Objects composition

- There are usually two main methods for reusing an already existent code: ***object composition*** and ***class inheritance***.

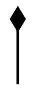
- In the ***object composition*** case, an object belonging to a certain compound class contains one or more objects belonging to other classes. In this case the relation used between classes is a *membership* relation.

- In the case of ***class inheritance*** the inheritance relation is in fact a *refining* relation: a derived class inherits a base class, and it adds to the base class some more specific information.

A. Defining compound classes and using composite objects

- The ***composition*** is a kind of a more general relation - *class association*, where objects of the compound class must manage the life cycle of their component objects, like object creation and their destruction.

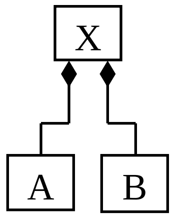
- In the UML language the composition relation is graphical represented as:



Where the diamond is related to the compound class.

Example:

```
class A;
class B;
class X {
  A *a;
  B *b;
  // ...
};
```



- The objects that are data members in a compound class can be *private* (or *protected*) or *public* members.
- The object components from a compound class are, usually, private (or protected), because in the compound class is used the functionality of the component classes, and not their interfaces.
- When the objects members of a compound class are ***public members*** they can be direct referred by the selection operator:

```
class A {
public:
  int n;
  A();
  // ...
};
class B {
public:
  double z;
  void f();
  B();
  // ...
};
class X {
  char c;
public:
  A a;
  B b;
  X();
  // ...
};
void Processing ()
{
  X x;
  x.a.n = 3;
  x.b.z = 4.5;
  x.b.f();
};
```

- When the component objects are ***private members*** into a compound class these members cannot be direct accessed by using the selection operator. Moreover, the compound class must contain its own functions for accessing the members of the component objects.

Example:

```
class A {
  int n;
public:
  int N() const
   { return n; }
  void g() {
    cout << ″g function in
      class A″ << endl;
  }
  A(int k = 0) { n = k; }
  // ...
};

class B {
  double z;
public:
  double Z() const
   { return z; }
  void f() {
    cout << ″f function in
      class B″ << endl;
  }
  void g() {
    cout << ″g function in
      class B″ << endl;
  }
  B(double k = 0) { z = k;
}
  // ...
};
```

```
class X {
  char c;
  A a;
  B b;
public:
  void f() { b.f(); }
  void ga() { a.g(); }
  void gb() { b.g(); }
  X();
  // ...
};
```

```
void Processing () {
  X x;
  x.f();
  x.ga();
  x.gb();
  // ...
}
```

- ***Object composition*** in ***Python*** is similar to the same relation of C++
  - It represents a "*Has-a*" relation

- However, there is a subtle difference, because of object reference mechanism used in Python:
  - Python classes can easily manage both ***composition*** and ***aggregation*** relations

- A simple example of *composition*, where an instance of a class is created *inside* the constructor of another class:

```
class A:
   def __init__(self, x_):
      self.x = x_
class B:
   def __init__(self, x_):
      self.a = A(x_)
```

- A simple example of *aggregation*, where the constructor of a class receives an *instance* (already created) of another class:

```
class A:
   def __init__(self, x_):
      self.x = x_
class B:
   def __init__(self, a_):
      self.a = a_
a = A(7)
b = B(a)
```

- Because Python functions can collect arbitrarily many positional arguments using the **`*varargs`** syntax, a constructor can use this property for constructing a list of arbitrary objects already created (a particular case of aggregation of object into an container)

```
class A:
   def __init__(self, x_):
      self.x = x_
class B:
   def __init__(self, y_):
      self.y = y_
class Container:
   def __init__(self, *args):
      self.members = list(args)
   def addElem(self, elem):
      self.members.append(elem)
a = A(5)
b = B(7)
c = Container(a, b)
a1 = A(9)
c.addElem(a1)
```

- *Object composition* is one of the two variants used in OOP paradigm that allows *code reuse*
  - One can use already defined classes for extending their functionalities
- Example:

```
class Aditive:
    def __init__(self, x_, y_):
        self.x = x_
        self.y = y_
    def add(self):
        return self.x + self.y
    def sub(self):
        return self.x - self.y
class Multiplicative:
    def __init__(self, x_, y_):
        self.x = x_
        self.y = y_
    def mul(self):
        return self.x * self.y
    def div(self):
        return self.x / self.y
```

```
class Algebraic:
  def __init__(self, x_, y_):
      self.x = x_
      self.y = y_
      self.m1 = Aditive(x_, y_)
      self.m2 = Multiplicative (x_, y_)
   def pow(self):
      return self.x ** self.y
   def add(self):
      return self.m1.add()
   def sub(self):
      return self.m1.sub()
   def mul(self):
      return self.m2.mul()
   def div(self):
      return self.m2.div()

   alg = Algebraic(3,2)
   r1 = alg.add()    # >>> 3
   r2 = alg.sub()    # >>> 1
   r3 = alg.mul()    # >>> 9
   r4 = alg.pow()    # >>> 9
```

