# OOP_Encapsulation

October 28, 2018

```
In [1]: #include <iostream>

        using namespace std;
```

# 1 Object Oriented Programming (OOP): Encapsulation

**FE 522 C++ Programming in Finance**

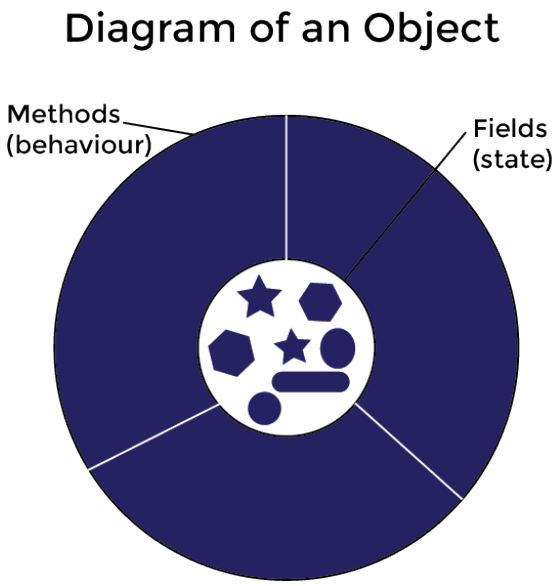
**Thiago W. Alves**

## 1.1 Objects

- Object oriented programming (OOP):
  - Programming paradigm based on the idea of defining our own data types with *classes*, where the data types are particular to the domain of the problem being solved.
  - It's the main paradigm of C++ programming, and part of what triggered the necessity to expand C.
- Objects have two main characteristics:
  - **State:** represented by the member variables (built-in or user-defined types).
  - **Behavior:** exposed with member methods (functions).

## 1.2 Ideal

- Our ideal of program design is to represent the concepts of the application domain directly in code:
  - If you understand the application domain, you understand the code, and vice versa.
- By having a state, and proving methods that can change this state, an object controls what other parts of a program can do with it:
  - e.g. if a bike has 6 gears, the method that shifts gears can reject any value lower than 1 or higher than 6.




Objects.png

## 1.3 OOP

- Methods operate over the state of an object, and serve as the main mechanism of communication between objects.
- **Encapsulation:** hiding the state of an object (with *private* or *protected*) and requiring all interactions to be made with methods is known as **data encapsulation**, one of the three main mechanisms of OOP.

### 1.3.1 Benefits

- **Modularity:** the definition of a class may be written and maintened completely independent from the definition of other classes.
- **Data hiding:** by interacting only through methods, implementation details stay hidden from the outside world (other parts of the same program and/or other programs).
- **Code reuse:** complex and specialized classes that have been previously implemented and tested can be reutilized in several programs.
- **"Easy to delete, easy to debug":** if a specific class becomes problematic, it is easy to remove it and substitute it with something better. (https://programmingisterrible.com/post/173883533613/code-to-debug)

## 1.4 Classes & Objects

- Classes are definined with the keyword *class*:
  - A class is a user-defined data type.
- Once defined, objects of the class may be created:

– Even though we have been using the term *object* to describe any reserved space in memory for a variable (a named object), the term *object* is commonly used in the literature to indicate an instance of a class.

### 1.4.1 Definition

- Defining a class:

```
In [2]: enum OptionType {
            CALL,
            PUT
        };
```

```
In [3]: class EuropeanOption {
        public:
            OptionType m_optionType; // option type
            double m_s;              // spot price
            double m_k;              // strike price
            double m_r;              // interest rate
            double m_t;              // time to maturity
            double m_vol;            // volatility
            double m_price;
        };
```

### 1.4.2 Creating Objects

- To create an object, as with any other variable, simply state the name of a class followed by the name that you wish to give to the object:

```
In [4]: EuropeanOption option;
```

- To access public members of a class, use the dot (.) operator:

```
In [5]: option.m_s = 100.00;
```

```
In [6]: cout << option.m_s << '\n';
```

```
100
```

### 1.4.3 Member Visibility

- **Access control (visibility) of the members of a class:** the *public* keyword is used to indicate that the members that follow are public (visible to anyone).
- Specifying members as *public* implicates that these members can be accessed directly from any function that has access to the object.
- Members of a class may be defined as *public*, *private*, or *protected*.

