Object Oriented Paradigm is a programming approach revolves around **objects** that focuses on modeling real world entities and make it easier to reason about them and using them in the context of a program.

It also helps us define various types of relationships between objects and how to use them together,

but relationship between objects is beyond the scope of this discussion.

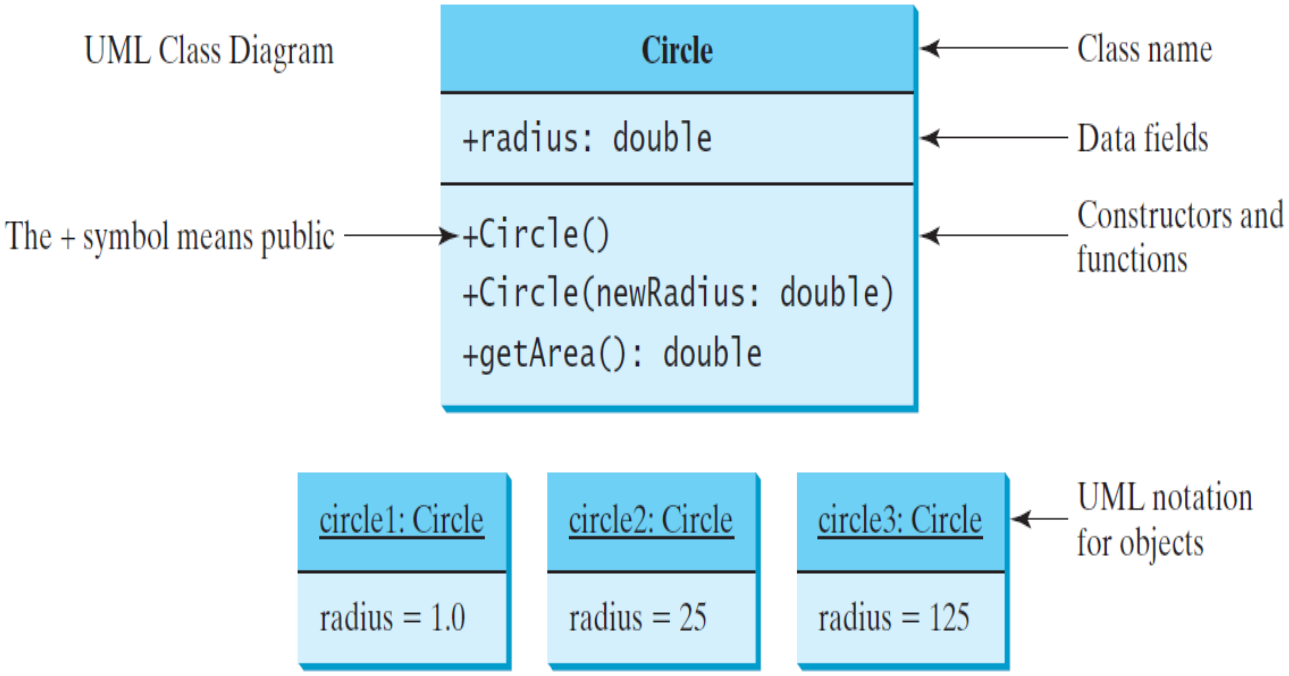
In C++ and many other programming languages Object Oriented relies on two parts:-

- Objects representing real world entities (for example **string** represents text, **ofstream** represents a file).
- Classes/Structs that describes the structure and the behavior of an object and is considered a blueprint to construct objects from.

A Class consists of three primary parts:-

- Class Name
- Data fields: consists of the variables that makes up the an instance of a class.
- Methods: functions that are tied to an instance of a class and **has access to it’s data fields**

We can represent a class in a graphical way with a UML Diagram Like the following



- The first part represents the **name of the Class.**
- The second part represents the **datafileds.**
- The Third part represents the Methods of the class.

Methods that do not have a return type and their name is similar to the name of the class are called **Constructors**.

Constructors are special methods/functions that are used to make objects (instances of particular class) and they are **overloadable**.

They are used to initialize data fields and do any needed computations to make an object.

For example considering this class:-

```
//class name
class Student{

    //data fields
    string name;
    string major;
    double gpa;

    //methods
    Student(string name, string major, double gpa){
        this->name = name;
        this->major = major;
        this->gpa = gpa;
    }

    void say_hi(){
        cout << "hi my name is " << name << endl;
    }
};
```

The function Student is a constructor that is used to make objects of the type Student and initialize it’s data field.

