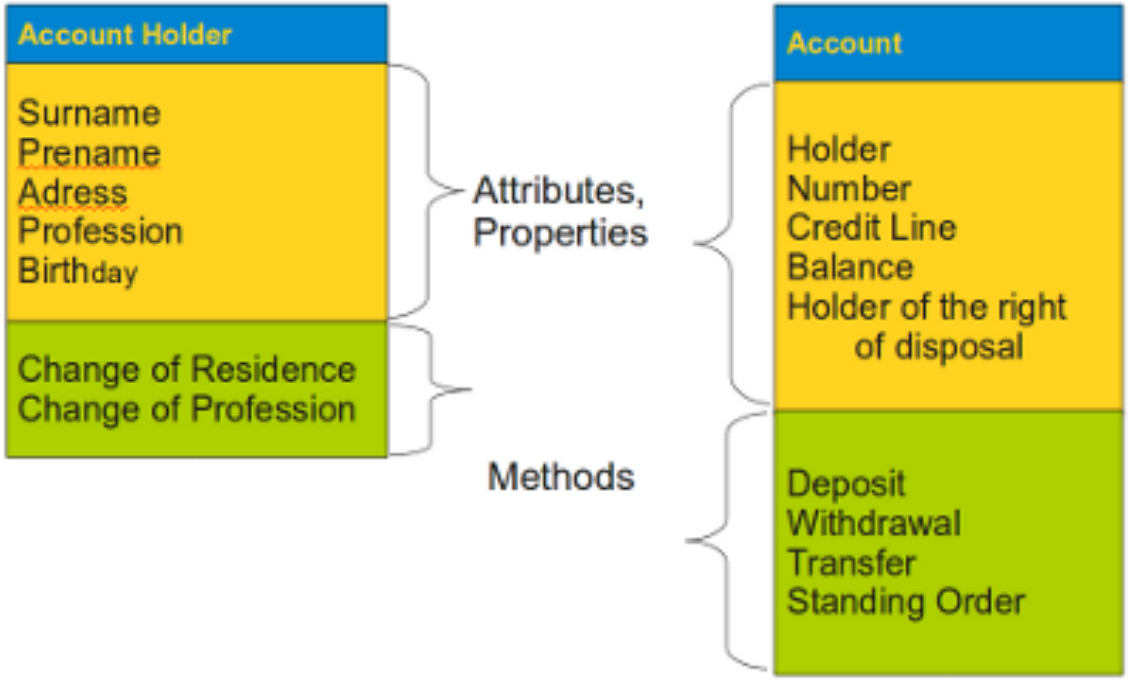
- A class defines a data type, which contains variables, properties and methods - a class describes the abstract characteristics of a real-life thing
- An instance is an object of a class created at run-time
- The set of values of the attributes of a particular object is called its state.

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

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

Objects

- Model of a physical or logical item
  - e.g. a student, an exam, a window
- Characterized by
  - identity
  - attributes (or data or properties or status)
  - 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

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

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

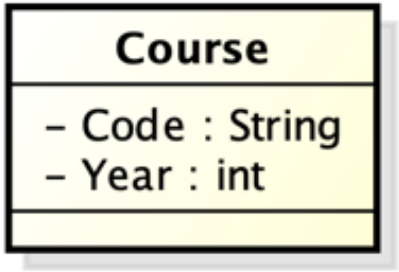
Class and Object

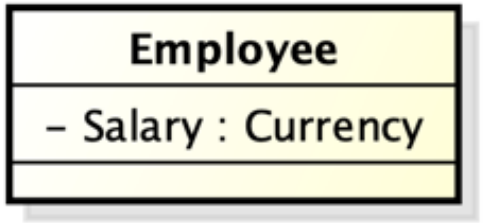
- A class is a type definition
  - Typically no memory is allocated until an object is created from the class
- The creation of an object is called instantiation. The created object is often called an instance
- There is no limit to the number of objects that can be created from a class
- Each object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

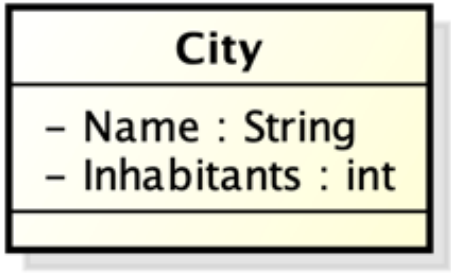
Attribute

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

Attribute


Course
- Code : String
- Year : int


Employee
- Salary : Currency


City
- Name : String
- Inhabitants : int

Methods

- Classes usually contain attributes and properties and methods for these instances and properties

Methods

- Classes usually contain attributes and properties and methods for these instances 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.

Methods

- Classes usually contain attributes and properties and methods for these instances 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 describe an operation that can be performed on an object

Methods

- Classes usually contain attributes and properties and methods for these instances 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 describe an operation that can be performed on an object
- Similar to functions in procedural languages

Methods

- Classes usually contain attributes and properties and methods for these instances 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 describe an operation that can be performed on an object
- Similar to functions in procedural languages
- A method represents the means to operat on o access to the attributes

Methods

- Classes usually contain attributes and properties and methods for these instances 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 describe an operation that can be performed on an object
- Similar to functions in procedural languages
- A method represents the means to operat on o access to the attributes
- Methods can only be called through instances of a class or a subclass, i.e. the class name followed by a dot and the method name

Message Passing

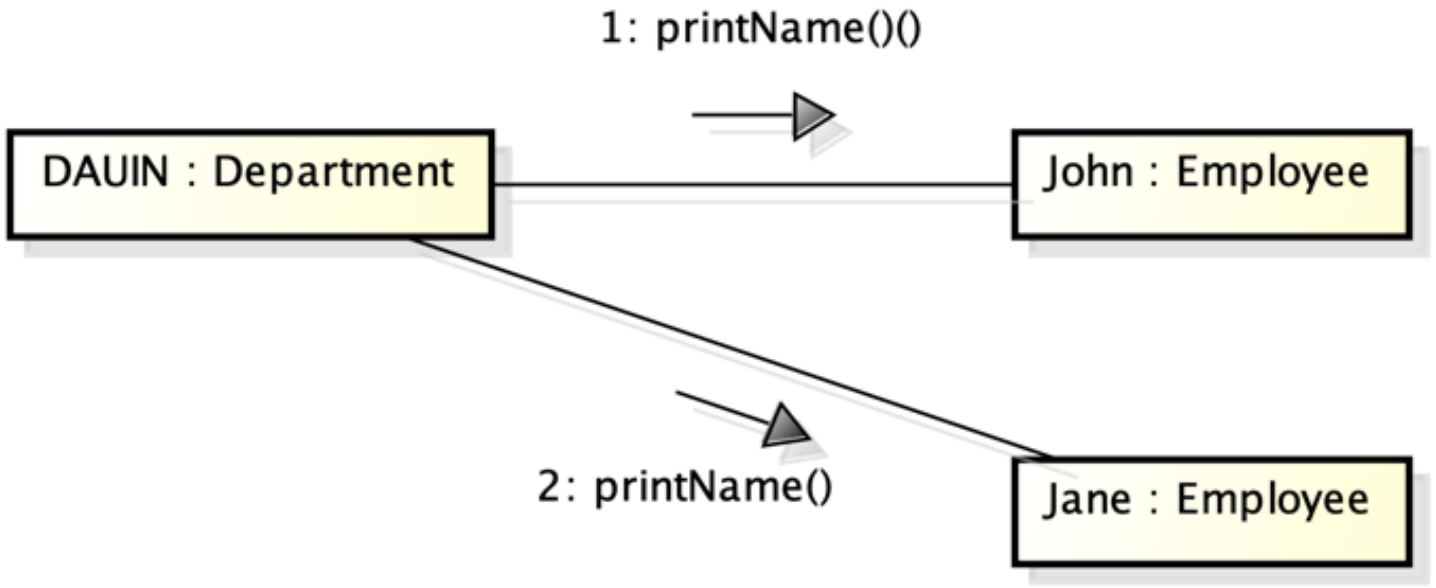
- Objects communicate by message passing
  - Not by direct access to object's local data
- A message is a service request

Message Passing

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

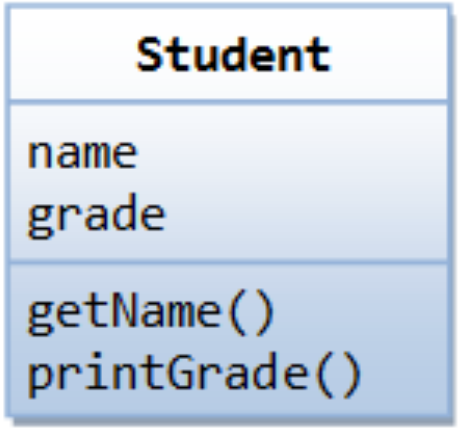
Note: this is an abstract view that is independent from specific programming languages.

Messages


1: printName()()
DAUIN : Department
John : Employee
2: printName()
Jane : Employee

Some Examples

Classname
(Identifier)
Data Member
(Static attributes)
Member Functions
(Dynamic Operations)



Some Examples

Classname
(Identifier)
Data Member
(Static attributes)
Member Functions
(Dynamic Operations)

Student
name grade
getName() printGrade()

Circle
radius color
getRadius() getArea()

Some Examples

Classname
(Identifier)
Data Member
(Static attributes)
Member Functions
(Dynamic Operations)

Student
name grade
getName() printGrade()

Circle
radius color
getRadius() getArea()

SoccerPlayer
name number xLocation yLocation
run() jump() kickBall()

Some Examples

Classname
(Identifier)
Data Member
(Static attributes)
Member Functions
(Dynamic Operations)

Student
name grade
getName() printGrade()

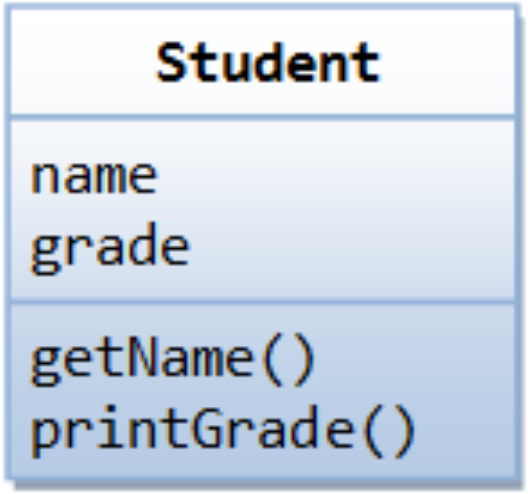
Circle
radius color
getRadius() getArea()

