Advanced Programming Practice

Unit-I

Programming Paradigm :-

Paradigm :- Paradigm can also be termed as method to solve some problem or to do some task.

Programming Paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.

"It is a fundamental style of programming that defines how the structure & basic elements of a computer program will be built.

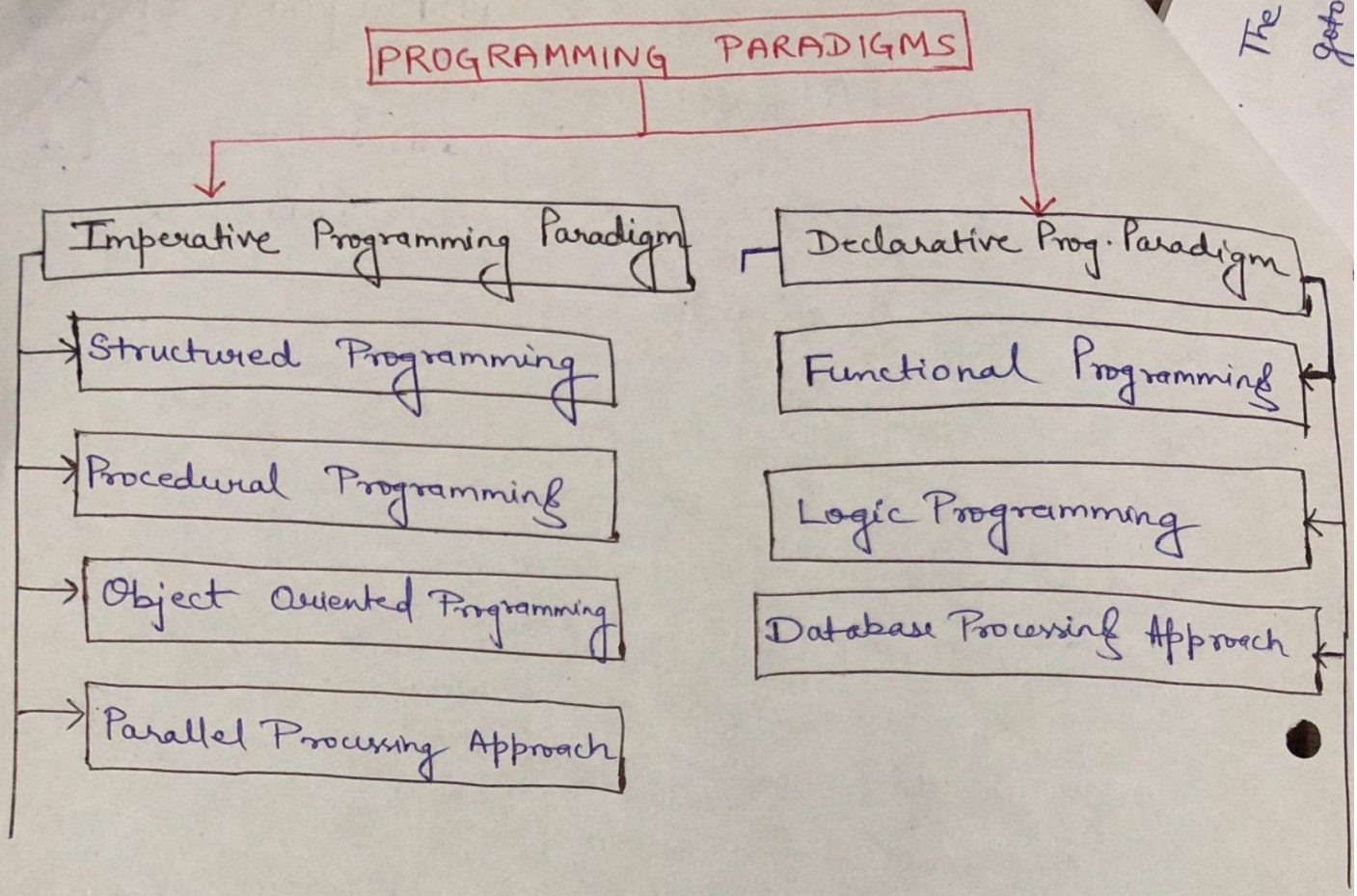
→ The style of writing program & the set of capabilities & limitation that a particular programming language has depends on the programming paradigm it supports.

→ While some programming language strictly follow a single paradigm, other may draw concepts from more than one.

* The are two approaches to programming :-

1) **Imperative programming :-** focuses on how to execute, defines control flow as statements that change a program state.

2) **Declarative programming :-** focuses on what to execute, defines program logic, but not detailed control flow.



* Structured Programming Paradigm:-

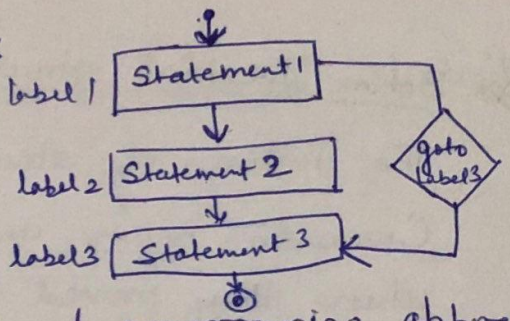
Structured programming Approach, as the word suggests can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any stament like GOTO, etc.

"Structured programming is a paradigm that is based on improving clarity & quality of programs by using subroutines, block structures & loops (for & while) & discouraging use of goto statement."

Goto statement allows unconditional jumping from goto to a labelled statement.

