ECE-122

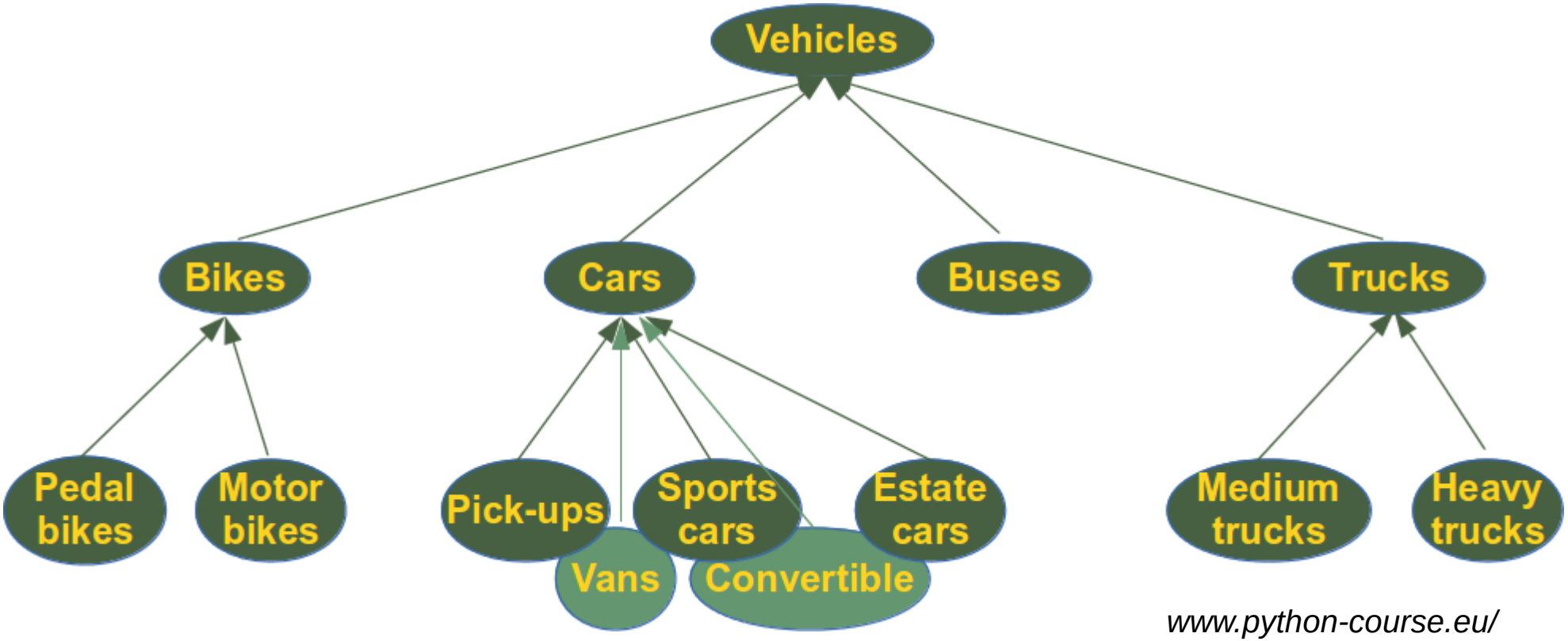
Chapter-3:
Object Oriented Programming and Applications

3.7 Inheritance

Inheritance: Overview

- A key element in OOP
- Inheritance allows a software developer to derive a new class from an existing one
  - The existing class is called the **parent** class, or **superclass**, or base class
  - The derived class is called the **child** class or **subclass**
- As the name implies, the child inherits characteristics of the parent
  - It means that the child class inherits the methods and data defined by the parent class
  - Syntax definition for the child class is `class MyChildClass(MyParentClass):`
- Main advantages:
  - We can avoid duplication of operations
  - Inheritance improves code reuse and general code portability

Concept- inheritance and hierarchy



www.python-course.eu/

Remark: This is an example of single inheritance. A child class can have only one parent class

Inherit methods

```
class Parent:    # define Parent class
   def __init__(self):
      print("Calling parent constructor")

   def parent_method(self):
      print("Calling parent method")
```

```
p=Parent()
p.parent_method()
```

```
Calling parent constructor
Calling parent method
```

```
class Child(Parent):   # define Child class
   def __init__(self):
      print("Calling child constructor")

   def child_method(self):
      print("Calling child method")
```

```
c=Child()
c.child_method()
c.parent_method()
```

```
Calling child constructor
Calling child method
Calling parent method
```

