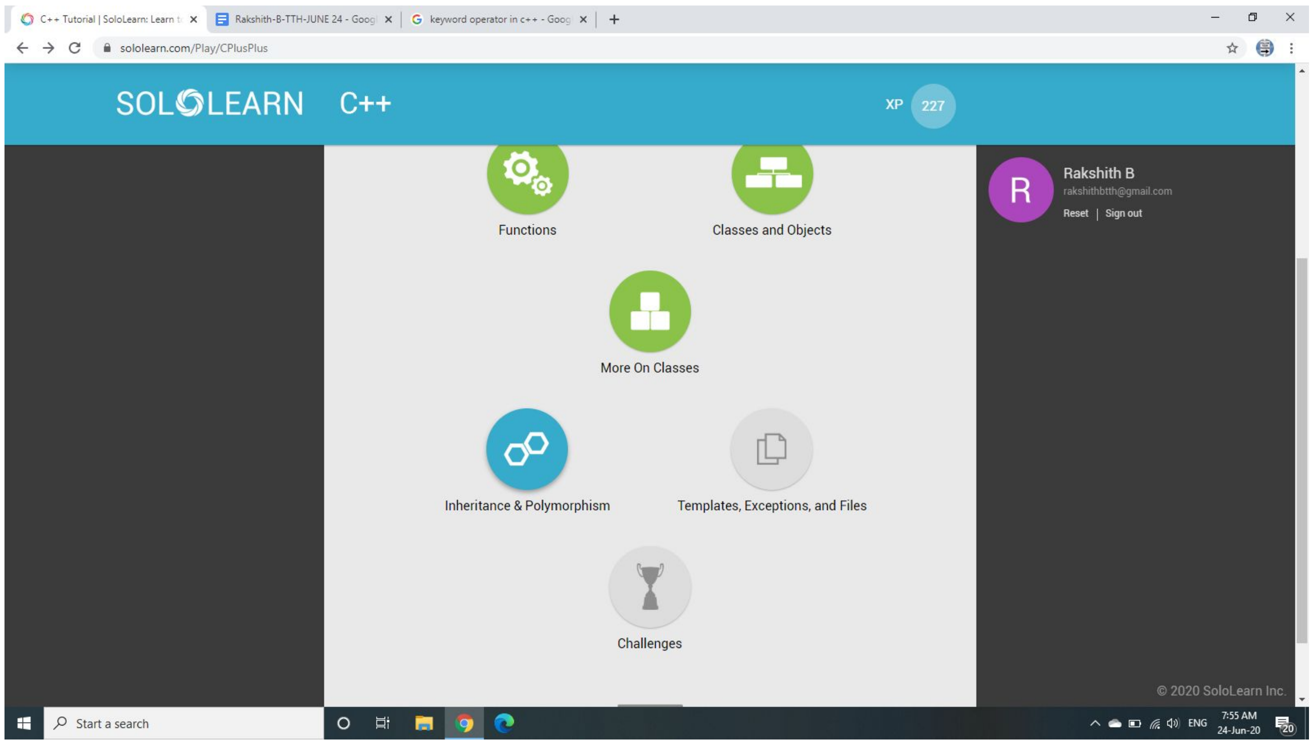
REPORT JUNE 24

Date:	**24 JUNE 2020**	**Name:**	**Rakshith B**
Course:	**Solo Learning**	**USN:**	**4AL16EC409**
Topic:	**Classes and Objects**	**Semester & Section:**	**6th SEM B**
Github Repository:	**Rakshith-B**		

FORENOON SESSION DETAILS
Image of session 

C++ Tutorial | SoloLearn: Learn
Rakshith-B-TTH-JUNE 24 - Goog
keyword operator in c++ - Goog
sololearn.com/Play/CPlusPlus
SOLOLEARN C++
XP 227
Functions
Classes and Objects
More On Classes
Inheritance & Polymorphism
Templates, Exceptions, and Files
Challenges
Rakshith B
rakshithbtth@gmail.com
Reset | Sign out
© 2020 SoloLearn Inc.
Start a search
ENG
7:55 AM
24-Jun-20