The syntax for this statement is

goto label;
...
lable : statement

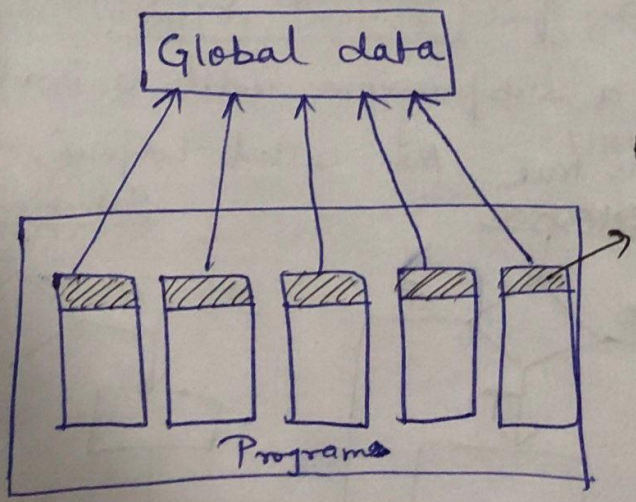


The languages that support structured programming approach are :

a) C
b) C++
c) Java
d) C# etc.
e) Ruby

*Programming Language Theory (PLT):- It is a branch of computer science that deals with the design, implementation, analysis, characterization, & classification of programming languages & their individual features. It falls within the discipline of computer science, both depending on & affecting mathematics, software engineering, linguistics & even cognitive science.

*



Modules have their own local data & also share the global data.

Structured program.

Bohm Jacopini structured program theorem :-

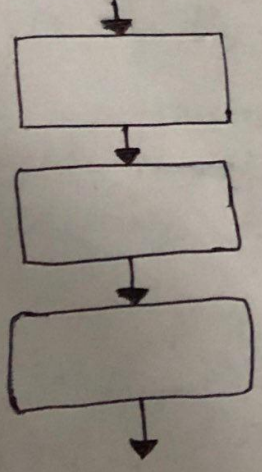
The history of structured programming began with Corrado Bohm and Giuseppe Jacopini in 1964 when they proved in their presented paper that only three control structures were required to write any program. The theorem made the goto statement unnecessary even though no programmer during the mid 1960's could write a code without it. As a result, the theorem was not widely accepted even when it was translated to English, but it served as a base in further research.

Bohm - Jacopini Theorem :-

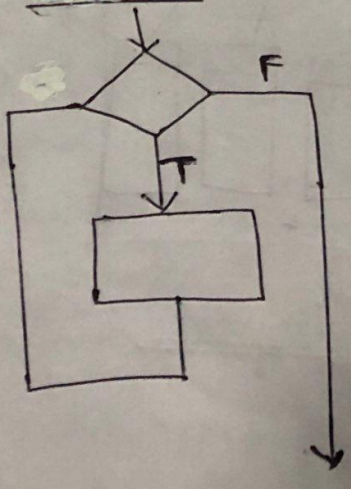
The Bohm - Jacopini theorem, also called structured program theorem, stated that working out a function is possible by combining sub-programs in only three manners :-

1. **Sequence** :- Executing one subprogram, & the other sub-program.
2. **Selection** :- Executing one of two subprograms according to the value of a Boolean expression, also called Decision.
3. **Iteration** :- Executing a subprogram until a Boolean expression is true. Also called Looping.

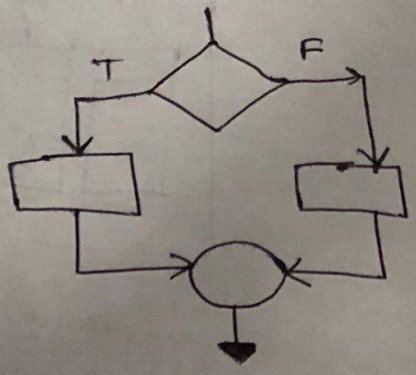
Sequence



Iteration

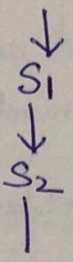


Selection



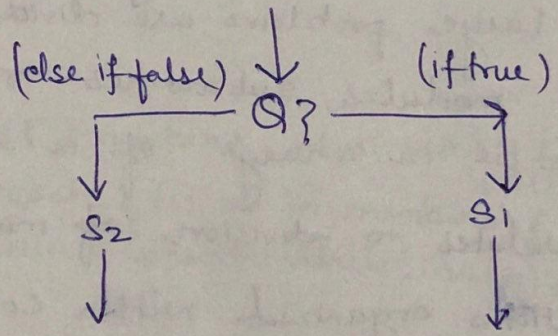
Using flowcharts, the control structures can be illustred like this

- Sequence (first do this, then do that): $S_1; S_2$



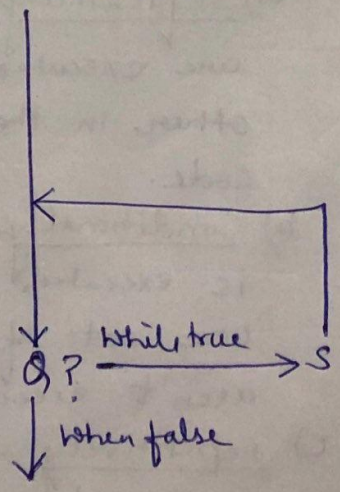
- Selection (or Decision) → (IF some statement Q is true,
  THEN do this
  ELSE do that)

IF Q
THEN S1
ELSE S2



- Repetition (or Looping) :- (WHILE some statement Q is true
  Do this).

WHILE Q DO S



Structured Programming was defined as a method used to minimize complexity that uses:

1. top-down analysis for problem solving
2. modularization for program structure & organization.
3. structured code for the individual modules.

1. Topdown analysis :- includes solving the problem & providing instructions for every step. When developing a solution is complicated, the high approach is to divide a large problem into several smaller problems & tasks.

2. Modular programming :- is a method of organizing the instruction of a program. Large problems are divided into smaller sections called modules, subroutines, or subprograms. Each subroutine is in charge of a specific job.

3. Structured coding :- relates to division of modules into set of instructions organized within control structures. A control structure defines the order in which a set of instructions are executed. The statements within a specific control structure are executed:

   a) Sequentially - denotes in which order the controls are executed. They are executed one after the other in the exact order they are listed in the code.

   b) Conditionally - allow choosing which set of control is executed. Based on the needed condition, one set of commands is chosen while others aren't executed.

   c) repetitively - allows the same control to be performed over & over again. Repetition stops when it meets certain terms or performs a defined no.

of iteration.