- If no access control qualifier is specified, members are private by default in classes, and public by default in structs.
- Private members are accessible from within a class definition, but not from outside.
- Later (when we discuss **inheritance**), we will also see the use of *protected*.

### 1.4.4 Public Members

- Public data members are not considered a good practice:
  - It is better to define public methods that access such data members, giving us more control on which modifications should be allowed to the state of an object.
  - The use of private data members also allow us to modify the internal representation of the state of an object without modifying the interface to this object (its public member functions).
- Defining a completely private version of the same class:

```
In [7]: class PrivateEuropeanOption {
            OptionType m_optionType; // option type
            double m_s;              // spot price
            double m_k;              // strike price
            double m_r;              // interest rate
            double m_t;              // time to maturity
            double m_vol;            // volatility
            double m_price;
        };
```

```
In [8]: PrivateEuropeanOption privateOption;
```

- And trying to access its members with the dot (.) operator will give you an error:

```
In [9]: privateOption.m_s = 100.00;
```

```
input_line_16:2:16: error: 'm_s' is a private member of 'PrivateEuropeanOption'
 privateOption.m_s = 100.00;
               ^
input_line_14:3:12: note: implicitly declared private here
    double m_s;              // spot price
           ^



        Interpreter Error:
```

### 1.4.5 Methods

- "Method" is the name used to reference any member function defined in a class, which can modify the state of an object (the values of its data members).
- Methods have access to all members of a class (data members and other methods), including *private* ones.
- It is also possible to configure the access control of methods with the keywords *public*, *private*, and *protected*.

```
In [10]: class EuropeanOption2 {
         public:
             OptionType m_optionType; // option type
             double m_s;              // spot price
             double m_k;              // strike price
             double m_r;              // interest rate
             double m_t;              // time to maturity
             double m_vol;            // volatility
             double m_price;

         public:
             bool computePrice();

         private:
             double normalCDF(double x);
         };
```

### 1.4.6 Interface vs. Implemetantion

- Separating the interface from the implementation: it is considered one of the good practices of software engineering, since it facilitates the process of modifying our programs without breaking other pieces of software.
- In C++, a class should be declared in a .h file (its interface) and defined in a .cpp file (its implementation):
  - The interface of a class contains its declaration, with its data members and method prototypes.
  - In the implementation file, each method must be reference the class it is part of, using the score resolution (::) operator.

```
In [11]: /*
         Compute the price of a plain vanilla european option (CALL or PUT)
         */
         bool EuropeanOption2::computePrice() {
             // parameters validation
             if(m_s <= 0.0 || m_k <= 0.0 || m_t <= 0.0 || m_vol <= 0.0)
                 return false;
```

```
        // d1
        double d1 = (log(m_s / m_k) + (m_r + m_vol * m_vol / 2.0) * m_t);
        d1 /= (m_vol * sqrt(m_t));

        // d2
        double d2 = d1 - m_vol * sqrt(m_t);

        // price computation
        if(m_optionType == CALL)
            m_price = m_s * normalCDF(d1) - m_k * exp(-m_r * m_t) * normalCDF(d2);
        else
            m_price = m_k * exp(-m_r * m_t) * normalCDF(-d2) - m_s * normalCDF(-d1);

        return true;
    }
```

- Methods do not need to be specified in any particular order:

```
In [12]: double EuropeanOption2::normalCDF(double x)
         {
             return std::erfc(-x / std::sqrt(2.0)) / 2.0;
         }
```

- And access control modifiers need only to be specified in the interface (in the .h file).