Report –

What is an Object

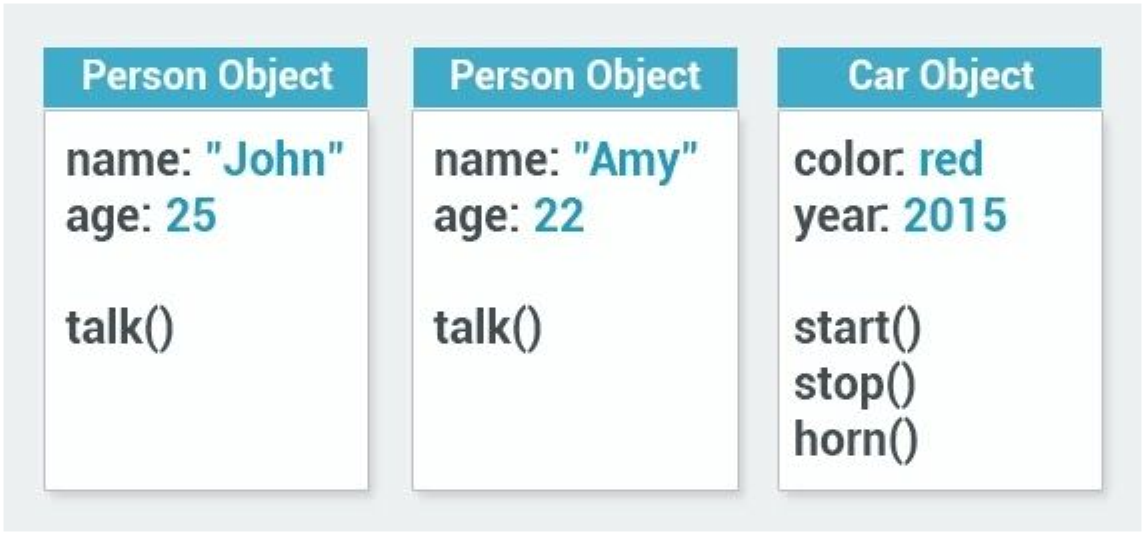
Object **O**riented **P**rogramming is a programming style that is intended to make thinking about programming closer to thinking about the real world.

In programming, **objects** are independent units, and each has its own **identity**, just as objects in the real world do.

Objects

An object might contain other objects but they're still different objects.
Objects also have **characteristics** that are used to describe them. For example, a car can be red or blue, a mug can be full or empty, and so on. These characteristics are also called **attributes**. An attribute describes the current **state** of an object.

Objects can have multiple attributes (the mug can be **empty**, **red** and **large**).
In the real world, each object **behaves** in its own way. The car **moves**, the phone **rings**, and so on.
The same applies to objects - behavior is specific to the object's type.
In programming, an object is **self-contained**, with its own **identity**. It is separate from other objects.
Each object has its own **attributes**, which describe its current state. Each exhibits its own **behavior**, which demonstrates what they can do.



In computing, objects aren't always representative of physical items.
For example, a programming object can represent a date, a time, a bank account. A bank account is not tangible; you can't see it or touch it, but it's still a well-defined object - it has its own **identity**, **attributes**, and **behavior**.

Objects are created using **classes**, which are actually the focal point of OOP.

The class **describes** what the object will be, but is separate from the object itself.
In other words, a class can be described as an object's **blueprint**, description, or definition.
You can use the same class as a blueprint for creating multiple different objects. For example, in preparation to creating a new building, the architect creates a blueprint, which is used as a basis for actually building the structure. That same blueprint can be used to create multiple buildings.

Programming works in the same fashion. We first define a class, which becomes the blueprint for creating objects.

Each class has a **name**, and describes **attributes** and **behavior**.

In programming, the term **type** is used to refer to a class name: We're creating an object of a particular **type**.

Methods

Method is another term for a class' behavior. A method is basically a **function** that belongs to a class.

