Unit-03/Lecture-01

•Two fundamental abstraction facilities

–Process abstraction

•Emphasized from early days

–Data abstraction

•Emphasized in the1980s

Fundamentals of Subprograms

General Subprogram Characteristics

•Each subprogram has a single entry point

•The calling program is suspended during execution of the called subprogram

•Control always returns to the caller when the called subprogram's execution terminates.

Basic Definition

•A subprogram definition describes the interface to and the actions of the subprogram abstraction

-In Python, function definitions are executable; in all other languages, they are non-executable

•A subprogram call is an explicit request that the subprogram be executed

•A subprogram header is the first part of the definition, including the name, the kind of subprogram, and the formal parameters

•The parameter profile of a subprogram is the number, order, and types of its parameters

•The protocol is a subprogram's parameter profile and, if it is a function, its return type

•Function declarations in C and C++ are often called prototypes

•A subprogram declaration provides the protocol, but not the body, of the subprogram

•Parameter: A formal parameter is a dummy variable listed in the subprogram header and used in the subprogram

•Argument: An actual parameter represents a value or address used in the subprogram call statement

Argument/Parameter Correspondence

•Positional

•The binding of actual parameters (arguments) to formal parameters is by position: the first actual parameter is bound to the first formal parameter and so forth

•Safe and effective

•Keyword

•The name of the formal parameter to which an actual parameter (argument) is to be bound is specified with the actual parameter

•Advantage: Parameters can appear in any order, thereby avoiding parameter correspondence errors

•Disadvantage: User must know the formal parameter’s names

Parameters

Actual/Formal Parameter Keyword Correspondence: Python Example

sumer

(length = my_length

,list = my_list

,sum = my_sum

)

Parameters: length, list, sum

Arguments: my_length, my_list, my_sum

Unit-03/Lecture-02

Formal Parameter Default Values

•In certain languages (e.g., C++, Python, Ruby, Ada, PHP), formal parameters can have default values (if no actual parameter is passed)

•In C++, default parameters must appear last because parameters are position-ally associated

•Variable numbers of parameters

•C# methods can accept a variable number of parameters as long as they are of the same type

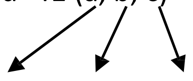
—the corresponding formal parameter is an array preceded by parameters.

•In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is preceded by an asterisk.

•In Python, the actual is a list of values and the corresponding formal parameter is a name with an asterisk

- **Positional**–The binding of actual parameters to formal parameters is by position. The first actual parameter is bound to the first formal parameter and so forth. It is Safe and effective

Caller d= f1 (a, b, c)



int f1(int x, float y, int z)

- **Keyword**- The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter

In Ada, Sumer (Length => My_Length, List => My_Array, Sum => My_Sum);

Advantage: Parameters can appear in any order.

Disadvantage: User must know the formal parameter’s names.

Unit-03/Lecture-03

Procedures and Functions

•There are two categories of subprograms

•Procedures are collection of statements that define parameterized computations. These computations are enacted by single call statements. Procedure defines new statements. For Ex- Ada doesn’t have a sort statement, a user can build a procedure to sort arrays of data and use a call to that procedure in place of unavailable sort statement.

•Functions structurally resemble procedures but are semantically modelled on mathematical functions. If a function is a faithful model, it produces no side effects that are; it modifies neither its parameter nor any variables defined outside the function. Such a pure function returns a value that is only desired effect.

•They are expected to produce no side effects.

•In practice, program functions have side effects.

Design Issues for Subprograms

•Are local variables static or dynamic?

•Can subprogram definitions appear in other subprogram definitions?

•What parameter passing methods are provided?

•Are parameter types checked?

•If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?

•Can subprograms be overloaded?

•Can subprogram be generic?

Local Referencing Environments

Variables that are defined inside subprograms are called local variables. Local variables can be either static or stack dynamic "bound to storage when the program begins execution and are unbound when execution terminates." [redacted]

Advantages of using stack dynamic:

a. Stack dynamic variables are essential for recursive subprograms.
b. Flexibility they provide the subprogram.
c. Storage for locals is shared among some subprograms.