SoccerPlayer
name number xLocation yLocation
run() jump() kickBall()

Car
plateNumber xLocation yLocation speed
move() park() accelerate()

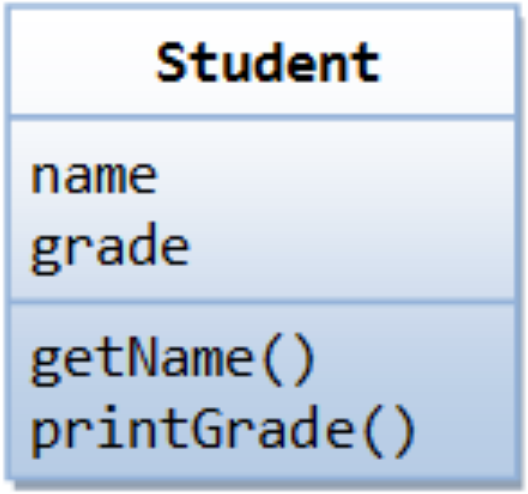
Some Examples

Class Definition

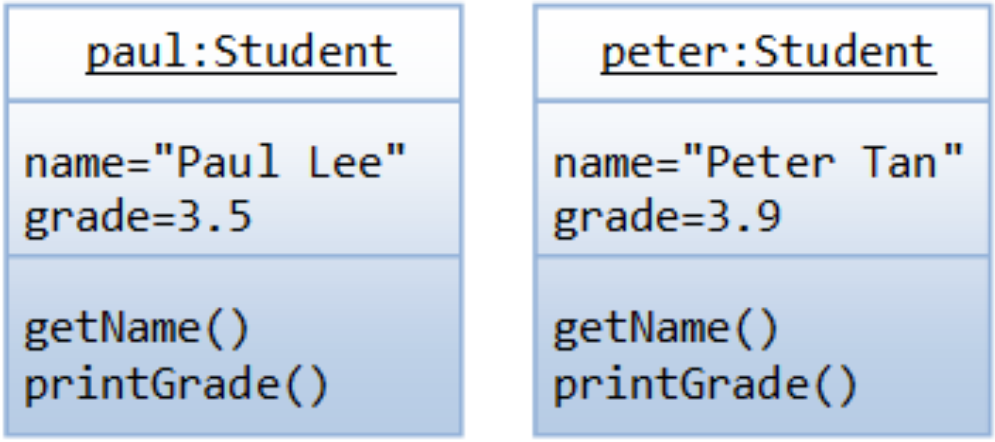


Some Examples

Class Definition

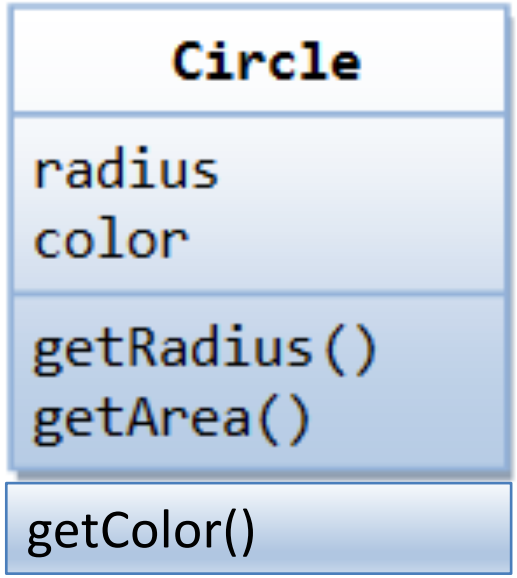


Instances



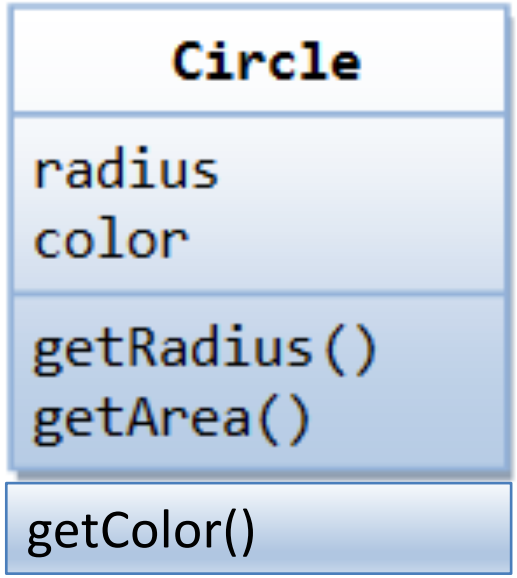
Some Examples

Class Definition

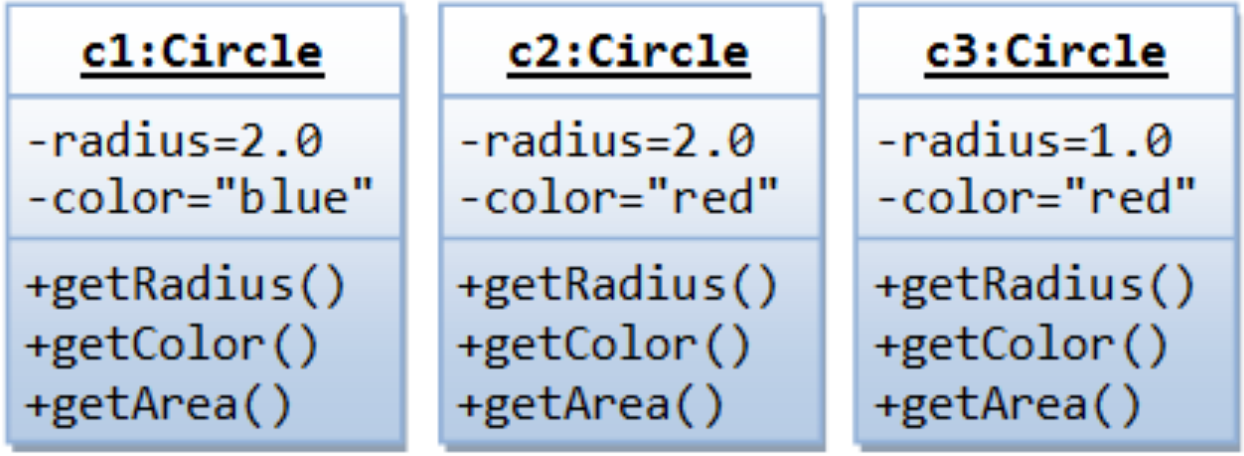


Some Examples

Class Definition



Instances



Some Exercises

Define a class to describe:

- A person
- A dog
- A car

Definition of classes

```
class name:
   "documentation"
   statements
    …
```

-or-

```
class name(base1, base2, ...):
   "documentation"
   def __init__(self, arg1, arg2, ...):
       self.x = base1
       self.y = base1
    …
   def name(self, arg1, arg2, ...):
       …
```

Definition of classes

```
class name:
    "documentation"
    statements
     …
```

-or-

```
class name(base1, base2, ...):
    "documentation"
    def __init__(self, arg1, arg2, ...):
        self.x = base1
        self.y = base1
     …
    def name(self, arg1, arg2, ...):
        …
```

Class definition

Definition of classes

```
class name:
   "documentation"
   statements
    …
```

-or-

```
class name(base1, base2, ...):
   "documentation"
   def __init__(self, arg1, arg2, ...):
       self.x = base1
       self.y = base1
    …
   def name(self, arg1, arg2, ...):
       …
```

statements can be **method definitions**

Definition of classes

```
class name:
   "documentation"
   statements
    ...
```

-or-

```
class name(base1, base2, ...):
   "documentation"
   def __init__(self, arg1, arg2, ...):
       self.x = base1
       self.y = base1
    ...
   def name(self, arg1, arg2, ...):
       ...
```

__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.

Definition of classes

```
class name:
   "documentation"
   statements
    …
```

-or-

```
class name(base1, base2, ...):
   "documentation"
   def __init__(self, arg1, arg2, ...):
       self.x = base1
       self.y = base1
    …
   def name(self, arg1, arg2, ...):
       …
```

statements can be **class variable assignments**

Object Methods

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

Object Methods

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

- `self` *must* be the first parameter to any object method
  - represents the "implicit parameter" (`this` in Java)

Object Methods

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

- `self` *must* be the first parameter to any object method
  - represents the "implicit parameter" (`this` in Java)
- *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

```
if __name__ == “__main__”:     # define the main

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

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

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

Example

```
if __name__ == "__main__":     # define the main

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


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

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

To create an instance, simply call the class object

Example

```
if __name__ == “__main__”:     # define the main

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

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

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

To use methods of the instance, call using dot notation

Example

```
if __name__ == "__main__":     # define the main

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

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

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

To inspect instance variables, use dot notation

Fields

name = value

- Example:

```
class Point():
    def __init__(self):
      self.x = 0
      self.y = 0

# main
if __name__ == "__main__":
      p1 = Point()
      p1.x = 2
      p1.y = -5
```

- can be declared directly inside class (as shown here) or in constructors (more common)

Import a Class

`import` **class**

– client programs must import the classes they use

point_main.py

```
1  from Point import *
2
3  # main
4  if __name__ == "__main__":
5      p1 = Point()
6      p1.x = 7
7      p1.y = -3
8      ...
9
10 # Python objects are dynamic (can add fields any time!)
       p1.name = "Tyler Durden"
```

Example Class

Exercise: Write `distance`, `set_location`, and `distance_from_origin` methods.

point.py

```
 1  from math import *
 2
 3  class Point:
 4
 5      def __init__(self, x, y):
 6          self.x = x
 7          self.y = y
 8
 9      def distance_from_origin(self):
10          return sqrt(self.x * self.x + self.y * self.y)
11
12      def distance(self, other):
13          dx = self.x - other.x
14          dy = self.y - other.y
15          return sqrt(dx * dx + dy * dy)
16
17
```