B. Creation and destruction of composite objects. The constructor initializer list

- If a class contains objects of other classes as data members, the constructors and the destructor of the compound class must manages the calling of the appropriate constructors and the destructors of the component classes, in order to:
  - Initialize the sub-object of the composite object,
  - Destroy these sub-objects respectively.

- The C++ language allows a proper initialization of the sub-objects of a composite object by using a ***constructor initializer list***. A constructor initializer list is specified between the header of the constructor and its body.

- Each element from the constructor initialization list is associated to a class member and it is specified by its member name and arguments that will be passed to its constructor.

- For example, for the class *X* from the previous example, the class constructor can be described as:

```
X::X(): a(5), b(5.3), c(‘x’)
{
  cout << ”constructor of the class X”;
}
```

- For every element from the constructor initializer list, it is called a constructor of the class that the respective sub-object belongs to.

- In the constructor initializer list can appear, by extension, elements for those members which belong to a predefined data type. This extension allows a consistent syntax, which treats an initializer variable having a predefined type as a ***pseudo-constructor***.

- The ***pseudo-constructor*** notion is used in the C++ language also for initialization of variables belonging to a predefined type outside a certain class, as in the following sequence:

```
int n(3);
int* p = new int(7);
```

- A good programming style imposes as the constructor initialization list to have the number of elements equal to the number of data members of the compound class, ensuring a safe initialization of the data members before the execution of the statements from the body of the constructor function.

- There are cases when the explicit call of the constructors is not necessary in the constructor initializer list. For example, if a sub-object has a default constructor, or a constructor with default values for parameters, and it is desired its call (and not the call of another constructor), the corresponding initializer element from the initialization list can be omitted.

- The order of calling the constructors for the initialization of the members of a composite object *is not necessary* the order in which the members appear in the constructor initializer list. The (pseudo-)constructors are called always *in the order in which the members appear in the class declaration*.

- The order of calling of destructors is always reverse of the order of calling of constructors:

1) The destructors for sub-objects are called after the destructor call of the composite object. If a sub-object is also composed from other objects, this process is recursively performed.
2) The destructors of the sub-objects are called in the reverse order of the declaration of these sub-objects in the compound class.

Example:

```
#include <iostream>
using namespace std;
class A1 {
  int p;
public:
  A1(int k = 0) {
    p = k;
    cout << ″A1 class constructor″ << endl;
  }
  ~A1() {cout<<″A1 class destructor″<<endl;}
};
```

```
class A2 {
  int q;
public:
  A2(int k = 0) {
    q = k;
    cout << ″A2 class constructor″ << endl;
  }
  ~A2() {cout<<″A2 class destructor″<<endl;}
 };

class A {
  A1 a1;
  A2 a2;
public:
  A(int i = 0, int j = 0): a1(i), a2(j) {
    cout << ″A class constructor″ << endl;
  }
  ~A() {cout<<″A class destructor″<<endl;}
 };
```

```
class B {
  double z;
public:
  B(double k = 0) {
    z = k;
    cout<< ″B class constructor″ << endl;
  }
  ~B() {cout<<″B class destructor″<<endl;}
 };

class X {
  A a;
  B b;
public:
  X(int i=0, int j=0, double k=0): a(i, j), b(k)
  {
    cout << ″X class constructor″ << endl;
  }
  ~X() {cout<<″X class destructor″<<endl;}
 };
```

```
int main() {
  cout << ″main function starts″ << endl;
  {
    X x(7,9,3.33);
    cout << ″x object was created″ << endl;
  }
  cout << ″main function ends″ << endl;
  return 0;
}
```

The program output is the following:

```
main function starts                 X class destructor
A1 class constructor                 B class destructor
A2 class constructor                 A class destructor
A class constructor                  A2 class destructor
B class constructor                  A1 class destructor
X class constructor                  main function ends
x object was created
```

- Class constructors in ***Python*** do not have *initializer lists*