Disadvantages:

a. Allocation/de allocation time- There is the cost of the time required to allocate initialize and de allocate such variables for each call to the subprogram.
b. Indirect addressing "only determined during execution."- Access to stack dynamic local variables must be indirect whereas accesses to static variables can be direct. This indirectness is because the place in the stack where a particular local variable will reside can be determined only during execution.
c. Finally when all local variables are stack dynamic, subprograms cannot be history sensitive, that is, they cannot retain data values of local variables between calls.

Ex- for a history- sensitive subprogram is one whose task is to generate pseudo random numbers. Each call to such a subprogram computes one pseudorandom number, using the last one it computed.

Advantages of using static variables:

a. Static local variables can be accessed faster because there is no indirection.

b. No run-time overhead for allocation and de allocation.

c. Allow subprograms to be history sensitive.

Disadvantages:

a. Inability to support recursion.

b. Their storage can't be shared with the local variables of other inactive subprograms.

Ex:

```
int adder( int list[ ], int listlen) {

static int sum = 0;

int count; //count is stack-dynamic

for (count = 0; count < listlen; count++)

sum += list[count];

return sum;

}
```

Here the variable sum is static and count is stack dynamic. Ada, C++, Java and C# have only stack dynamic local variables.

Unit-03/Lecture-04

Parameter Passing Methods

Following are the ways in which parameters are transmitted to and/or from called subprograms

– Pass-by-value

– Pass-by-result

– Pass-by-value-result

– Pass-by-reference

– Pass-by-name

Formal parameters are characterized by one of three distinct semantic models-

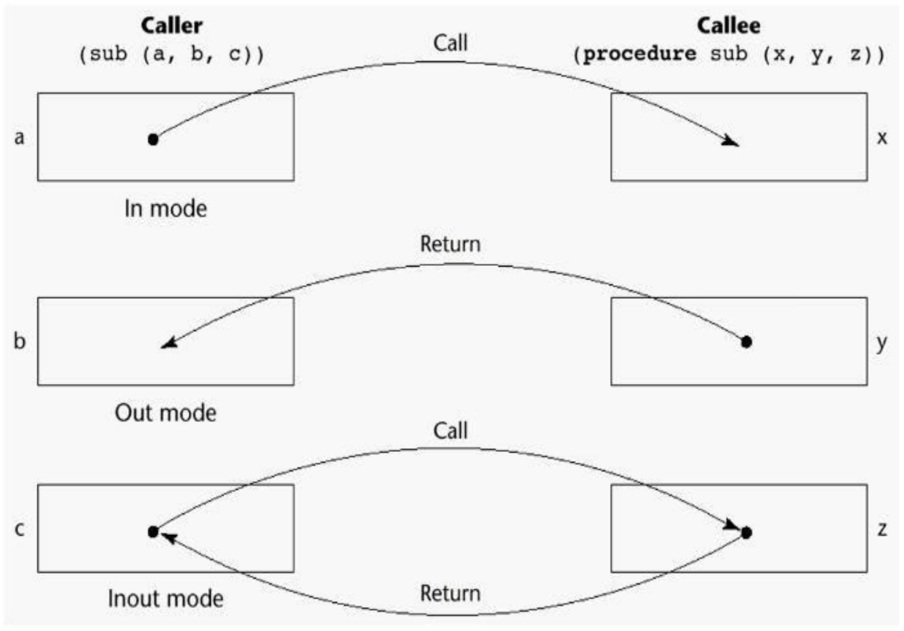
i. They can receive data from the corresponding actual parameters.
ii. They can transmit data to the actual parameter.
iii. They can do both.

These three semantic models are called in mode, out mode, and in out mode respectively.

Data transmission takes place by two ways-

- An actual value is moved.
- An access path is transmitted.

Models of Parameter Passing



1. **Pass by Value (In mode)**

The value of the actual parameter is used to initialize the corresponding formal parameter

–Normally implemented by copying

–Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)

–Disadvantages (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)

–Disadvantages(if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

2. **Pass by Result (out mode)**

When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's

actual parameter when control is returned to the caller, by physical move

–Require extra storage location and copy operation

•Potential problem: sub(p1, p1); whichever formal parameter is copied back will represent the current value of p1

3. **Pass by Value Result (In out mode)**

•A combination of pass-by-value and pass-by-result

•Sometimes called pass-by-copy

•Formal parameters have local storage

•Disadvantages: –Those of pass-by-result

–Those of pass-by-value

4. **Pass by Reference (In out mode)**

•Pass an access path

•Also called pass-by-sharing

•Advantage: Passing process is efficient (no copying and no duplicated storage)