Complete Point Class

point.py

```
 1  from math import *
 2
 3  class Point:
 4      def __init__(self, x, y):
 5          self.x = x
 6          self.y = y
 7
 8      def distance_from_origin(self):
 9          return sqrt(self.x * self.x + self.y * self.y)
10
11      def distance(self, other):
12          dx = self.x - other.x
13          dy = self.y - other.y
14          return sqrt(dx * dx + dy * dy)
15
16      def translate(self, dx, dy):
17          self.x += dx
18          self.y += dy
19
20      def __str__(self):
21          return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Main Principles of OOP

- Interface
- Encapsulation
- Inheritance
- Polymorphism

Interface

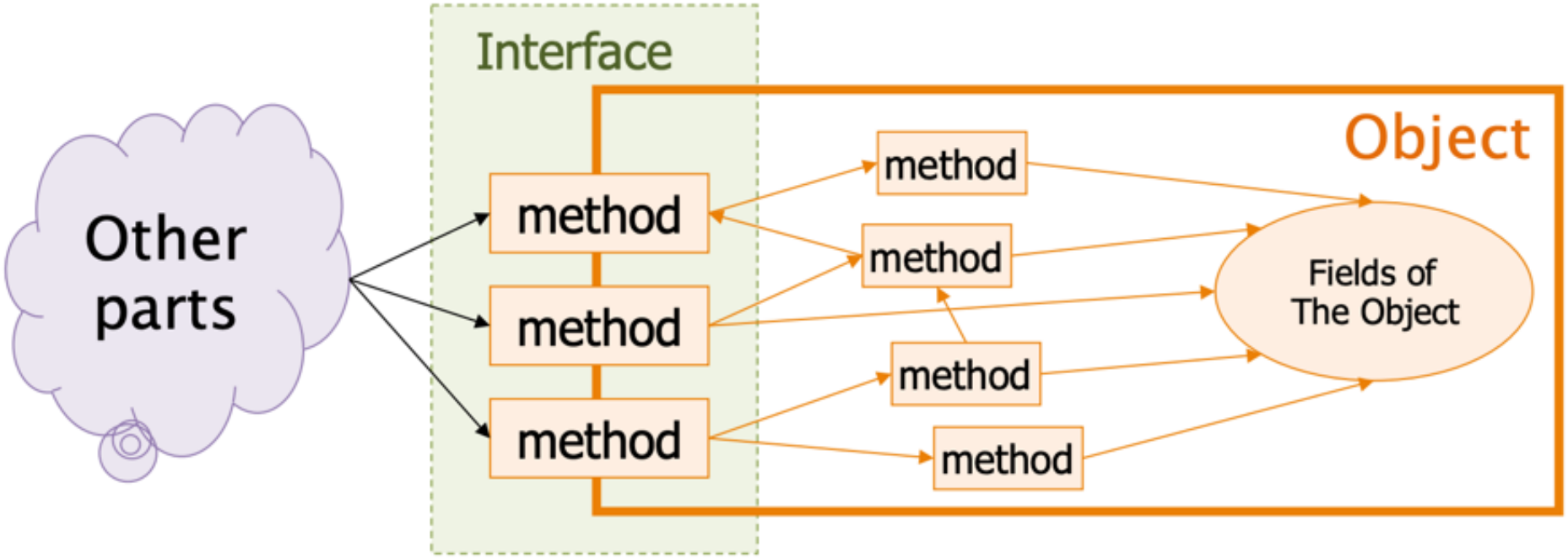
- An interface is a set of messages an object can receive
  - Each message is mapped to an internal “function” within the object
  - The object is responsible for the association (message -> function)
  - Any other message is illegal