There are three types of constructors we need to be aware of:-

- Normal Constructors: constructors with parameters that initialize the object's data fields and they aren't called automatically for any reason.
- Default Constructors: constructors with no parameters that are called automatically when declaring an object of a class.
- Copy Constructors: constructors that takes a const reference of an object of the same type and are used to copy one object to another object, and they are called automatically when using assignment (=).

For example the following constructors of class **Student**:-

```
//default constructor
Student()
: name(), major(), gpa() {}

//copy constructor
Student(const Student & other){
    this->name = other.name;
    this->major = other.major;
    this->gpa = other.gpa;
}

//normal constructor with parameters
Student(string name, string major, double gpa)
: name(name), major(major), gpa(gpa) {}
```

If we used these constructors as follows, notice which constructor is being called:-

```
    //calling default constructor
    Student s1;
    Student s2();
    Student s3 = Student();

    //calling parameterized normal constructor
    Student s4("Ahmad", "Computer Engineer", 3.52);
    Student s5 = Student("Ahmad", "Computer Engineer"
, 3.52);

    //calling copy constructor
    Student s6 = s4;
    Student s7(s4);
    Student s8 = Student(s4);
```

Copy Constructors are provided by the compiler if no such constructor is written by the programmer.

Default Constructors are also provided by the compiler but only if no other constructors **of any type** present for the class.

Data Fields can be accessed with the following syntax:-

```
objectName.fieldName
```

Methods are functions that are bounded to an instance of a class and have access to it’s fields via **this pointer.**

They can be called by the following syntax:-

```
objectName.methodName(param1, param2, …)
```

for example the following methods for class Student:-

```
void print_info(){
    cout << "My name is: " << name << endl;
    cout << "I study: " << major << endl;
    cout << "my GPA is: " << gpa << endl;
}

int compare_gpa(double other_gpa){
    if (gpa < other_gpa)
        return -1;
    else if(gpa > other_gpa)
        return 1;
    else
        return 0;
}
```

If we execute the following Code:-

```
Student s("Ahmad", "Computer Engineering", 3.52);
s.print_info();
cout << s.compare_gpa(2.89) << endl;
cout << "name: " << s.name << '\n';
```

Output:

My name is: Ahmad

I study: Computer Engineering

my GPA is: 3.52

1

name: Ahmad

Instead of directly accessing fields we usually use getters (accessors) and setters (mutators) to read and write to a field.

For example a getter and a setter for field gpa:-

```
double getGPA(){
    return gpa;
}
void setGPA(double gpa){
    this->gpa = gpa;
}
```

We call them as follows:-

```
s.getGPA(); // read value of GPA
s.setGPA(3.42); //set value of gpa to 3.42
```

Visibility Modifiers are special labels that are used to set the fields and methods of class/struct accessible from outside of it or not:-

There are two types of visibility modifiers:-

- Public: sets the methods and variables to be accessible from outside of the class.
- Private: sets the methods and variables to be inaccessible outside of the class and only accessible from inside of it.

For example:-

```
class Student{

private:
    string name;
    string major;
    double gpa;

public:
    Student(string name, string major, double gpa)
    : name(name), major(major), gpa(gpa) {}

    void print_info(){
        cout << "My name is: " << name << endl;
        cout << "I study: " << major << endl;
        cout << "my GPA is: " << gpa << endl;
    }

    double getGPA(){
        return gpa;
    }
    void setGPA(double gpa){
        this->gpa = gpa;
    }
};
```

The fields of the class Student are private and can’t be accessed outside of the class and all the methods and the constructors are public and are accessible outside of the class

But the field gpa has a getter and a setter and can be accessed using these methods.

This is called Encapsulation

Visibility difference between a class and a struct:-

Class and Struct are very similar to each other with very few differences between them.

The main difference we are interested in is the default visibility their methods and variables which and can be summed as follows:-

- Everything inside a class is **private** by default unless declared otherwise.
- Everything inside a struct is **public** by default unless declared otherwise

Objects can be **const** as their fields can not be changed:-

```
s.name = "Omar"; //this would be illegal
s.setGPA(3.42); //this also is illegal
```

Objects can be constructed and used without being binded to a variable:-

```
//output: 11
cout << string("hello world").length() << '\n';
```

We usually put the **class prototype** in one file denoted **.h** by and the **implementation** in another file denoted by **.cpp** :-

In Student.h :-

```
#ifndef STUDENT_H
#define STUDENT_H
class Student{
private:
    string name;
    string major;
    double gpa;
public:
    void print_info();
}
#endif
```

In Student.cpp:-

```
void Student::print_info(){
    cout << "My name is: " << name << endl;
    cout << "I study: " << major << endl;
    cout << "my GPA is: " << gpa << endl;
}
```