•Disadvantages

–Slower accesses (compared to pass-by-value) to formal parameters

–Potentials for unwanted side effects (collisions)

–Unwanted aliases (access broadened)

5. **Pass By Name (In out mode)**

- By textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding

Unit-03/Lecture-05

Overloaded Subprograms

An overloaded subprogram is one that has the same name as another subprogram in the same referencing environment

–Every version of an overloaded subprogram has a unique protocol

–Unique protocol means that the number, order, or types of parameters must differ or the return type must differ

•C++, Java, C#, and Ada include predefined overloaded subprograms and also allow users to write multiple versions of subprograms with the same name.

Because each version of an overloaded subprogram has a unique parameter profile, the compiler can disambiguate occurrences of calls to them by the different type parameters.

But when parameter coercions are allowed, complicate the disambiguation process enormously. The issue is that if no method's parameter profile matches the number and types of the actual parameters in a method call, but two or more methods have parameter profiles that can be matched through coercions which method should be called. A language designer has to decide how to rank all the different coercions so that the compiler can choose the method that best matches the call.

In Ada, the return type of an overloaded function can used to disambiguate calls. Therefore two overloaded functions can have the same parameter profile and differ only in their return types. Ex- If an Ada program has 2 functions named Fun, both of which take an Integer parameter but one returns an Integer and one returns a float the following call would be legal:-

```
A, B: Integer;

A: = B + Fun (7);
```

In this call, the call to Fun is bound to the version of Fun that returns an Integer because choosing the version that returns a float would cause a type error.

Overloaded subprograms that have default parameters can lead to ambiguous subprogram calls.

Ex- In C++

```
                    void fun (float b= 0.0);

                    void fun();

                    .....

                    fun ();
```

The call is ambiguous and will cause a compilation error.

Unit-03/Lecture-06

Generic Subprograms

A generic or polymorphic subprogram takes parameters of different types on different activations

• Overloaded subprograms provide ad hoc polymorphism

• A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides parametric polymorphism

Examples of parametric polymorphism: C++ template

```
<Class Type>

Type max (Type first, Type second) {

return first > second ? first: second;

}
```

• The above template can be instantiated for any type for which operator > is defined. For example

```
int max (int first, int second) {

return first > second? first : second;

}
```

Another example of Generic Subprogram in C++

A generic subprogram (function) for swapping integer, float and character type elements-

```
# include <iostream.h>

# include <conio.h>

void swap (T &a, T &b)

{
```

```
                        T temp;

                       temp = a;

                       a = b;

                       b = temp;

}

Void main ()

{

                       Int x=10, y=20;

                       Float a=1.2, b= 2.4;

                       Cout<<"Swapping integer values \n";

                       Cout<<"values of x and y before swapping\n";

                       Cout<<" X="<<x<<"Y="<<y;

                       Swap (x,y);

                       Cout<<" values of x and y after swapping\n";

                       Cout<<" X="<<x<<"Y="<<y;

                       Cout<<"Swapping float values \n";

                       Cout<<"values of a and b before swapping\n";

                       Cout<<" A="<<a<<"B="<<b;

                       Swap (a,b);

                       Cout<<" values of a and b after swapping\n";

                       Cout<<" A="<<a<<"B="<<b;

                       getch();

}
```

Unit-03/Lecture-07

Co routines

A co routine is a subprogram that has multiple entries and controls them itself

• Also called symmetric control: caller and called co routines are on a more equal basis

• A co routine call is named a resume

• The first resume of a co routine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the co routine.