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

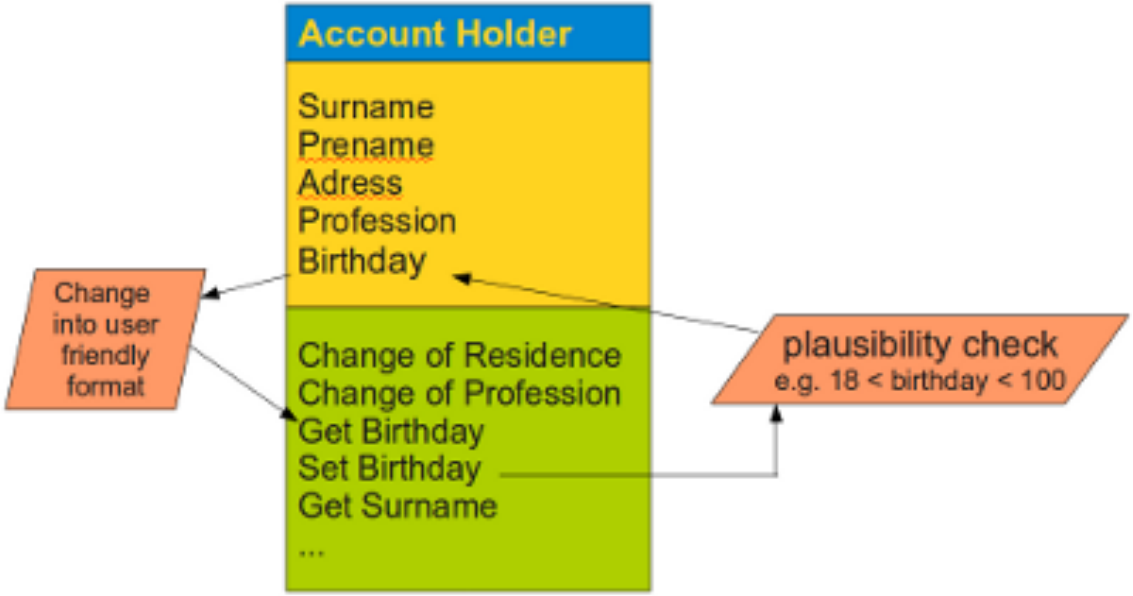
Interface

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



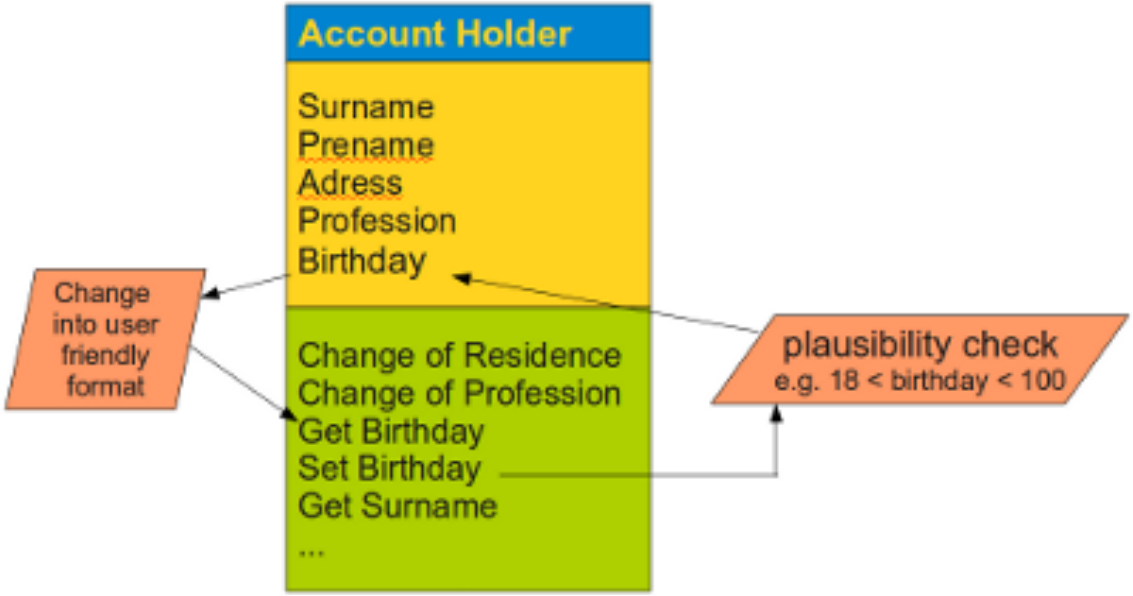
Encapsulation

- Dividing the code into a public interface, and a private implementation of that interface



Encapsulation

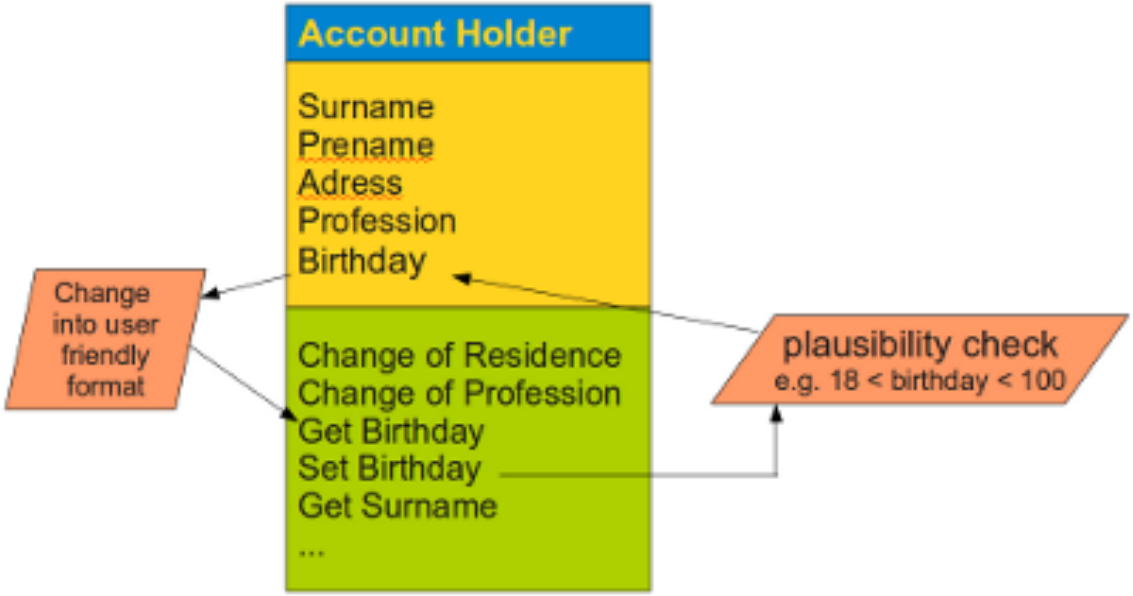
- Dividing the code into a public interface, and a private implementation of that interface



- 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 objects definition

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 objects 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

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

Encapsulation

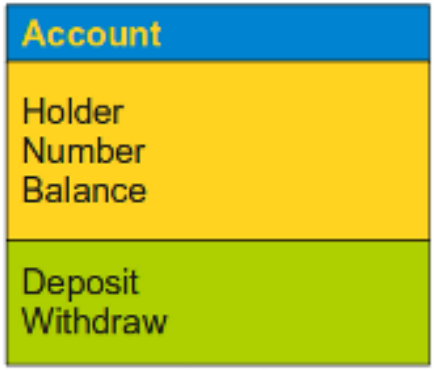
- Simplified access
  - To use an object, the user need only comprehend the interface. No knowledge of the internals are necessary
- Self-contained