Advantages of structured programming

- Programs are more easily & more quickly written.
- Programs have greater reliability
- Programs require less time to debug & test.
- Programs are easier to maintain.
- User Friendly, Machine Independent,

Representations in different Languages :-

Java :

```
List <int> numbers = new ArrayList<>();
...
for (int n : numbers) {
System.out.println(" " +n);
}
```

C# :

```
int [] numbers = new int [] {0,1,1,2,3,5 & 13};
foreach (int n in numbers) {
System.Console.WriteLine(n);
}
```

Ruby :

```
array = [1,2,3,4,5]
array.each do|i|
puts i
end
```

Python :-

```
for var in list(range(5)):
    print (var)
```

Disadvantages :-

1) Since it is machine-Independent, so it takes time to convert into machine code.
2) The converted machine-code is not the same as for assembly language.
3) The program depends upon changeable factors like data-types. Therefore it needs to be updated with the need on the go.
4) Usually the development in this approach takes longer time as it is language-dependent. Where-as in the case of assembly language the development takes lesser time as it is fixed for the machines.

Structured Programming in Python

(1) 'print()' function :- (Sequence)

```
print ('Hello users, welcome to Python')
print ("Hi")
print ("Hi 'Python' ")   o/p  Hi 'Python'
print ('Hi "Python" ')   o/p  Hi "Python"
```

(2) Airthmatic Operations

+, -, /, *, %, //, **

add, sub, div, mul, remainder, floor division, Power

(3) input() :- used to take input from user

```
a = int (input ('Enter a no:'))
a = float (input ('Enter a decimal no:'))
a = input ('Enter a string')
```

By default return type of input() is 'string'.

Dynamically Typed

```
a=10 #take by default int
a=10.0 # float
a='abc' # string
```

Note:- no need to specify data-type.

4) type() :- used to display type of a variable.

```
a=10
print (type(a))      o/p  <class 'int'>

a=10.0
print(type(a))       o/p  <class 'float'>

a='abc'
print(type(a))       o/p  <class 'str'>
```

5) Override allowed

```
a=10
print (type(a))
a = "durga"
print (type(a))
```

o/p: <<class 'str'>

Decision Making in Python :-

(if, if-else, Nested if, if-elif-else ladder)

(1) If statement :-

Syntax :-

```
if condition :
    # Statements to execute if
    # condition is true.
```

Python Code

```
i=10
if (i>15):
    print ("10 is less than 15")
print ("I am not in if")
```

o/p - I am not in if

(2) If-else :-

Syntax

```
if (condition):
    #Executes this block if
    #Condition is true.
else:
    # Executes this block if
    # condition is false
```

```
i=201
if (i<150):
    print ("i is smaller than 150")
    print ("I'm in if Block")
else:
    print ("I is greater than 150")
    print ("I'm in else Block")
print ("I'm not in if and not in else Block")
```

O/P
```
I is greater than 150
I'm in else Block
I'm not in if and not in else Block
```

3) if...elif...else

Syntax :-

```
if (condition):
    Body of if
elif (condition):
    Body of if
else:
    Body of else
```

e.g.

```
num = 3.4
if num>0;
    print("Positive number")
elif num==0:
    print ("zero")
else:
    print ("Negative number")
```

O/P. Positive number

4. Nested if :-

We can have a if...elif..else statement inside another if...elif...else statement.

eg.

```
num = float(input("Enter a number:"))
if num>=0:
    if num == 0:
        print("zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

O/P.

```
Enter a no: 5
Positive number

Enter a no: -1
Negative number
Enter a no: 0
zero
```

* To check whether number is positive or negative.

→
```
num = int(input("Enter a number:"))
if num > 0:
    print("Positive number")
elif num == 0:
    print("zero")
else:
    print("Negative number")
```

i/p: 3.4, 0, -4.5.

or
```
num = float(input("Enter a number:"))
if num >= 0:
    if num == 0:
        print("zero")
    else:
        print("Positive no")
else:
    print("Negative no")
```

Loop in Python :-

The for loop in Python is used to iterate over a sequence (list, tuple

→
```
fruits = ["apple", "banana", "cherry"]
for x in fruits
  print(x)
```

→
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

o/p apple, banana

```
for x in fruits
  if x == "banana":
      break
  print(x)
```

apple.

* Range function

→
```
for x in range(6):
    print x
```

o/p- 0 to 5 not six

→
```
for x in range(2,6):
    print(x)
```

o/p 2, 3, 4, 5

Print the Pattern

→
```
n=int(input("enter the no. of rows:"))
for i in range(1,n+1):
    for j in range(1, i+1):
        print(j, end=" ")
    print()
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

→ Factorial of a number :-

```
num = int(input("Enter a number"))
Enter a number: 5
factorial = 1
if num < 0
    print("factorial not exist")
elif num == 0:
    print("fact is 1")
else:
    for i in range(1, num + 1):
        factorial = factorial * i
print("The factorial of", num, "is", factorial)
```

→ Printing pattern :-

```
num = int(input("enter no. of rows"))
for i in range(1, num+1)
    for j in range(1, i+1):
        print("*", end = " ")
    print()
```

```
*
* *
* * *
* * * *
* * * *
```

end = " " for same line with space.

→ Vowel & Consonant :-

```
ch = input("Enter a character:")
if (ch == 'A' or ch == 'a' or ch == 'E' or ch == 'e' or
    ch == 'I' or ch == 'i' or ch == 'O' or ch == 'o' or
    ch == 'U' or ch == 'u'):
        print(ch, "is a vowel")
else:
    print(ch, "is a consonant")
```

While Loop :-

Syntax

```
i = 1
while i<=5 :
    print ("Hello")
    i = i+1
```

Floor division

9//2 = 4

for loop

Program for even & odd.

```
num = int(input("Enter a number:"))
if (num % 2) == 0:
        print("{0} is Even".format(num))
else:
        print("{0} is Odd".format(num))
```

format() function is used for accessing arguments by position

Armstrong No:-

```
num = int(input("Enter a no."))
sum = 0
temp = num
while temp > 0:
        digit = temp % 10
        sum += digit **3
        temp //= 10

if
if num == sum:
    print(num, "is a Armstrong numbe")
else:
    print(num, "is not an Armstrong no.")
```

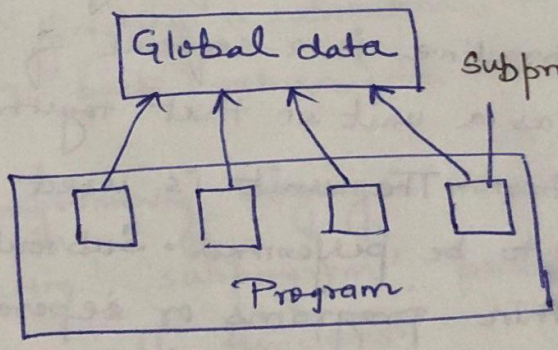
O/P:-663
407

Procedural Programming

Procedural Languages are computer languages used to define the actions that a computer has to follow to solve a problem.