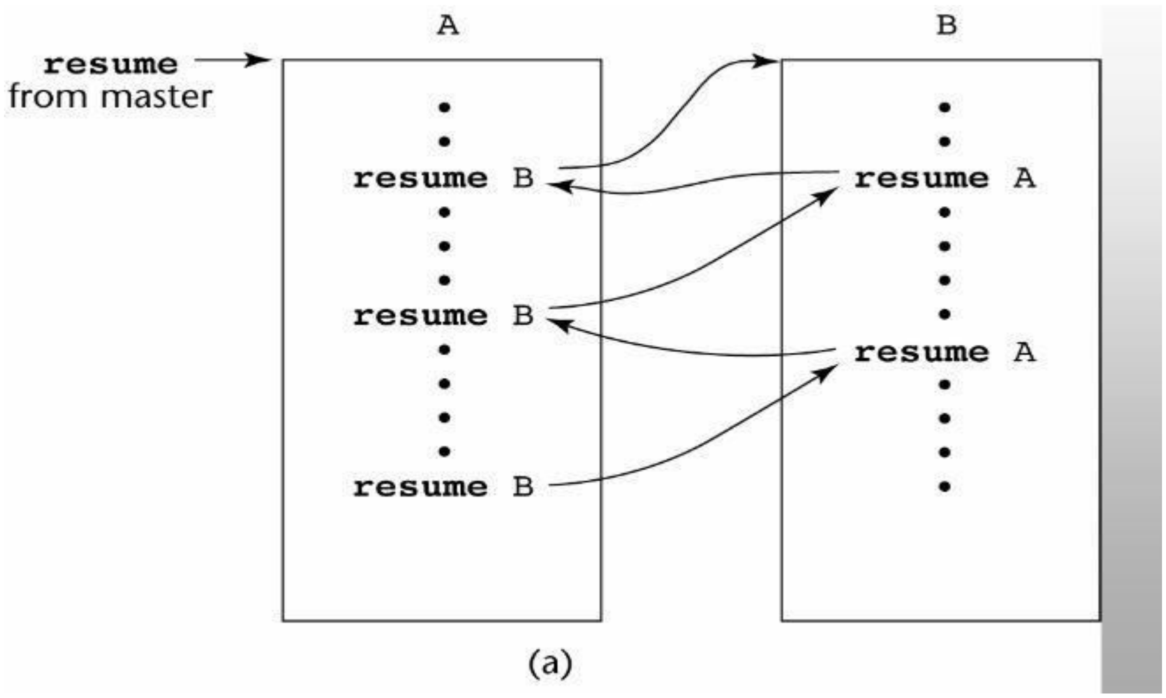
• Co routines repeatedly resume each other, possibly forever

• Co routines provide quasi-concurrent execution of program units (the co routines); their execution is interleaved, but not overlapped

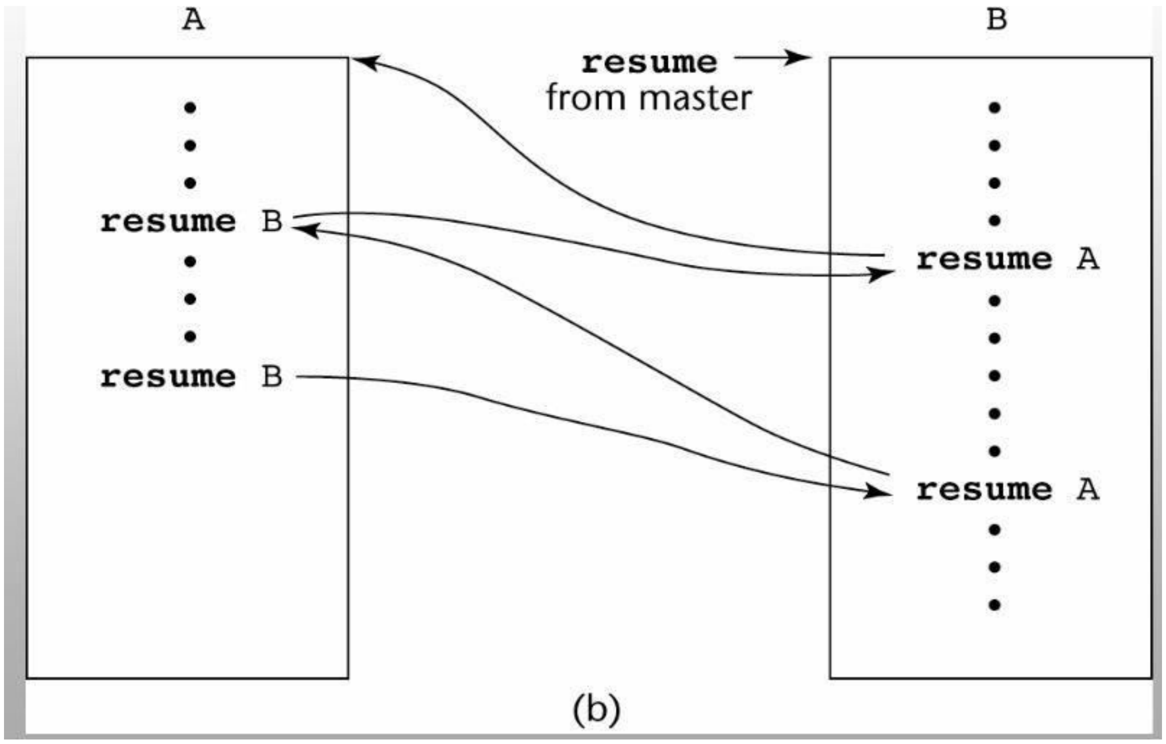
Only one co routine executes at a given time. Rather than executing to their other ends, co routines often partially execute and then transfer control to other routines. When restarted a co routine resumes execution just after the statement it used to transfer control elsewhere. This sort of interleaved execution sequence is related to the way multiprogramming operating systems work. In the case of co routines, this is sometimes called Quasi Concurrency.