  - Once the interface is defined, the programmer can implement the interface (write the object) without interference of others
- Ease of evolution
  - Implementation can change at a later time without rewriting any other part of the program (as long as the interface does not change)

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 at a later time 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)

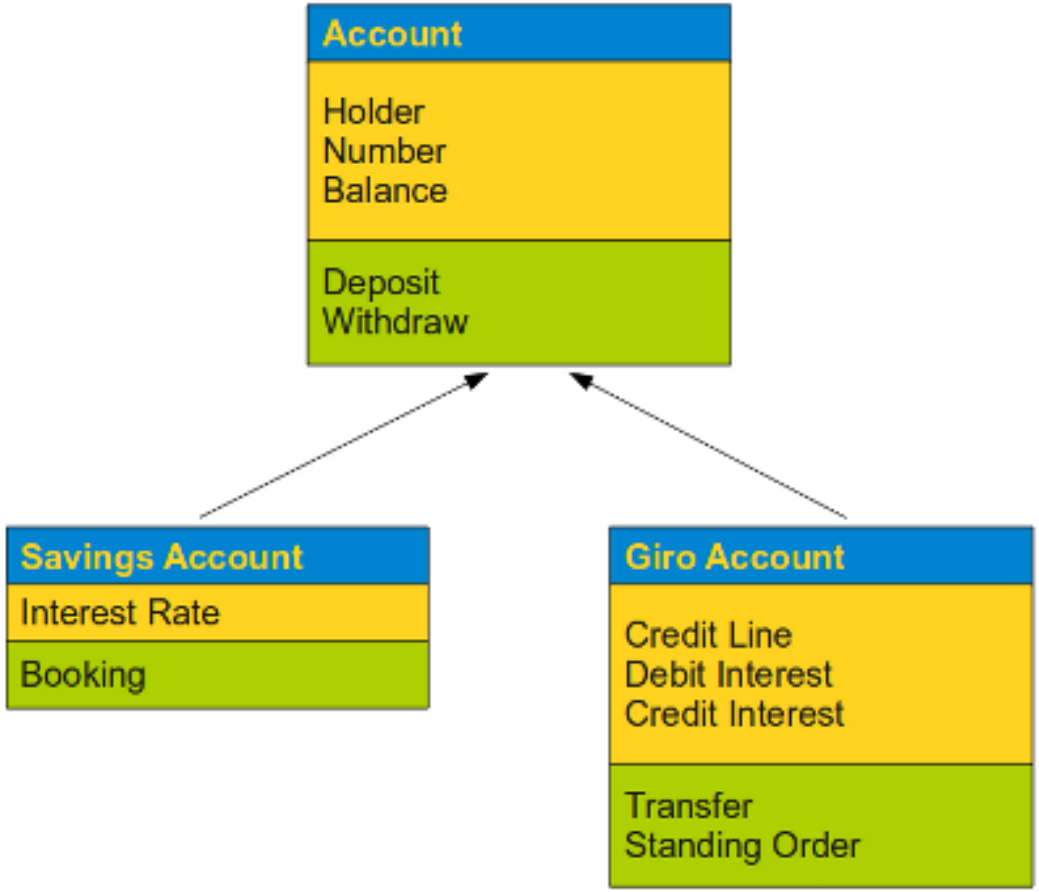
Inheritance

- Classes can inherit other classes - a class can inherit attributes and behaviour (methods) from other classes, called super-classes



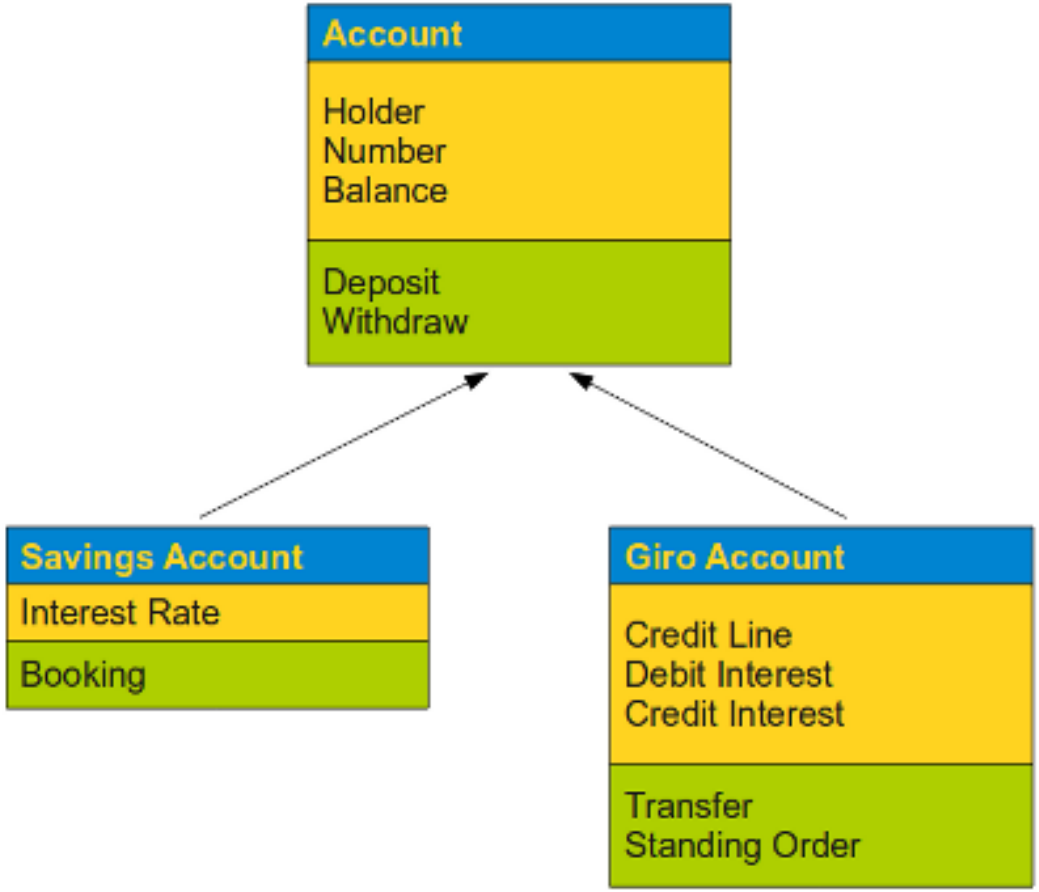
Inheritance

- 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



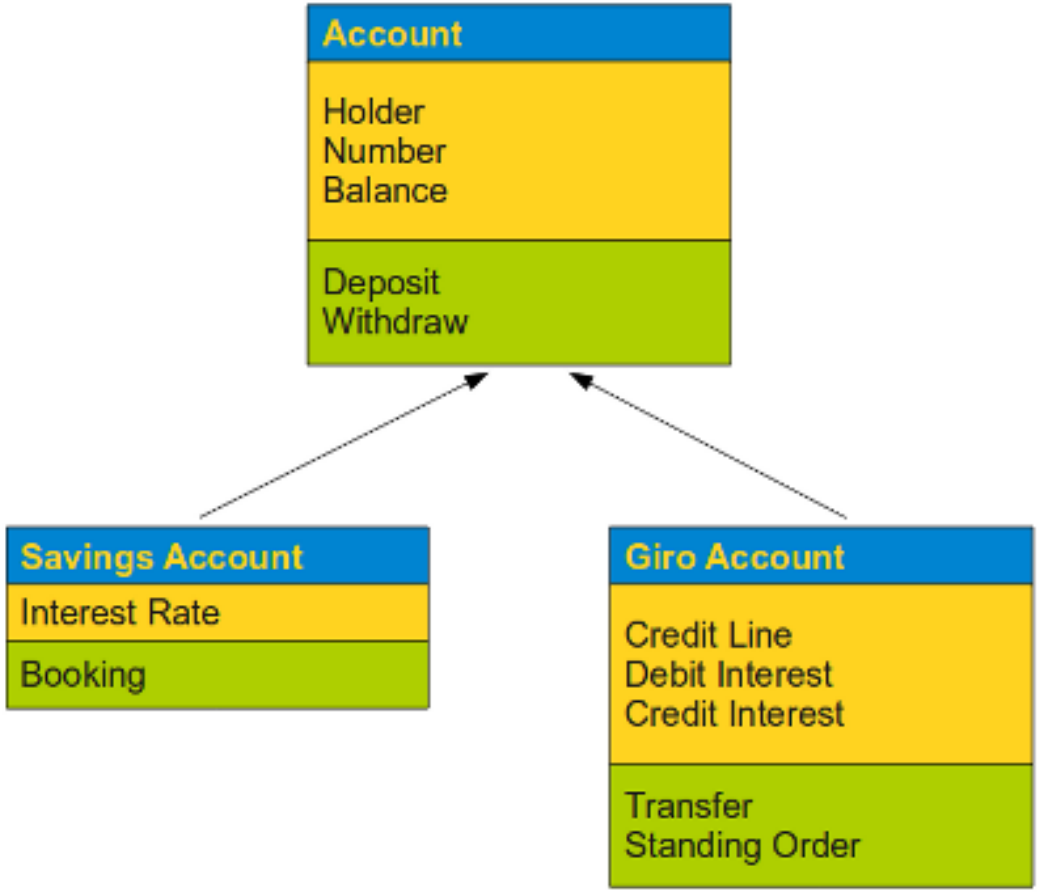
Inheritance

- 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

- 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

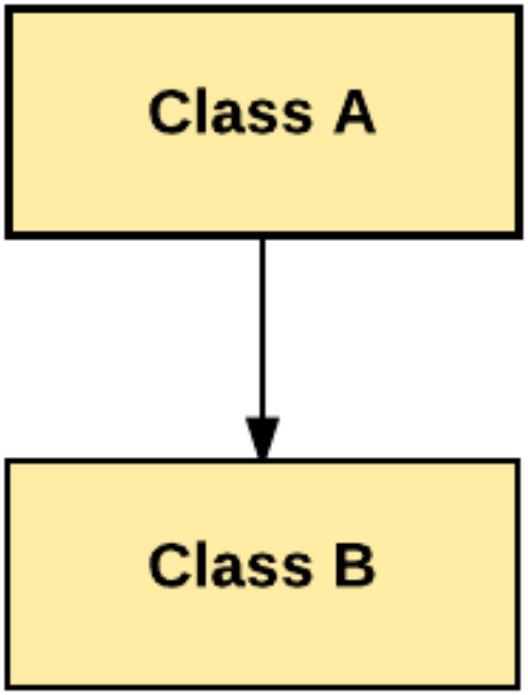


- Inheritance is used to create new classes by using existing classes - new ones can both be created by extending and by restricting the existing classes