In procedural languages, a program is divided into n number of subroutines that access global data. To avoid repetition of code, each subroutine performs a well-defined task. A subroutine that needs the service provided by another subroutine can call that subroutine.

Therefore, with 'jump', 'goto', & 'call' instructions, the sequence of execution of instructions can be altered.



Structure of a procedural program.

Procedural programming is a programming paradigm, derived from structured programming. Structure is based on the concept of the procedure call.

There are different names for procedures:

- Routines
- Subroutines
- Methods
- Functions

Procedure might be called at any point during a program's execution, including by other procedures or itself (recursion).

Procedural programming is also referred to as imperative or structured programming.

Routines :- Procedural programming relies on procedures, also known as routines. A procedure contains series of computational steps to be carried out (we call that a function). Procedural programming is intuitive in the sense that it is very similar to how you would except a program to work.

→ If you want a computer to do something, you should provide step-by-step instructions on how to do it. Many of the early programming languages like FORTRAN and COBOL are procedural.

→ A common technique in procedural programming is to repeat a series of steps using iteration.

Subroutine :- Subroutine is a sequence of program instructions packaged as a unit so that together they perform a specific task. The unit is used wherever that task needs to be performed. Subroutines can be defined within programs or seperately (libraries used by multiple programs).
Depending on the programming language, subroutine can be called a

→ procedure
→ function,
→ routine
→ method
→ or a subprogram.

ie. "Callable unit" is sometimes used as a generic term.

Callable units behave in the same manner as a program but they are coded so they can be called several times. They can also be called from different places like from the program itself or other subroutines after which they return to the first instruction that comes after the call.

If a subroutine is coded so that it calls itself, it is recursive.

Subprograms can also have variables called Local variables. Subroutines should also perform only one task & have minimal dependency on the rest of the code.

2nd Functions:- A function is a named block of code that performs a task & then returns control to the caller. A function is often executed (called) several times, from several different places, during a single execution of the program. After finishing a subroutine, the program will branch back (return) to the point after the call.

Note that other programming Languages may distinguish between a 'function', 'subroutine', 'subprogram', 'procedure', or 'method', but in C, these are all functions.

Every function is defined exactly once while the program can declare & call a function as many times as necessary.

Example:- Suppose you writing code to print out the first 5 squares of numbers, do some intermediate processing, then print the first 5 squares again. We could write it like this:

In C without function.

```
#include <stdio.h>
int main (void)
{
int i;
for (i=1; i<= 5 ;i++)
{
printf ("%d", i * i);
}
for (i=1; i<=5 ; i++)
{
printf("%d", i*i);
}
return 0;
}
```

Using function

```
#include <stdio.h>
void Print_Squares (void)
{
 int i;
for (i=1; i<=5; i++)
{
 print ("%d", i*i);
}
}
int main (void)
{
Print_Squares();
Print_Squares();
return 0;
}
```

Recursion: A function calling itself is called recursion and normally there will be a condition that stops the recursion after a small, finite (defined) number of steps. Here's a simple function that does an infinite loop.

```
void infinite_recursion()
{
 printf ("Infinite loop!\n");
 infinite_recursion ();
}
```

If the call of the below function is printMe(3,0); the function will print the line Recursion 3 times.

```
void printMe (int j, int depth)
{
if (depth <j) {
Printf ("Recursion ! depth = %d j = %d\n", depth, j);
printMe (j, ++depth);
}
}
```

→ Recursion is most often used for jobs such as
a) directory tree scans,
b) seeking for the end of a linked list,
c) parsing a tree structure in a database &
d) factoring numbers

Functions in Python:-

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

1) Function blocks begin with the keyword def followed by the function name & parentheses (( )).
2) Any input parameters or arguments should be placed within these parentheses. You can also define paramters inside these parentheses.
3) The code block within every function starts with a colon (:) & is indented.
4) The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def function (parameters):
    function_suite
    return [expressions]
```

```
# Function definition is here
def printme(str):
    print(str)
    return

# Now you can call printme function
printme("This is first call to user defined function!")
printme("Again second call to the same function!")
```

o/p. This is first call to user defined function!
Again second call to the same function.

Program to print hello using function in Python

```
def hello():
    print("Hello")

hello()
```

Program to find factorial of a given no. in Python

```
def fact(n):
    if n==0:
        return 1
    else:
        return n*fact(n-1)

n=int(input("enter the number:"))
result=fact(n)
print("factorial of ",n," is ",result)
```

* Concepts & their Representation of Procedural Programming

1. Modularity :-

Modularity is the degree to which a system's components may be separated & recombined.
A modular approach encourages taking one step at a time, promotes team work & collaboration, & promotes testing on small & isolated units prior to integrating them into a large system.

→ Modules are self-contained units which can usually be reused & are easy to maintain & debug.

2. Variables :-

A variable is nothing but a name given to a storage area that programs can manipulate.
Eg. char, int, float, double are variables types in C.

3. Special variables- pointers :-

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.
A pointer is a variable whose value is the address of another variable, ie direct address of the memory location.

e.g. type *var_name; // syntax of pointer in C

e.g. int *ip;

int *ptr = NULL; // NULL pointer

Null pointer is a constant with a value of zero defined in several standard libraries.

4. Variable Scope:-

There are three places where variables can be declared in C programming language:

a. Local Variables:- Inside a function or a block

b. Global Variables:- Outside of all functions

c. In the definition of function parameters where they are called formal parameters.

e.g:- of global & local variables in C