A Class Example
For example, if we are creating a banking program, we can give our class the following characteristics:
name: BankAccount
attributes: accountNumber, balance, dateOpened
behavior: open(), close(), deposit()

The class specifies that each object should have the defined attributes and behavior. However, it doesn't specify what the actual data is; it only provides a **definition**.

Once we've written the class, we can move on to create objects that are based on that class.
Each object is called an **instance** of a class. The process of creating objects is called **instantiation**.

Declaring a Class
Begin your class definition with the keyword **class**. Follow the keyword with the class name and the class body, enclosed in a set of curly braces.
The following code declares a class called **BankAccount**:

```
class BankAccount {


};
```

Declaring a Class
Define all **attributes** and **behavior** (or members) in the body of the class, within curly braces.
You can also define an **access specifier** for members of the class.
A member that has been defined using the **public** keyword can be accessed from outside the class, as long as it's anywhere within the scope of the class object.

Creating a Class

Let's create a class with one public method, and have it print out "Hi".

```
class BankAccount {
 public:
   void sayHi() {
     cout << "Hi" << endl;
   }
};
```

The next step is to instantiate an object of our **BankAccount** class, in the same way we define variables of a type, the difference being that our object's type will be **BankAccount**.

```
int main()

{

 BankAccount test;

 test.sayHi();

}
```

Encapsulation

Part of the meaning of the word **encapsulation** is the idea of "surrounding" an entity, not just to keep what's inside together, but also to **protect** it.
In object orientation, encapsulation means more than simply combining attributes and behavior together within a class; it also means restricting access to the inner workings of that class.

The key principle here is that an object only reveals what the other application components require to effectively run the application. All else is kept out of view.

For example, if we take our **BankAccount** class, we do not want some other part of our program to reach in and change the **balance** of any object, without going through the **deposit()** or **withdraw()** behaviors.
We should **hide** that attribute, control access to it, so it is accessible only by the object itself.
This way, the **balance** cannot be directly changed from outside of the object and is accessible only using its methods.
This is also known as "**black boxing**", which refers to closing the inner working zones of the object, except of the pieces that we want to make public.
This allows us to change attributes and implementation of methods without altering the overall program. For example, we can come back later and change the data type of the **balance** attribute.

Constructors

Class **constructors** are special member functions of a class. They are executed whenever new objects are created within that class.

The constructor's name is identical to that of the class. It has no return type, not even void.

For example:

```
class myClass {
 public:
  myClass() {
   cout <<"Hey";
  }
  void setName(string x) {
   name = x;
  }
  string getName() {
   return name;
  }
 private:
  string name;
};


int main() {
 myClass myObj;


 return 0;
}


//Outputs "Hey"
```

Constructors can be very useful for setting initial values for certain member variables.

A default constructor has no parameters. However, when needed, parameters can be added to a constructor. This makes it possible to assign an initial value to an object when it's created, as shown in the following example:

```
class myClass {
 public:
  myClass(string nm) {
   setName(nm);
  }
  void setName(string x) {
   name = x;
  }
  string getName() {
   return name;
  }
 private:
  string name;
};
```

When creating an object, you now need to pass the constructor's parameter, as you would when calling a function:

```
class myClass {

 public:

  myClass(string nm) {

   setName(nm);

  }

  void setName(string x) {

   name = x;

  }
```

```
    string getName() {
      return name;
    }
  private:
    string name;
};
int main() {
  myClass ob1("David");
  myClass ob2("Amy");
  cout << ob1.getName();
}
//Outputs "David"
```

Source & Header

The header file (.h) holds the function declarations (prototypes) and variable declarations. It currently includes a template for our new **MyClass** class, with one default constructor.

MyClass.h

```
#ifndef MYCLASS_H
#define MYCLASS_H
class MyClass
{
  public:
    MyClass();
  protected:
  private:
};
#endif // MYCLASS_H
```

The implementation of the class and its methods go into the source file (.cpp).
Currently it includes just an empty constructor.
MyClass.cpp

```
#include "MyClass.h"

MyClass::MyClass()

{

 //ctor

}
```

Scope Resolution Operator

The **double colon** in the source file (.cpp) is called the **scope resolution operator**, and it's used for the constructor definition:

```
#include "MyClass.h"


MyClass::MyClass()

{

 //ctor

}
```

The scope resolution operator is used to define a particular class' member functions, which have already been declared. Remember that we defined the constructor prototype in the **header file**.