Type of Inheritance

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

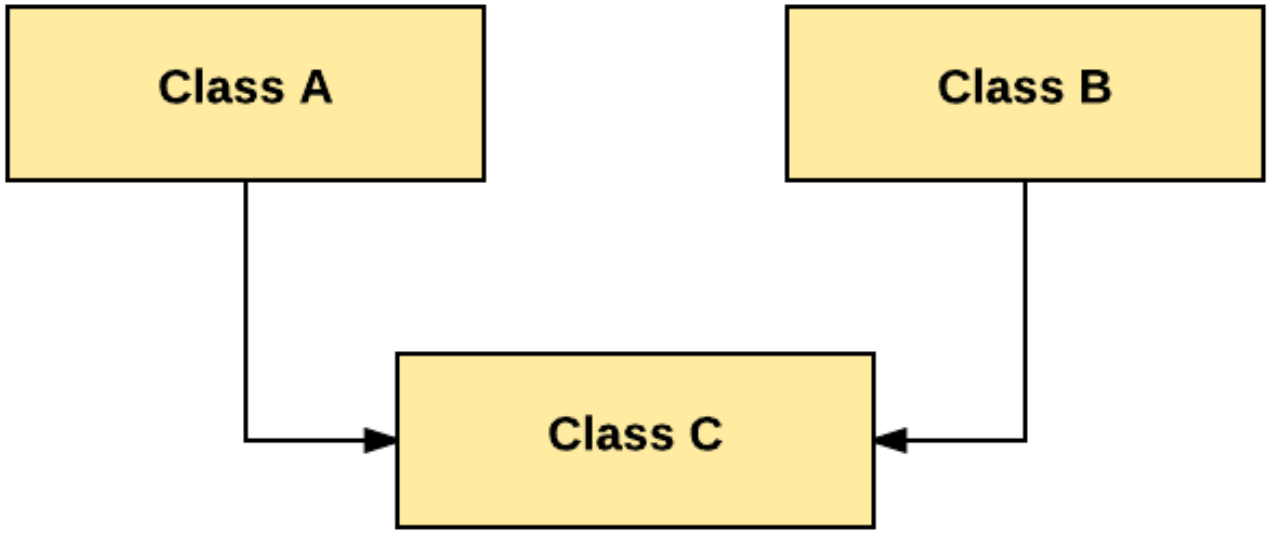
- Class B extends only Class A
- Class A is a super class and Class B is a Sub-class



Type of Inheritance

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

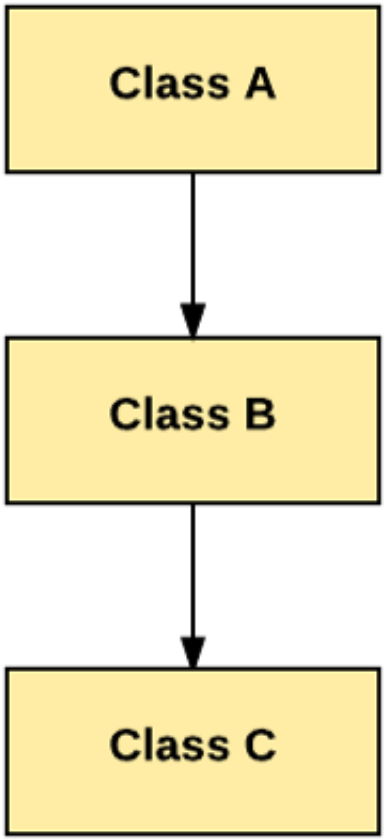
- Class C extends Class A and Class B both



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.

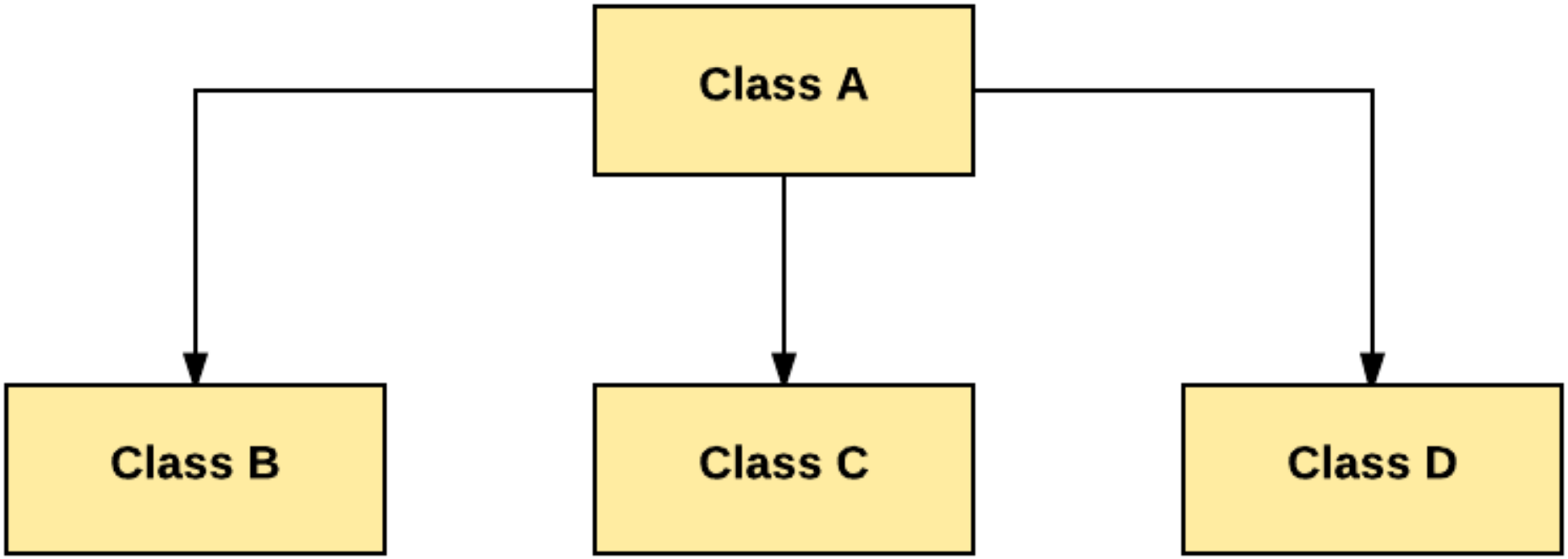
- Class C is subclass of B and B is a of subclass Class A



Type of Inheritance

In **Hierarchical Inheritance**, one class is inherited by many sub classes.

- Class B, C, and D inherit the same class



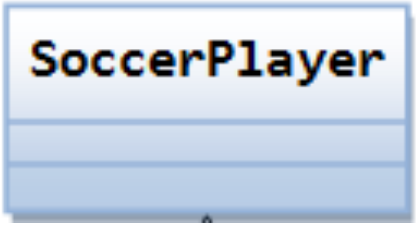
Why Inheritance?

- Frequently, a class is merely a modification of another class. In this way, there is minimal repetition of the same code

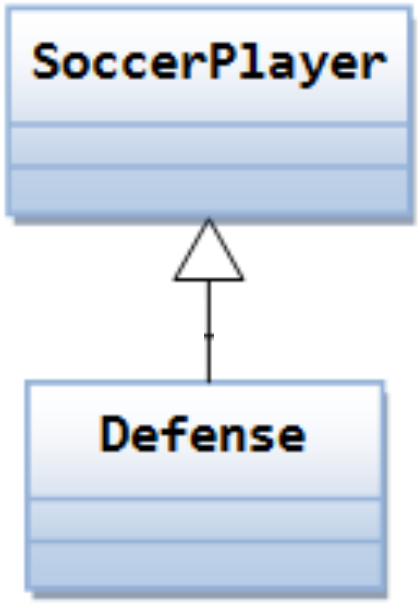
Why Inheritance?