```
#include <stdio.h>
/* global variable declaration */
int g;
int main()
{
/* local variable declaration */
int a, b;
/* actual initialization */
a=10
b=20
g= a+b;
Printf("value of a=%d, b=%d and g=%d\n", a, b, g);
return 0;
}
```

A program can have same name for local & global variable but value of local variable inside a function will take Preference. Following is an example:-

```
# include <stdio.h>
int g=20; // global variable
int main()
{
int g=10; //local variable
printf ("value of g =%d\n", g);
return 0;
}
```

o/p:

value of g=10

*Scope of variables inside block (function):

```
Int a;   // global variable
void f(void)
{
float a; // local
float b;

a=0.0; // global 'a' is unaffected
 {
  int a; //new variable
  a= 2; // outer 'a' is still set to 0.0
 }
 b=a;     /* sets 'b' to 0.0
}
// global 'a' is unaffected by anything done in f().
```

5. Static function:-

If a function should be called only from within the file in which it is declared, it is appropriate to declare it as a static function.
When a function is declared static, the compiler will compile the function to an object file in a way that prevents it from being called within a code in other files. Eg.

```
Static int compare (int a, int b)
{
 return (a+4<b)? a:b;
}
```

3rd Program to find fibonacci series using Recursion in Python.

```
def fib(n):
    if (n<0):
        print ("Incorrect input")
    elif n<=1:
        return n
    else:
        return (fib(n-1) + fib(n-2))
n=int (input ("Enter no. of terms"))
for i in range (n):
    print (fib(i))
```

O/p Enter a no. of terms 8

0
1
1
2
3
5
8
13

Other languages in Procedural Programming

1) Bliss :- "Basic Language for Implementation of System Software" is a system programming language created around 1970. It is typeless, Bliss is expression-oriented. It was perhaps the best known systems prog. lang. right up until C made its debut a few years later.

2) Chuck :- Chuck is a strongly-typed, strongly-timed, concurrent audio and multimedia programming language. It is compiled into virtual instructions, which is immediately run in the Chuck Virtual Machine.

3) Matlab :- Matrix Laboratory, MATLAB is the easiest & most productive software environment for engineers & scientists.

Imp Object-Oriented Programming

Object-oriented programming paradigm is based on the concept of objects, which are data structures that contain data, in the form of fields, often know as attributes; and code, in the form of procedures, often known as methods.

In object-oriented programming, programs are designed by making them out of objects that interact with one another.

Most popular languages are based, meaning that objects are instance of classes, which typically also determines their type.

→ Many of the widely used programming languages are multi-paradigm programming languages that support object-oriented programming, typically in combination with imperative, procedural programming. Some of the most significant object-oriented languages include C++, Java, Delphi, C#, Python, Smalltalk.

→ More recently, a number of languages have emerged that are primarily object-oriented, but that are also compatible with procedural methodology. Two such languages are Python and Ruby.

OOPS concepts:-

1) Polymorphism
2) Inheritance
3) Encapsulation
4) Abstraction

— Main Features.

5) Class
6) Objects
7) Methods

Objects :- An object is an abstract data type with the addition of polymorphism and inheritance.

An object has both state (data) and behaviour (code)

The object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the system. Instead, the data is accessed by calling specially written functions, called **methods**, which are bundled with the data.

The construct that combines data with a set of methods for accessing and managing data is called an object.

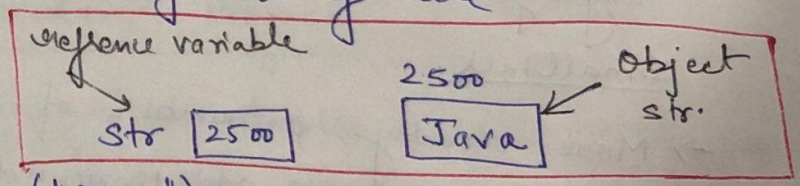
→ Program contains instructions to create objects. For the computer to be able to create an object, there has to be a definition, called a class.

→ An object is called an instance of a class, & it is an instance of exactly one class.

→ Using the operator new to create a class object is called **instantiating** an object of that class.

E.g. In Java, consider the following code

```
String str;
Str = "Java";
str = new String("Java");
```



In java, **new** is an operator. It causes the system to allocate memory space of a specific type, store specific data in that memory space, & return the address of the memory space.

The memory space 2500, where the string is stored, is called a string object.

String objects are called **instances** of the class String.

Methods :-

A method in object-oriented programming is a procedure associated with an object class.
An object is made up of behaviour and data where data is represented as properties of the object & behavior as methods.

→ Methods are also the interface an object presents to the outside world.

Example :- a Window object that has methods open and close.

Overriding :-

One of the most important capabilities that a method provides is overriding.
This is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes.

→ If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.

Encapsulation :- Methods also provide the interface that other classes use to access and modify the data properties of an object.

8mp

Python Code for Object & Classes

```
Class MyClass: // class keyword.
    x=5        // property          o/p = 5
p1=MyClass()   // p1 is object
print(p1.x)
```

The __init__() Function (Class Constructor)

All classes have a function called __init__(), which is always executed when the class is being initiated.

Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

Example

Create a class named Person, use the __init__() function to assign values for name and age:

```
Class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
p1 = Person("John", 36)
print(p1.name)
print(p1.age)
```

Note:- The __init__() function is called automatically every time the class is being used to create a new object.

Object Methods:

Objects can also contain methods. Methods in objects are functions that belong to the object.

Example :-

Insert a function that prints a greeting, & execute it on the p1 object of Person class.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Note :- The 'self' parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter does not have to be named 'self', you can call it whatever you like, but it has to be the first parameter of any function in the class:

```
class Person:
    def __init__(mysillyobject, name, age):
        mysillyobject.name = name
        mysillyobject.age = age

    def myfun(abc):
        print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Modify Object Properties :-

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def myfunc(self):
        print("Hello my name is" + self.name)
p1 = Person("John", 36)
p1.age = 40
print(p1.age)
```

Delete Object Properties

```
del p1.age          // dw
```

Delete Objects

```
del p1
```

pass Statement

Class definitions cannot be empty, but if you ~~to~~ have a 'class' definition with no content, put in the pass statement to avoid getting an error.