### 1.4.7 Method Calls

- Methods may only be called when associated with a particular object with the dot (.) operator:
  - A method operates over a particular object and not over the class as a whole.

```
In [13]: EuropeanOption2 option2;
         option2.m_optionType = CALL; // option type
         option2.m_s = 100.0;         // spot price
         option2.m_k = 100.0;         // strike price
         option2.m_r = 0.12;          // interest rate
         option2.m_t = 3.0;           // time to maturity
         option2.m_vol = 0.20;        // volatility

         // price computation
         if(option2.computePrice() == false) {
             cout << "It was not possible to compute the price of the option." << '\n';
         }
         else {
             cout << option2.m_price << '\n';
         }
```

```
32.4468
```

## 1.5 Constructors

- Constructors are used to initialize an object during its *construction* (creation).
- The constructor of a class is a special method that is always called when an object is constructed (created). This method is different than others methods in that:
  - It does not have a return value.
  - Its name is the same of its class'.

```
In [14]: class EuropeanOption3 {
         public:
             OptionType m_optionType; // option type
             double m_s;              // spot price
             double m_k;              // strike price
             double m_r;              // interest rate
             double m_t;              // time to maturity
             double m_vol;            // volatility
             double m_price;

         public:
             EuropeanOption3(); // default constructor (with no parameters)

             bool computePrice();

         private:
             double normalCDF(double x);
         };
```

- The values of the data members of a class can be initialized in the constructor of the class:

```
In [15]: EuropeanOption3::EuropeanOption3() {
             m_optionType = CALL;
             m_s = 100.0;
             m_k = 100.0;
             m_r = 0.12;
             m_t = 3.0;
             m_vol = 0.20;
             m_price = -1.0;      // invalid value
         }
```

- A class may have more than one constructor:
  - A constructor without any arguments is called a *default constructor*.

```
In [16]: class EuropeanOption4 {
         public:
             // default constructor (with no parameters)
             EuropeanOption4();
```

```
        void Initialize();

        // constructor with parameters
        EuropeanOption4(OptionType type, double s, double k, double r, double t, double v

        OptionType m_optionType; // option type
        double m_s;              // spot price
        double m_k;              // strike price
        double m_r;              // interest rate
        double m_t;              // time to maturity
        double m_vol;            // volatility
        double m_price;
};
```

- A constructor may call other methods of the same class.

```
In [17]: EuropeanOption4::EuropeanOption4() {
             Initialize();
         }
```

```
In [18]: void EuropeanOption4::Initialize() {
             m_optionType = CALL;
             m_s = 100.0;
             m_k = 100.0;
             m_r = 0.12;
             m_t = 3.0;
             m_vol = 0.20;
             m_price = -1.0;      // invalid value
         }
```

- A constructor may also, as any other method, contain different arguments (parameters).

```
In [19]: EuropeanOption4::EuropeanOption4(OptionType type, double s, double k, double r, double
             m_optionType = type;
             m_s = s;
             m_k = k;
             m_r = r;
             m_t = t;
             m_vol = vol;
             m_price = -1.0;      // invalid value
         }
```

- If no constructors are defined for a particular class, a default constructor is automatically provided.
- If there are non-default constructors defined, a default constructor is **not** automatically provided.

```
In [20]: class EuropeanOption5 {
         public:
```

```
        // constructor with parameters
        EuropeanOption5(OptionType type, double s, double k, double r, double t, double v

        OptionType m_optionType; // option type
        double m_s;              // spot price
        double m_k;              // strike price
        double m_r;              // interest rate
        double m_t;              // time to maturity
        double m_vol;            // volatility
        double m_price;
    };
```

- And the following becomes invalid:

```
In [21]: EuropeanOption5 option5;

input_line_28:2:18: error: no matching constructor for initialization of 'EuropeanOption5'
 EuropeanOption5 option5;
                 ^
input_line_27:1:7: note: candidate constructor (the implicit copy constructor) not viable: requ
class EuropeanOption5 {
      ^
input_line_27:1:7: note: candidate constructor (the implicit move constructor) not viable: requ
input_line_27:4:5: note: candidate constructor not viable: requires 6 arguments, but 0 were pro
    EuropeanOption5(OptionType type, double s, double k, double r, double t, double vol);
    ^



    Interpreter Error:
```