- 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

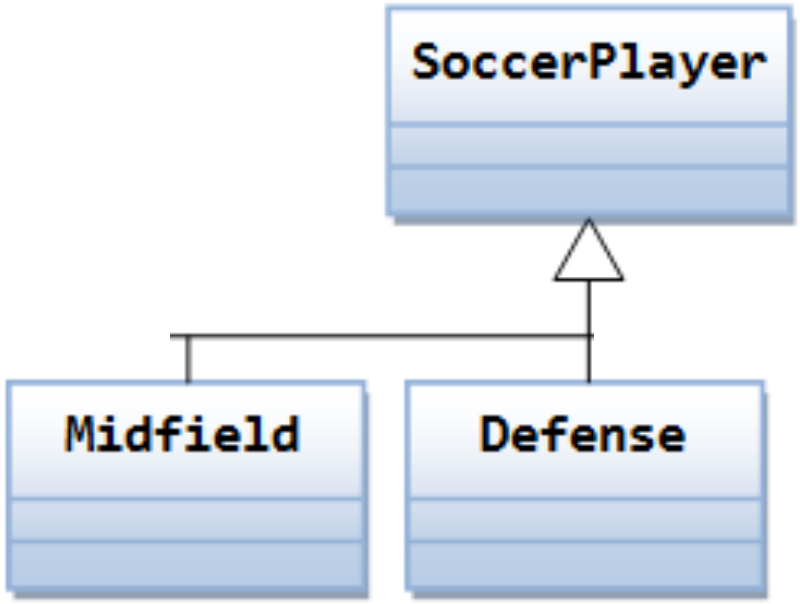


Some Examples



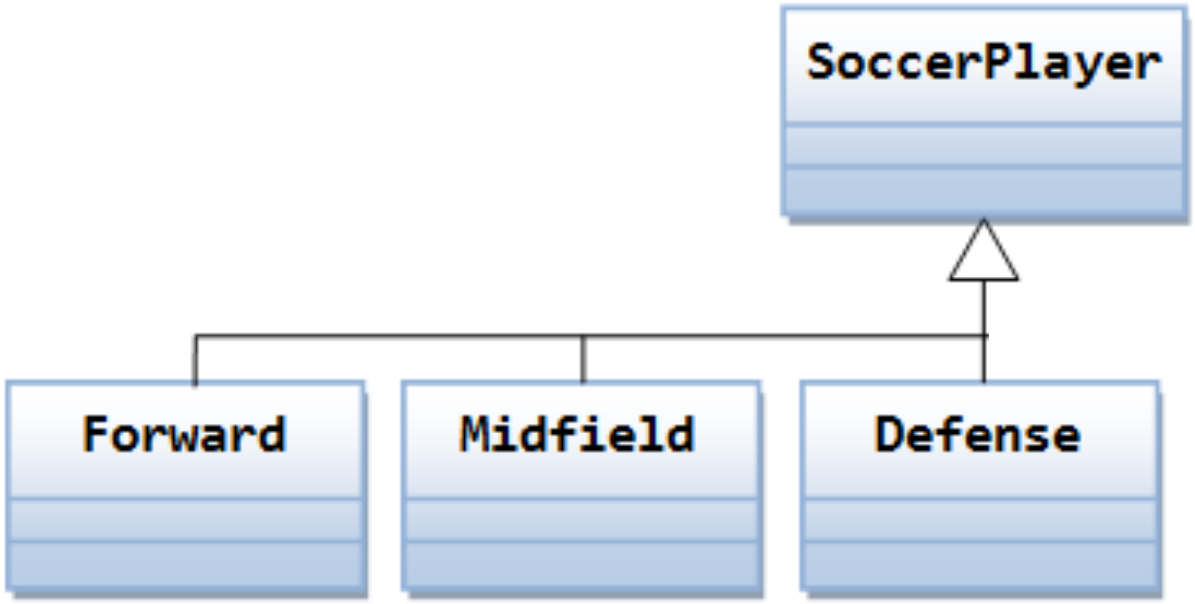
Single Inheritance

Some Examples



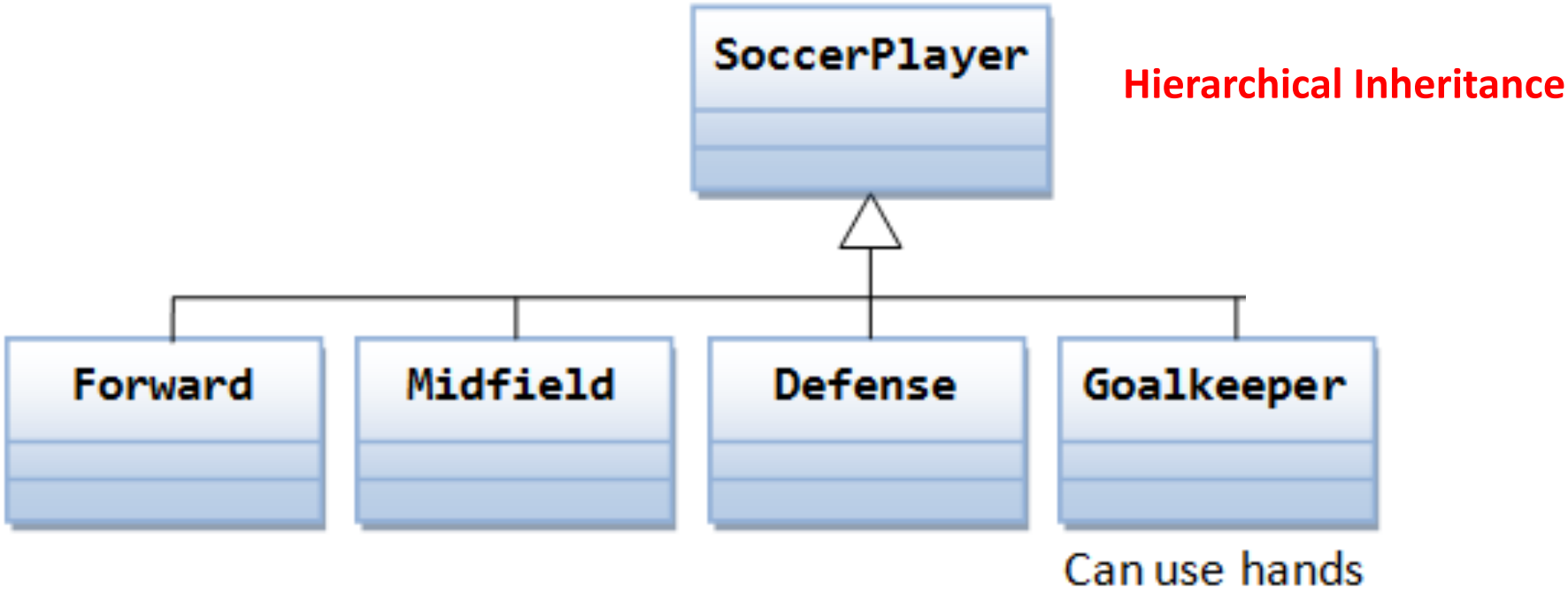
Hierarchical Inheritance

Some Examples



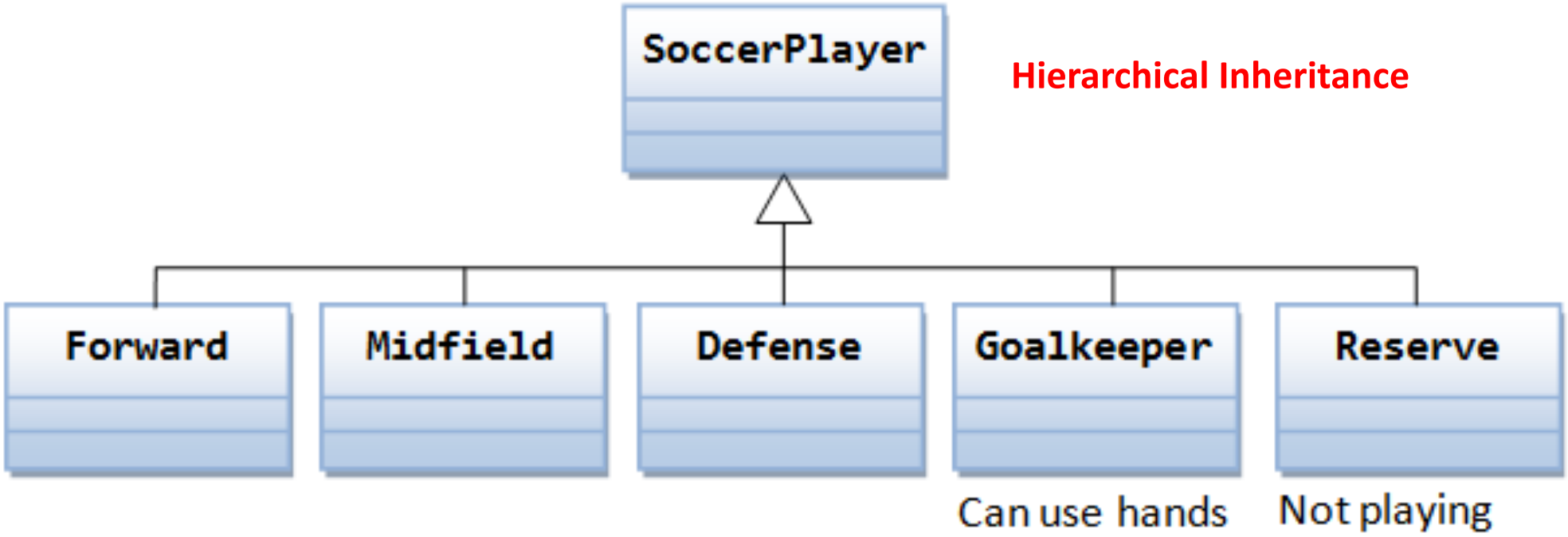
Hierarchical Inheritance

Some Examples



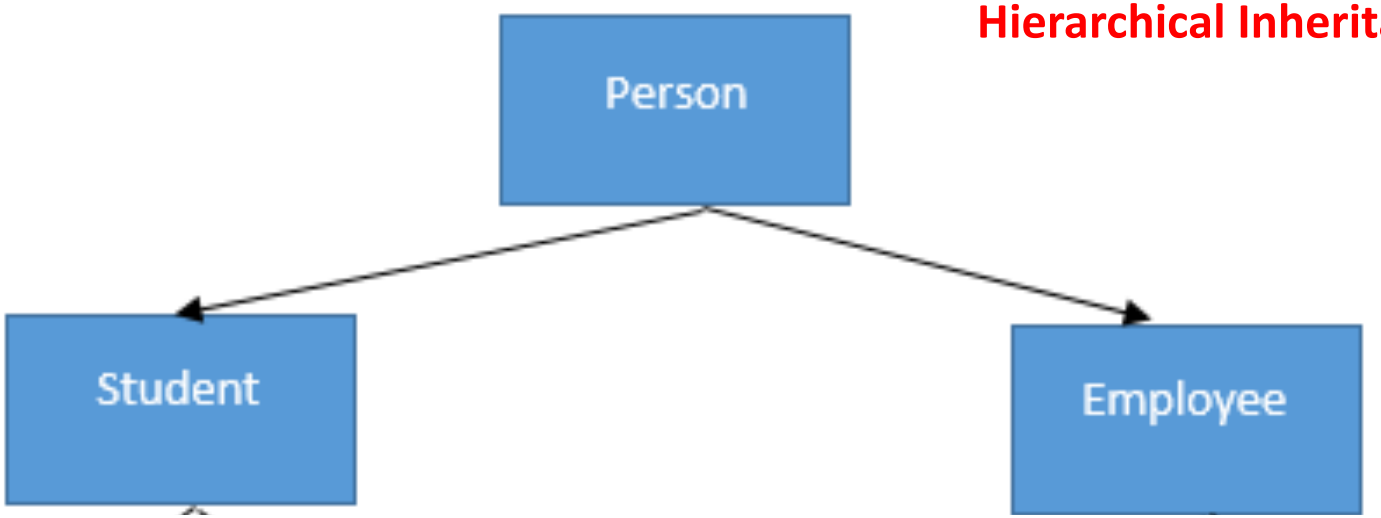
Hierarchical Inheritance

Some Examples



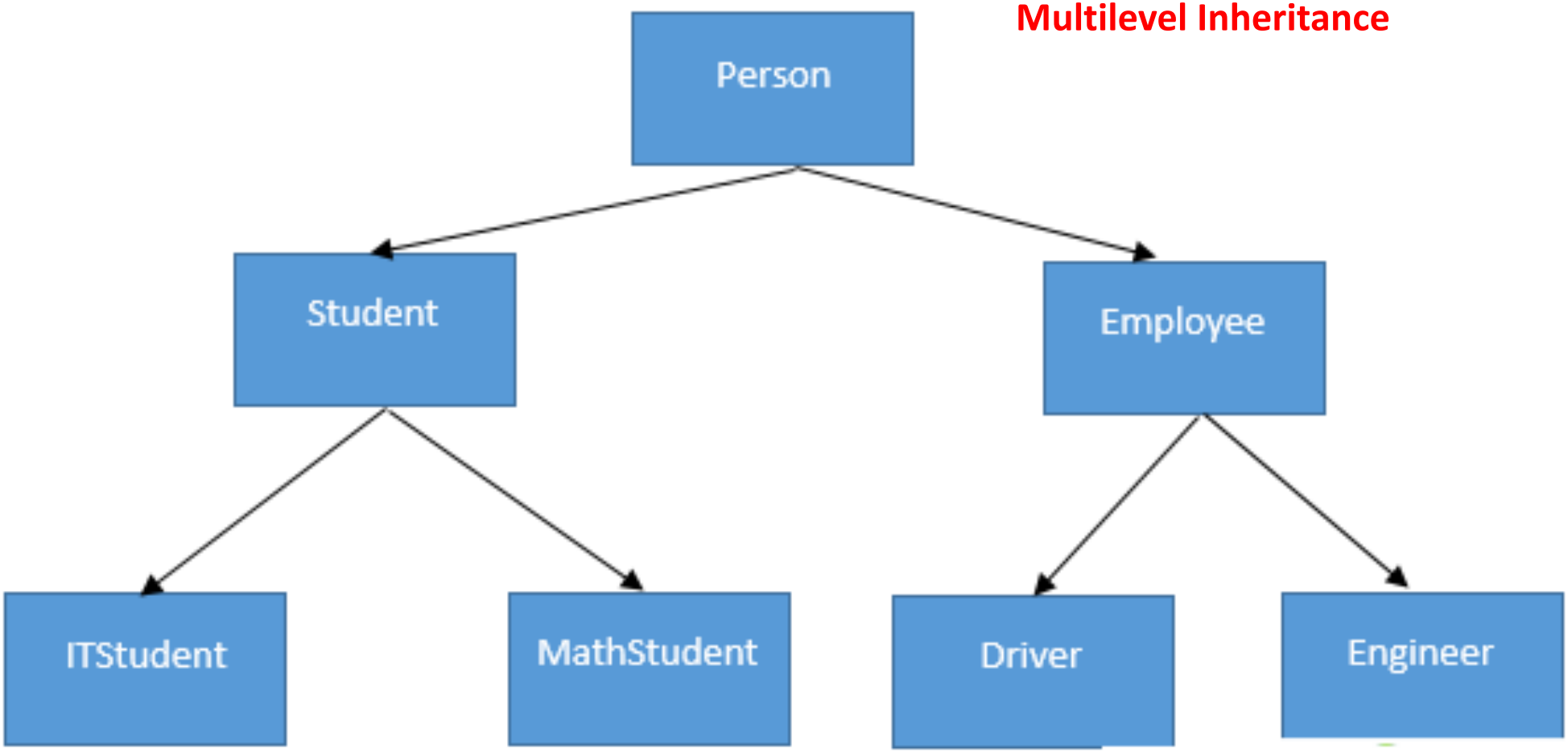
Some Examples

Hierarchical Inheritance

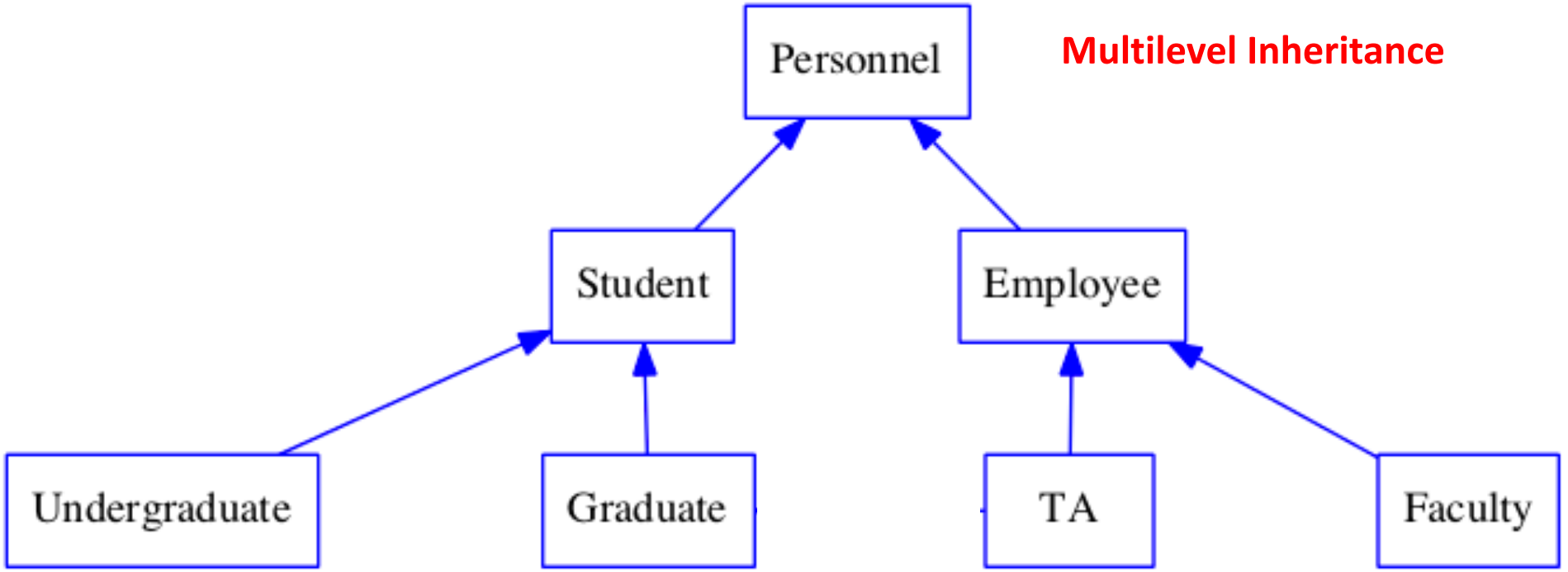


Some Examples

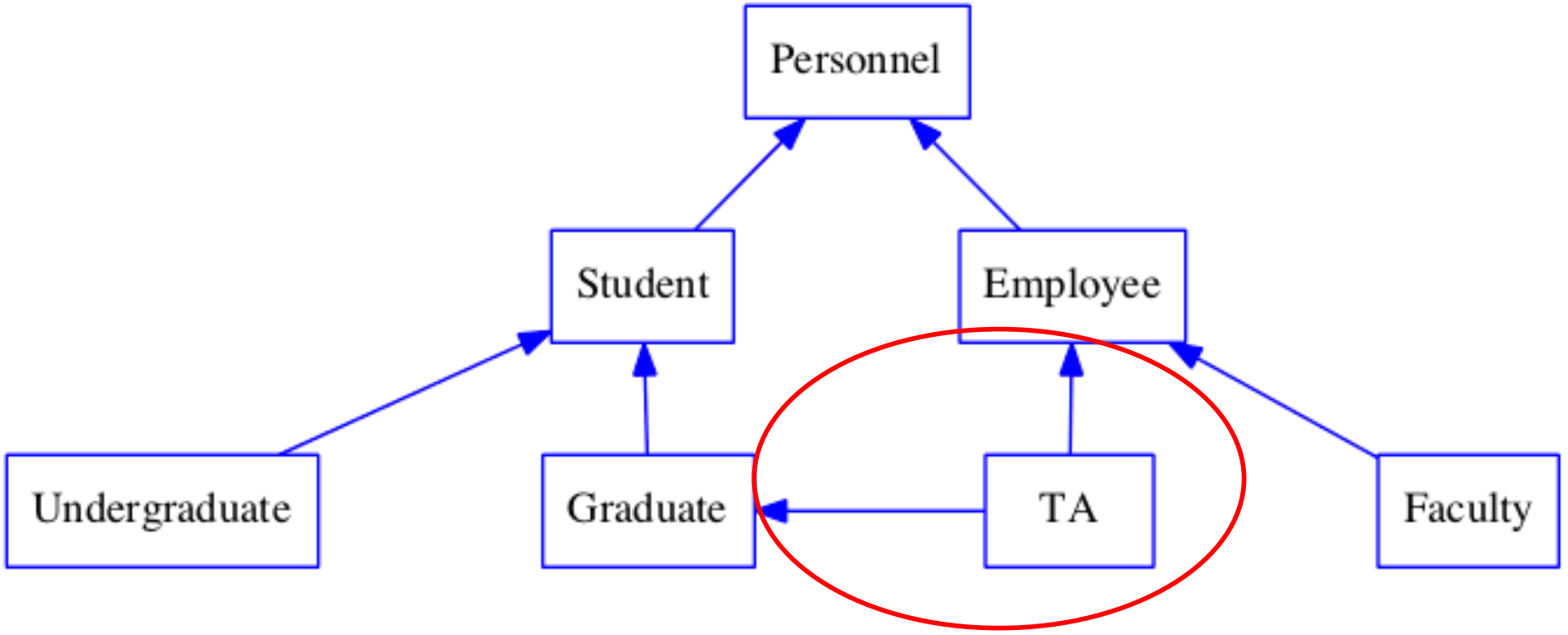
Multilevel Inheritance



Some Examples

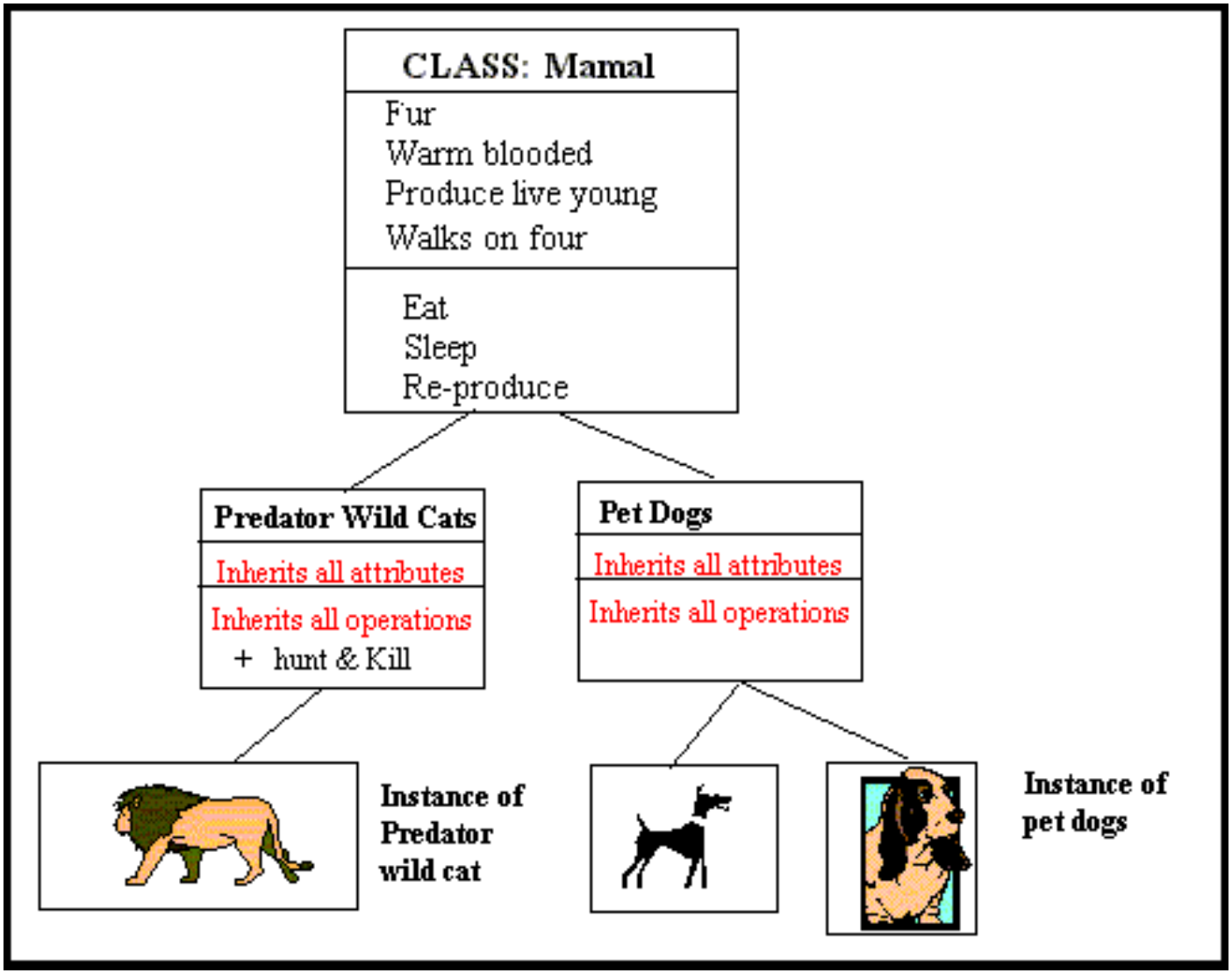


Some Examples

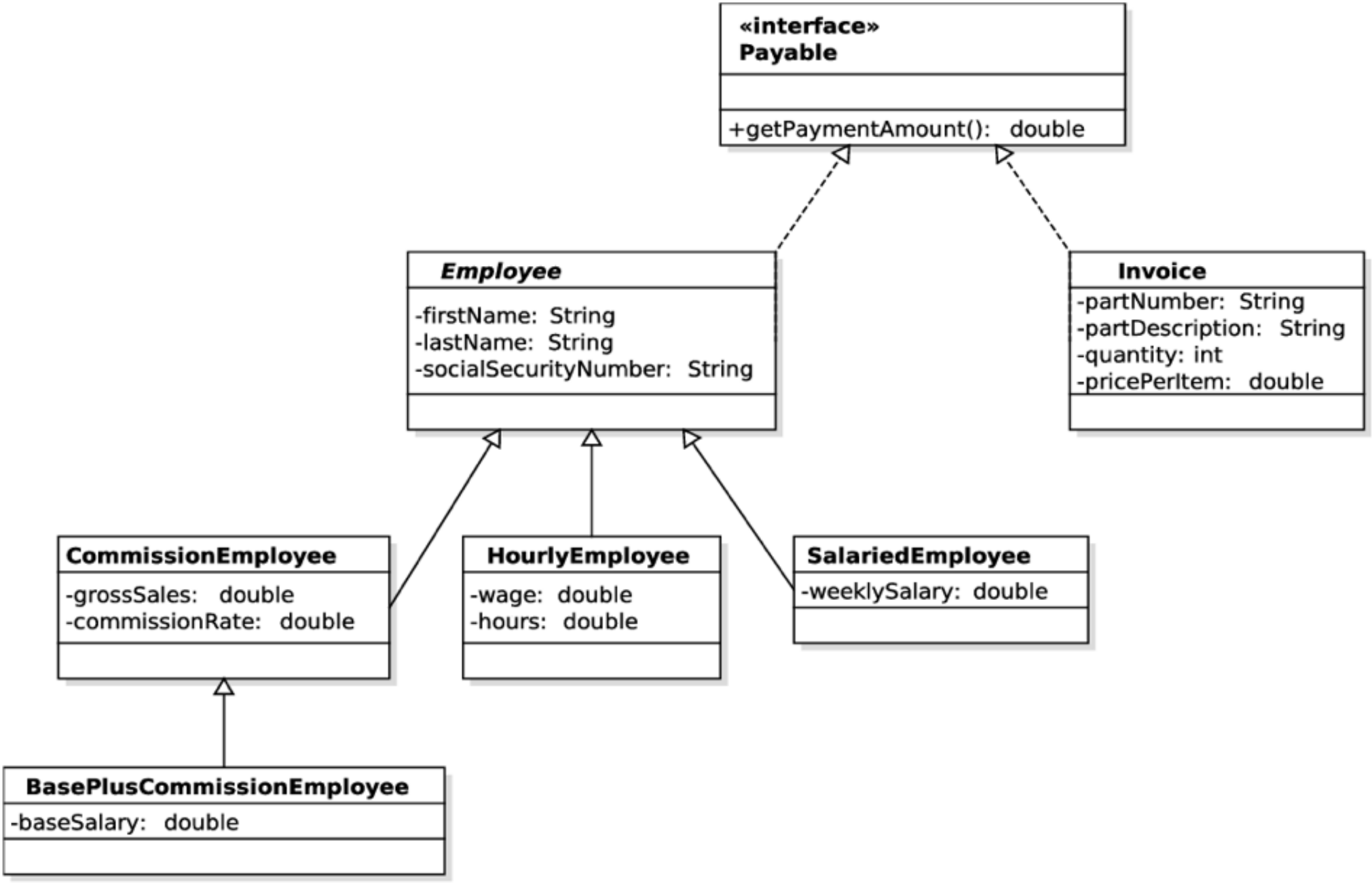


Multiple Inheritance

Some Examples



Some Examples



Inheritance

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

- Example:

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

Inheritance