```
class Person:
    pass
```

* Syntax of Constructor declaration :-

```
def __init__(self):
    # body of the constructor
```

Constructors are generally used for instantiating an object. The task of constructors is to initialize (assign values) to the data members of the class when an object of class is created. In Python __init__() method is called the constructor & is always

Called when an object is created.

Types of Constructors in Python

(a) default Constructors:- The default constructor is simple constructor which doesn't accept any arguments. It's definition has only one argument which is reference to the instance being constructed.

(b) parameterized Constructor:- constructor with parameters is known as parameterized constructor. The parameterized Constructor takes its first argument as a reference to the instance being constructed known as self & the rest of the arguments are provided by the programmer.

* Destructors in Python

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much needed in C++ because Python has a garbage collector that handles memory management automatically.

__del__() method is a known as a destructor method in Python. It is called when all references to the object have been deleted ie when an object is garbage collected.

Syntax of destructor declaration:-

```
def __del__(self):
    # body of destructor.
```

e.g:
```
class Employee:
    def __init__(self):
        print("Employee created")
    def __del__(self):
        print('Destructor called, Employee deleted.')
obj = Employee(); del obj
```

o/p: Employee created
Destructor called, Employee deleted.

Classes and instances

Class :- A class is a collection of a specific number of components. The components of a class are called the members of the class.

→ Class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions, methods).

→ A class is a collection of methods and data members. Methods are often called modules.

→ In many languages, the class name is used as the name for the class, the name for the default constructor of the class, and as the type of objects generated.

Instance :- When a constructor creates an object, that object is called an instance of the class.

→ The creation of an instance is called instantiation. Not all classes can be instantiated, like abstract classes, while those that can be instantiated are called Concrete class.

After instantiation, the member variables, specific to the object are called instance variables, to contrast with the class variables shared across the class.

Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP).

"It describes the idea of wrapping data & the methods that work on data within one unit."

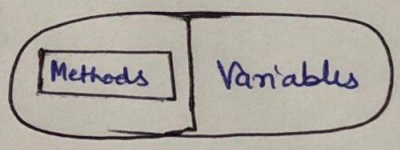
This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.

To prevent the accidental change, an object's variable can only be changed by an object's method.

Those type of variables are known as "private variable"

Example:-

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables etc.



Class

Data encapsulation:- also called data hiding, organizes the data and methods into a structure that prevents data access by any function (or method) that is not specified in the class. This ensures integrity of the data contained in the object. Encapsulation defines different access levels for data variables and member functions of the class. These access levels specifies the access rights, for eg.

(a) Public:- Any data or function with access level public can be accessed by any function belonging to any class. This is lowest level of data protection.

(b) Private :- Any data or function with access level private can be accessed only by the class in which it is declared. This is the highest level of data protection. In python, private variables are prefixed with a double underscore (__).

For eg. __var is a private variable of the class.

Note:- Functions defined inside a class are called class methods. (Private function)

```
class car:
    def __init__(self):
        self.__updatesoftware()
    def drive(self):
        print("driving")
    def __updatesoftware(self):
        print("updating software")

a = car()
a.drive()
```

O/p: updating software
driving

```
class car:
    #def __init__(self):
        # self.__updatesoftware()
    def drive(self):
        print("driving")
    def __updatesoftware(self):
        print("updating software")

a = car()
a.drive()
a.__updatesoftware() # this will give error as in class this private function.
```

Private Variables :-

```
Class car:
    __maxspeed = 0
    __name = " "
    def __init__(self):
        self.__maxspeed = 200
        self.__name = "supercar"
    def drive(self):
        print("driving"); print(self.__maxspeed)
#def setspeed(self, speed):
    # self.__maxspeed = speed
    # print(self.__max_speed)
a = car()
a.drive()
#a.setspeed(100)
a.__maxspeed = 100      // can't modify.
a.drive()
```

O/P:
```
driving
200
driving
200
```

* Polymorphism :- Poly - many morphism - forms

Polymorphism is the ability of an object to take on many forms. The most common use of "polymorphism" in OOP occurs when a parent class reference is used to refer to a child class object.

In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class.

For e.g. in python 'print' function can take int, float and string as argument. same function can have many arguments depending upon the requirements

Python code to show polymorphism

```
Class Demo:
    def display(self):          // Function 1
        print('Value:', 0)

    #def display(self, x):      // Function 2
        #print('Value:', x)

D = Demo()
D.display()
```

O/P Value : 0

If we don't pass the argument in display function then it will take first display function, else second.

```
Class Demo:
    def display(self):
        print('Value', 0)
    def display(self, x):
        print('value:', x)
D = Demo()
D.display(10)
```

O/P:

Value : 10

Note:- Always latest method will be available for calling.

Using Inheritance :-

```
class A:
    def display(self):
        print('Value:', 0)

class B(A):
    def display(self):
        print('Value:', 1)