Remark: Constructors are not inherited

Inherit class attributes

```
class Parent:    # define Parent class
   family_name="CodeWell"
   def __init__(self):
      print("Calling parent constructor")

   def parent_method(self):
      print("Calling parent method")
```

```
p=Parent()
print("Parent family name:",Parent.family_name)
```

```
Calling parent constructor
Parent family name: CodeWell
```

```
class Child(Parent):   # define Child class
   def __init__(self):
      print("Calling child constructor")

   def child_method(self):
      print("Calling child method")
```

```
c=Child()
print("Child family name:",Child.family_name)
```

```
Calling child constructor
Child family name: CodeWell
```

Inherit instance attributes

- Same attributes

```
class Parent:    # define Parent class
   def __init__(self,family_name):
      self.family_name=family_name
      print("Calling parent constructor")

   def parent_method(self):
      print("Calling parent method")
```

```
p=Parent("CodeWell")
print("Parent family name:",p.family_name)
```

```
Calling parent constructor
Parent family name: CodeWell
```

```
class Child(Parent):   # define Child class
   def __init__(self,family_name):
      Parent.__init__(self,family_name)
      print("Calling child constructor")

   def child_method(self):
      print("Calling child method")
```

```
c=Child("CodeWell")
print("Child family name:",c.family_name)
```

```
Calling parent constructor
Calling child constructor
Child family name: CodeWell
```

Call the Parent !!

Remark: use the parent's constructor to set up the "parent's part" of the object

Inherit instance attributes

- Extended attributes (additional arguments)

```
class Parent:    # define Parent class
   def __init__(self,family_name):
      self.family_name=family_name
      print("Calling parent constructor")

   def parent_method(self):
      print("Calling parent method")
```

```
p=Parent("CodeWell")
print("Parent family name:",p.family_name)
```

```
Calling parent constructor
Parent family name: CodeWell
```

```
class Child(Parent):   # define Child class
   def __init__(self,family,first):
      Parent.__init__(self,family)
      self.first=first
      print("Calling child constructor")

   def child_method(self):
      print("Calling child method")
```

```
c=Child("CodeWell","Lucy")
print("Child name:",c.first,c.family_name)
```

```
Calling parent constructor
Calling child constructor
Child name: Lucy CodeWell
```

Call the Parent constructor to assign attribute family_name, and then assign new attribute first

Inherit instance attributes

- Summary:
  - Constructor in Parent class

```
def __init__(self,arg1,arg2,arg3):
    self.arg1=arg1
    self.arg2=arg2
    self.arg3=arg3
```

  - Constructor in Child class (two approaches)

```
def __init__(self,arg1,arg2,arg3,arg4):
    Parent.__init__(self,arg1,arg2,arg3)
    self.arg4=arg4
```

```
def __init__(self,arg1,arg2,arg3,arg4):
    super().__init__(arg1,arg2,arg3)
    self.arg4=arg4
```

  - Here the super reference is used to refer to the parent class
- Private attributes defined in Parent class, cannot be explicitly accessed in the Child class.
  - Use getter-setter methods (since methods are inherited)
  - Use protected attributed (only one underscore in front of name)

Super-Methods

- A method in the parent class can be invoked explicitly in the child class using the super reference

```
class Parent:    # define Parent class
   def __init__(self):
      print("Calling parent constructor")

   def parent_method(self):
      print("Calling parent method")
```

```
class Child(Parent):   # define Child class
   def __init__(self):
      print("Calling child constructor")

   def child_method(self):
      print("Calling child method and",end="\n\t")
      super().parent_method()
      # alternatively, this is good too
      # Parent.parent_method(self)
```

```
p=Parent()
c=Child()
p.parent_method()
c.child_method()
```

```
Calling parent constructor
Calling child constructor
Calling parent method
Calling child method and
        Calling parent method
```

Overriding- definition

- A child class can override the definition of an inherited method in favor of its own. A subclass can provide a different implementation of a same name method that is already defined by its Parent class.
- Method is said to be overridden if parameters, and return type are the same

```
class Parent:    # define Parent class
   def __init__(self):
      print("Calling parent constructor")

   def my_method(self):
      print("Calling parent method")
```

```
class Child(Parent):   # define Child class
   def __init__(self):
      print("Calling child constructor")

   def my_method(self):#method is overridden
      print("Calling child method")
```

```
p=Parent()
c=Child()
p.my_method()
c.my_method()
```

```
Calling parent constructor
Calling child constructor
Calling parent method
Calling child method
```