### 1.6 Setters and Getters

- Private data members of a class may only be accessed by methods of the same class.
- It is a common practice to provide public **set** and **get** methods for clients of the class to use in order to *set* or *get* the current value of a particular private data member.
- It is acknowledged as a better practice than using public data members:
  - a **set** method can monitor any attempt of changing the value of a data member: making a data member private and monitoring all access with a public method helps us to ensure data integrity, and is, therefore, a good programming practice.
  - a **get** method may, for example, properly format a value before returning it.

```
In [22]: class EuropeanOption6 {
         private:
             OptionType m_optionType; // option type
             double m_s;              // spot price
```

```
        double m_k;             // strike price
        double m_r;             // interest rate
        double m_t;             // time to maturity
        double m_vol;           // volatility
        double m_price;

    public:
        // Getters
        OptionType getOptionType();
        double getSpot();
        double getStrike();
        double getR();
        double getT();
        double getVol();
        double getPrice();

        // Setters
        bool setOptionType(OptionType type);
        bool setSpot(double s);
        bool setStrike(double k);
        bool setR(double r);
        bool setT(double t);
        bool setVol(double vol);
        bool setPrice(double price);
    };
```

- *Option Type* getter and setter:

```
In [23]: OptionType EuropeanOption6::getOptionType() {
             return m_optionType;
         }
```

```
In [24]: bool EuropeanOption6::setOptionType(OptionType type) {
             if(type != CALL && type != PUT) {
                 return false;
             }

             m_optionType = type;
             return true;
         }
```

```
In [25]: EuropeanOption6 option6;
```

```
In [26]: option6.setOptionType(PUT);
         cout << ((option6.getOptionType() == 0) ? "CALL" : "PUT") << '\n';
```

```
PUT
```

- *Price* getter and setter:

```
In [27]: double EuropeanOption6::getPrice() {
             return m_price;
         }

In [28]: bool EuropeanOption6::setPrice(double price) {
             if(price < 0.0) {
                 return false;
             }

             m_price = price;
             return true;
         }

In [29]: cout << option6.setPrice(option2.m_price) << '\n';

1


In [30]: cout << option6.setPrice(-100.00) << '\n';

0


In [31]: cout << option6.getPrice() << '\n';

32.4468
```

### 1.7 Constant Objects

- A *const* qualifier in the declaration of an object indicates that it is a constant and that none of its data members can be modified.
- When a constant object is created, the compiler forbids the use of any of its class' methods, since it has no way of distinguishing which methods would actually try to change the current state of the object.
- So if we try to use our previous implementation of getOptionType() with a *const* object, we will be given an error:

```
In [32]: const EuropeanOption6 constOption6{};

In [33]: OptionType type6 = constOption6.getOptionType();

input_line_40:2:21: error: member function 'getOptionType' not viable: 'this' argument has type
 OptionType type6 = constOption6.getOptionType();
                    ^~~~~~~~~~~~
input_line_30:1:29: note: 'getOptionType' declared here
OptionType EuropeanOption6::getOptionType() {
                            ^
```

```
Interpreter Error:
```

- There is, however, a way to inform the compiler that a method does not modify any of the class' data members, and that we should then be able to call it from a constant object.
- All one has to do is to use the keyword *const* after the closing parenthesis of the paremeter list of a method.
- .h file:

```
In [34]: class EuropeanOption7 {
         private:
             OptionType m_optionType; // option type
             double m_s;              // spot price
             double m_k;              // strike price
             double m_r;              // interest rate
             double m_t;              // time to maturity
             double m_vol;            // volatility
             double m_price;

         public:
             // Getters
             OptionType getOptionType() const;
             double getPrice() const;

             // Setters
             bool setOptionType(OptionType type);
             bool setPrice(double price);
         };
```

- .cpp file:

```
In [35]: OptionType EuropeanOption7::getOptionType() const {
             return m_optionType;
         }
```

```
In [36]: double EuropeanOption7::getPrice() const {
             return m_price;
         }
```

- Now we should be able to use our new implementation of getOptionType() with a *const* object:

```
In [37]: const EuropeanOption7 constOption7{};
```

```
In [38]: OptionType type7 = constOption7.getOptionType();
```

```
In [39]: cout << ((type7 == 0) ? "CALL" : "PUT") << '\n';
```

```
CALL
```