b = B()
b.display()
```

Output :-

Value : 1

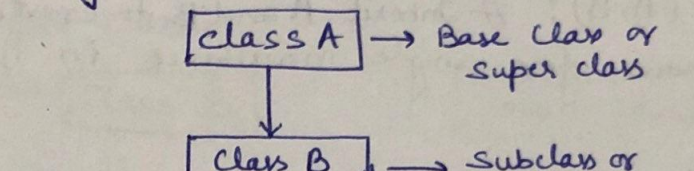
* Inheritance :-

The capability of a class to derive properties and characteristics from another class is called Inheritance. It is one of the important feature of OOP.

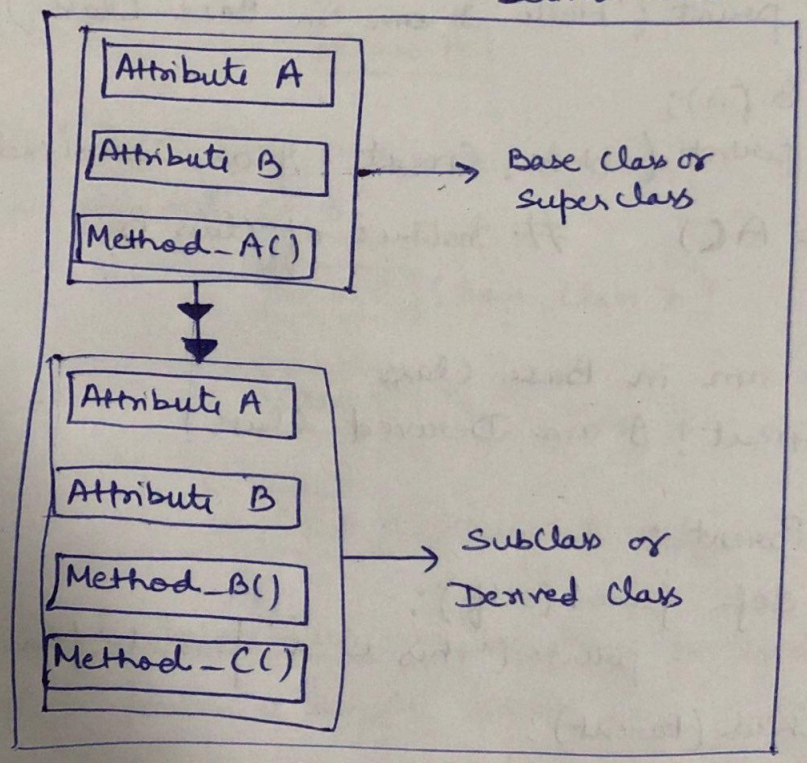
Derived class or Sub class :- The class that inherits properties from another class is called sub-class or Derived class.

Base class or Super class :- The class whose properties are inherited by sub-class is called Base class

e.g.



or



Simple example of inheritance.

The procedure of creating a new class from one or more existing classes is called inheritance.

→ When a class inherits another class it inherits all features (like variables and methods) of the parent class.

→ A new child class automatically gets all of the methods & attributes of the existing parent class.

The syntax of define inheritance (to inherit a single base class) in Python is:

```
class Derived_Class_Name (Single_Base_Class_Name):
        Body_of_Derived_Class
```

The syntax to inherit multiple base class is:

```
class Derived_Class_Name (Comma_Seperated_Base_Class_Name):
        Body_of_Derived_Class
```

e.g. `class C(A,B): # Inherit A and B to create New class C.`

* Simple program for single inheritance in Python.

```
Class A:
        print ('Hello I am in Base Class')
Class B (A):
        print ('WOW! Great! I am Derived Class!')
Ob2 = A()      # Instance of class B
```

Output

```
Hello I am in Base Class
WOW! Great! I am Derived Class!
```

⇒

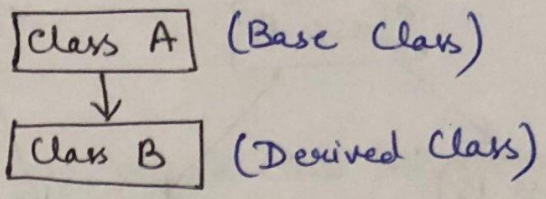
```
class Parent:
        def func1(self):
                print ('this is a parent function')
class Child (Parent):
        def func2(self):
                print ("this is a child function")
Ob = Child()
Ob.func1()
Ob.func2()
```

O/P:

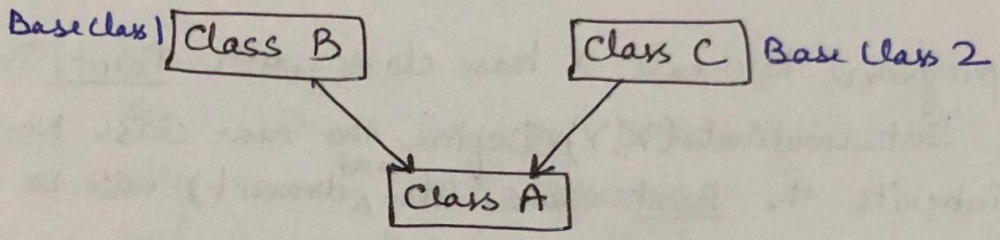
```
this is a parent function
this is a child function.
```

Types of Inheritance

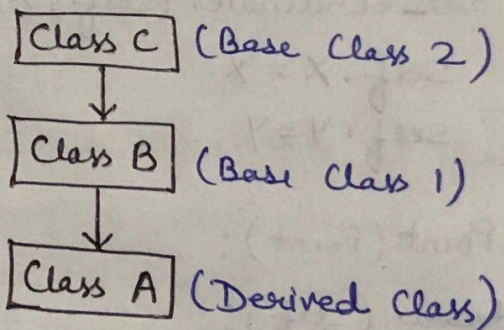
(1) **Single Inheritance:-** In this a class is allowed to inherit from only one class, i.e. one subclass is inherited by one base class only.



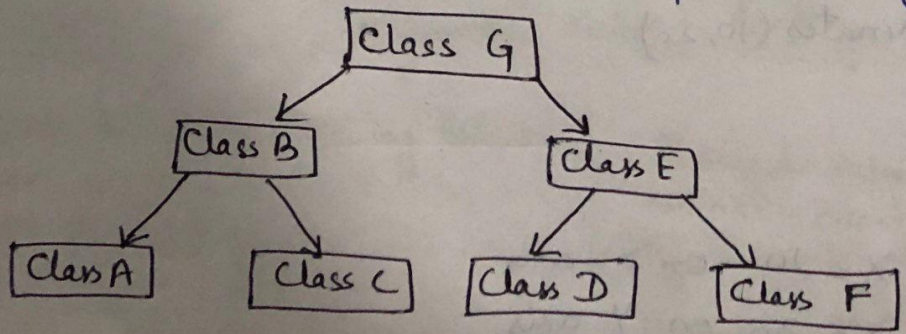
(2) **Multiple Inheritance:-** A class can inherit from more than one classes i.e. one sub class is inherited from more than one base class.



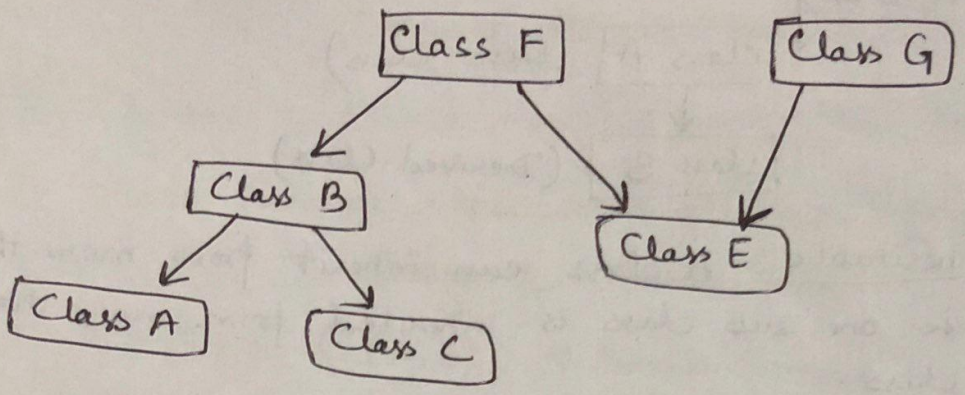
3) **Multilevel Inheritance:-** A derived class is created from another derived class.



4) **Hierarchical Inheritance:-** more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



5. Hybrid (Virtual) Inheritance :- It is implemented by combining more than one type of inheritance. For eg. Combining Hierarchical & multiple inheritance.



Q. Write program to create a base class with Point. Define the method Set_Cordinate(X,Y). Define the new class New_Point, which inherits the Point class. Also add draw() method. inside the subclass.