Remark: __init__ is overridden too

Overriding- application

- Overriding methods provides a powerful way to customize classes without rewriting the class. If needed, a super method called in the child class can be overridden in the process.
  For example: overriding __str__

```
class Parent:    # define Parent class
   def __init__(self):
      print("Calling parent constructor")
   def __str__(self):
      return "Parent method"
```

```
class Child(Parent):   # define Child class
   def __init__(self):
      print("Calling child constructor")
   def __str__(self):#method is overridden
      return super().__str__()+"overridden by Child method"
      # alternatively, this is good too
      # return Parent.__str__(self)+" overridden by Child method"
```

```
p=Parent()
c=Child()
print(p)
print(c)
```

```
Calling parent constructor
Calling child constructor
Parent method
Parent method overridden by Child method
```

Multiple inheritance

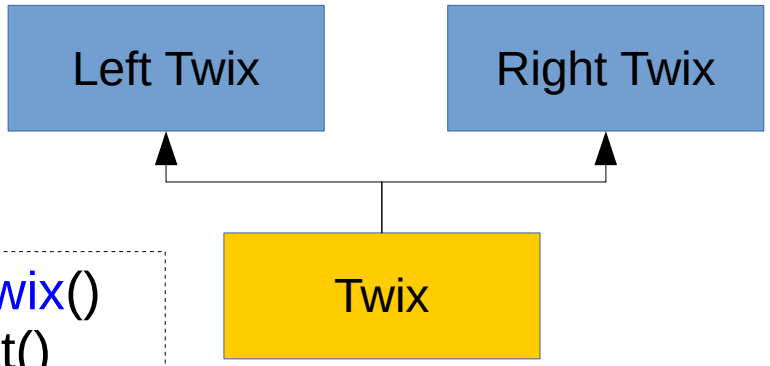
- Multiple inheritance allows a class to be derived from two or more classes:
  - Classes inherit the members of all parents
  - Java does not support it (as many other OO languages), but Python does
  - Collisions (same method name in two parents), have to be resolved.

`class Child(Parent1,Parent2,etc.):`

```
class LeftTwix():   # define first-parent of twix
   def __init__(self): print("Enter left twix")
   def eat(self): print("Eating left twix")

class RightTwix():   # define second-parent of twix
   def __init__(self): print("Enter right twix")
   def eat(self): print("Eating right twix")

class Twix(LeftTwix,RightTwix):   # define Child
   def __init__(self): print("Enter twix")
   def eat(self): print("Eating whole twix")
```




```
t=Twix()
t.eat()
```

```
Enter twix
Eating whole twix
```

```
class Twix(LeftTwix,RightTwix):   # define Child
   def __init__(self): print("Enter twix")
```

```
Enter twix
Eating left twix
```

Remark: follows the order of inheritance declaration (left,right)

Multiple inheritance

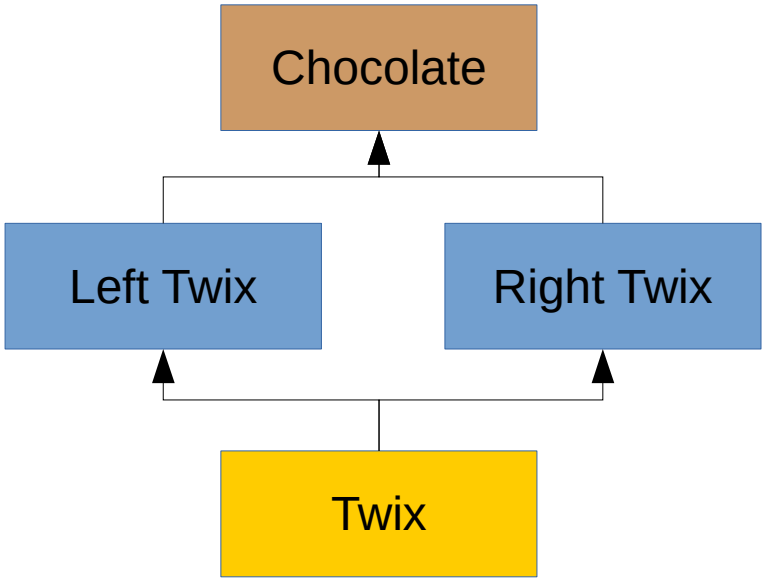
- Super operator used with the diamond situation (Resolution order example)

```
class Chocolate():   # define grand-parent of twix
   def __init__(self): print("Enter chocolate")

#define first-parent of twix and child of chocolate
class LeftTwix(Chocolate):
   def __init__(self):
      print("Enter left twix")
      super().__init__()

#define second-parent of twix and child of chocolate
class RightTwix(Chocolate):
   def __init__(self):
      print("Enter right twix")
      super().__init__()

class Twix(LeftTwix,RightTwix):   # define Child
   def __init__(self):
      print("Enter twix")
      super().__init__()
```




```
t=Twix()
```

```
Enter twix
Enter left twix
Enter right twix
Enter chocolate
```