```
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: **class.method(object, parameters)**
- constructors: **class.__init__(parameters)**

Calling Superclass Methods

- methods: **class.method(object, parameters)**
- constructors: **class.__init__(parameters)**

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

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

Polymorphism

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

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)`

Generating Exceptions

`raise` **ExceptionType**`("`**message**`")`

- useful when the client uses your object improperly
- types: `ArithmeticError`, `AssertionError`, `IndexError`, `NameError`, `SyntaxError`, `TypeError`, `ValueError`

Generating Exceptions

`raise` **ExceptionType**`("message")`

- useful when the client uses your object improperly
- types: `ArithmeticError`, `AssertionError`, `IndexError`, `NameError`, `SyntaxError`, `TypeError`, `ValueError`
- Example:

```
class BankAccount:
    ...
    def deposit(self, amount):
        if amount < 0:
            raise ValueError("negative amount")
        ...
```

Exception Handling

When an exception occurs, Python will normally stop and generate an error message. These exceptions can be handled using the **try** statement.

Exception Handling

```
try:
  print(x)
except:
  print("An exception occurred")
```

The **try** block will generate an exception, because x is not defined.

Since the try block **raises** an error, the **except** block will be executed. Without the try block, the program will crash and raise an error.

Catching Exceptions

```
Class Example():
    def foo(self, x):
        return 1/x

    def bar(self, x):
        try:
            print self.foo(x)
        except ZeroDivisionError, message:
            print ("Can't divide by zero: %s" % message)

if __name__ == "__main__":
    test = Example()
    test.bar(0)
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error.

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error.

Print one message if the try block raises a NameError and another message for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Exception Handling

You can use the **else** keyword to define a block of code to be executed if no errors were raised.

Exception Handling

You can use the **else** keyword to define a block of code to be executed if no errors were raised.

In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Exception Handling

The **finally** block, if specified, will be executed regardless if the try block raises an error or not.

This can be useful to close objects and clean up resources.

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Exception Handling

Try to open and write to a file that is not writable:

```
try:
   f = open("demofile.txt")
   f.write("Lorum Ipsum")
except:
   print("Something went wrong when
writing to the file")
finally:
   f.close()
```

The program can continue, without leaving the file object open.

File Objects

- `f = open(`*`filename`*`[, `*`mode`*`[, `*`buffersize`*`])`
  - mode can be "r", "w", "a" (like C stdio); default "r"
  - append "b" for text translation mode
  - append "+" for read/write open
  - buffersize: 0=unbuffered; 1=line-buffered; buffered
- methods:
  - `read([`*`nbytes`*`]), readline(), readlines()`
  - `write(`*`string`*`), writelines(`*`list`*`)`
  - `seek(`*`pos`*`[, `*`how`*`]), tell()`
  - `flush(), close()`
  - `fileno()`

Modules

- The highest level structure of Python
- Each file with the py suffix is a module
- Each module has its own namespace

Modules

- Collection of stuff in *foo*.py file
  - functions, classes, variables
- Importing modules:
  - import re; print re.match("[a-z]+", s)
  - from re import match; print match("[a-z]+", s)
- Import with rename:
  - import re as regex
  - from re import match as m
  - Before Python 2.0:
    - import re; regex = re; del re

Modules: Imports

import mymodule	Brings all elements of mymodule in, but must refer to as mymodule.<elem>
from mymodule import x	Imports x from mymodule right into this namespace
from mymodule import *	Imports all elements of mymodule into this namespace

References

- https://www.python-course.eu/python3_object_oriented_programming.php
- Fowler, M. “UML Distilled: A Brief Guide to the Standard Object Modeling Language - 3rded.”, Addison-Wesley Professional (2003)