```
Class Point :                                  # Base Class
    def Set_Cordinates (self, X, Y):
        self.X = X
        self.Y = Y

Class New_Point (Point):                       #Derived Class
    def draw (self):
        print ("Locate Point X= ', self.X,' On X axis')
        print ("Locate Point Y = ', self.Y,' on Y axis)

P = New_Point()
P. Set_Cordinates (10, 20)
P. draw()
```

O/P

Locate Point X = 10 on X axis
Locate Point Y = 20 on Y axis

* Write a simple program to demonstrate the concept of multilevel inheritance.

```
Class A:                                    #Base Class
    name = ' '
    age = 0
Class B(A):                                 #Derived Class Inheriting Base Class A
    height = ' '
Class C(B):
    Weight = ' '                            #Derived Class inheriting Base Class B

    def Read(self):
        print('Please Enter the Following Values')
        self.name = input('Enter Name:')
        self.age = (int(input('Enter Age :')))
        self.height = (input('Enter Height :'))
        self.weight = (float(input('Enter Weight :'))

    def Display(self):
        print("Entered Values are as follows')
        print('Name =', self.name)
        print('Age =', self.age)
        print('Height =', self.height)
        print('Weight =', self.weight)

B1 = C()                    #Instance of Class C
B1.Read()                   #Invoke Method Read
B1.Display()                #Invoke Method Display.
```

O/p:

```
Please Enter the Following Values
Enter Name : Amit
Enter Age : 25
Enter Height : 5"7'
Enter Weight : 60.7

Entered Values are
Name = Amit
Age = 25
Height = 5"7'
Weight = 60.7
```

Abstraction:-

Abstraction means hiding the complexity & only showing the essential features of the object. So in a way, Abstraction means hiding the real implementation & we, as a user, knowing only how to use it.

Real world example

(a) Vehicle which we drive with out caring or knowing what all is going underneath.

(b) A TV set where we enjoy programs without knowing the inner details of how TV works.

Abstraction in Python

Abstraction in Python is achieved by using abstract classes and interfaces.

Abstract class:-

An abstract class is a class that generally provides incomplete functionality and contain one or more abstract methods. Abstract methods are the methods that generally don't have any implementation, it is left to the sub-classes to provide implementation for the abstract methods.

Interface:-

An interface should just provide the method names without method bodies.
Subclasses should provide implementation for all the methods defined in an interface.

Note:- In Python there is no support for creating interfaces explicitly, you will have to use abstract class. In Python you can create an interface using abstract class.

Abstract classes in Python

Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs). This module is called - abc.

The following Python code uses the abc module and defines an abstract base class:

```
from abc import ABC, abstractmethod
class AbstractClassExample(ABC):
    def __init__(self, value):
        self.value = value
        super().__init__() #accessing inherited methods.
    @abstractmethod
    def do_something(self):
        pass
```

Now, we will define a subclass using the previously defined abstract class. We define two classes inheriting from our abstract class:

```
class DoAdd(AbstractClassExample):
    def do_something(self):
        return self.value + 42
class DoMul(AbstractClassExample):
    def do_something(self):
        return self.value * 42
x = DoAdd(10)
y = DoMul(10)
print(x.do_something())
print(y.do_something())
```

O/P: 52
420

Other Languages of Object oriented Programming

1) **BETA** :- BETA is a pure object-oriented language originating within the "Scandinavian School" in object-orientation where the first object-oriented language Simula was developed.
Among its notable features, it introduced nested classes, & unified classes with procedures into so called patterns.

2) **Cecil** :- Cecil is a pure object-oriented programming language that was developed by Craig Chambers at the University of Washington in 1992 to be part of Vorter project there.
Like Objective-C, all object services in Cecil are invoked by message passing, and the language supports run-time class identification.

3) **Lava** :- Lava is an experimental, visual object-oriented, interpreter-based programming language with an associated programming environment.
→ Lava Programming Environment or LavaPE) that uses structure editors instead of text editors.
→ Only comments, constants, and new identifiers may be entered as text.

2mk

Difference Between Structured Programming & Object-Oriented Programming.

Structured Programming	Object-Oriented Programming
1. Structured Programming is designed which focuses on process/logical structure & then data required for that process.	Object-Oriented Programming is designed which focuses on data.
2. Structured programming follows top-down approach	Object-oriented follows bottom-up approach.
3. Structured Programming is also known as Modular Programming and a subset of procedural programming lang".	Object oriented Programming supports inheritance, encapsulation, abstraction, polymorphism, etc.
4. In Structured Programming, Programs are divided into small self contained functions.	In Object Oriented Programming, Programs are divided into small entities called objects.
5. Structured Programming is less secure as there is no way of data-hiding	5. OOP is more secure as having data hiding feature.
6. Structured Programming can solve moderately complex programs.	5. OOP can solve any complex programs.
7. Structured Programming provides less reusability, more function dependency.	OOP provides more reusability less function dependency
8. Less abstraction & less flexibility	More abstraction & more flexibility.