Source & Header

To use our classes in our main, we need to include the **header** file.

For example, to use our newly created **MyClass** in main:

```
#include <iostream>

#include "MyClass.h"

using namespace std;

int main() {

 MyClass obj;

}
```

Destructors

Remember constructors? They're special member functions that are automatically called when an object is created.
Destructors are special functions, as well. They're called when an object is **destroyed** or **deleted**.

Destructors

The name of a **destructor** will be exactly the same as the class, only prefixed with a **tilde (~)**. A destructor can't return a value or take any parameters.

```
class MyClass {
 public:
   ~MyClass() {
    // some code
   }
};
```

After declaring the destructor in the header file, we can write the implementation in the source file MyClass.cpp:

```
#include "MyClass.h"
#include <iostream>
using namespace std;
MyClass::MyClass()
{
 cout<<"Constructor"<<endl;
}
MyClass::~MyClass()
{
 cout<<"Destructor"<<endl;
}
```

#ifndef & #define

We created separate header and source files for our class, which resulted in this header file.

```
#ifndef MYCLASS_H
```

```
#define MYCLASS_H
class MyClass
{
 public:
 MyClass();
 protected:
 private:
};
#endif // MYCLASS_H
```

Member Functions

Let's create a sample function called **myPrint()** in our class.

MyClass.h

```
class MyClass
{
 public:
  MyClass();
  void myPrint();
};
```

MyClass.cpp

```
#include "MyClass.h"
#include <iostream>
using namespace std;
MyClass::MyClass() {
}
void MyClass::myPrint() {
 cout <<"Hello"<<endl;
}
```

Constants

A **constant** is an expression with a fixed value. It cannot be changed while the program is running.
Use the **const** keyword to define a constant variable.

```
const int x = 42;
```

Constant Objects

As with the built-in data types, we can make class objects constant by using the **const** keyword.

```
const MyClass obj;
```

All const variables must be initialized when they're created. In the case of classes, this initialization is done via constructors. If a class is not initialized using a parameterized constructor, a public default constructor must be provided - if no public default constructor is provided, a compiler error will occur.

Once a const class object has been initialized via the constructor, you cannot modify the object's member variables. This includes both directly making changes to public member variables and calling member functions that set the value of member variables.

Member Initializers

Recall that **constants** are variables that cannot be changed, and that all const variables must be initialized at time of creation.

C++ provides a handy syntax for initializing members of the class called the **member initializer list** (also called a **constructor initializer**).

Member Initializers

Consider the following class:

```
class MyClass {
 public:
  MyClass(int a, int b) {
   regVar = a;
   constVar = b;
  }
 private:
   int regVar;
   const int constVar;
};
```

This class has two member variables, **regVar** and **constVar**. It also has a constructor that takes two parameters, which are used to initialize the member variables.
Running this code returns an **error**, because one of its member variables is a **constant**, which cannot be assigned a value after declaration.

In cases like this one, a **member initialization list** can be used to assign values to the member variables.

```
class MyClass {
public:
 MyClass(int a, int b)
 : regVar(a), constVar(b)
 {
 }
private:
 int regVar;
 const int constVar;
};
```

Composition

In the real world, complex objects are typically built using smaller, simpler objects. For example, a car is assembled using a metal frame, an engine, tires, and a large number of other parts. This process is called **composition**.

In C++, object composition involves using classes as member variables in other classes.
This sample program demonstrates composition in action. It contains **Person** and **Birthday** classes, and each **Person** will have a **Birthday** object as its member.
Birthday:

This

Every object in C++ has access to its own address through an important pointer called the this pointer.
Inside a member function this may be used to refer to the invoking object.
Let's create a sample class:

```
class MyClass {
public:
 MyClass(int a) : var(a)
 { }
private:
 int var;
};
```