Co routines are created in an application by a program unit called Master Unit, which is not a co routine. When created, co routine executes their initialization code and then returns control to that master unit.

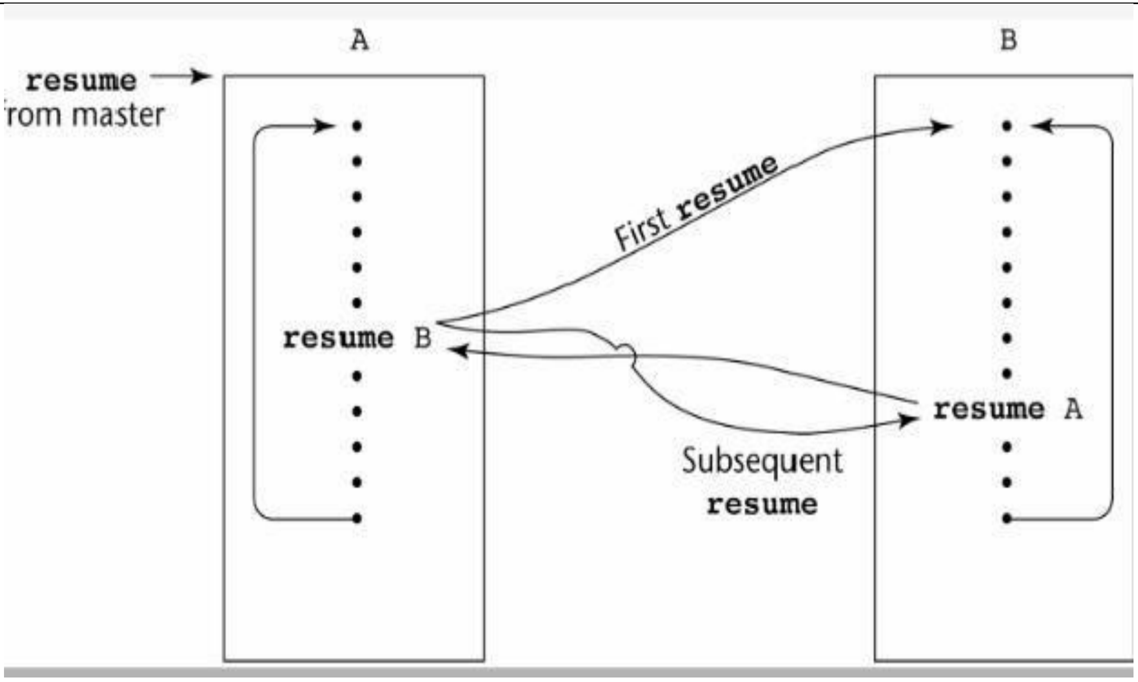
Possible Execution controls



Possible Execution controls



Possible Execution Controls with Loops


A
B
resume
rom master
resume B
First resume
resume A
Subsequent
resume

Unit-04/Lecture-01

Abstraction

An abstraction is a view or representation of an entity that includes only the most significant attributes. There are two kinds of abstraction i.e. process and data abstraction.

Process Abstraction- All subprograms are process abstraction because they provide a way for a program to specify that some process is to be done, without providing the details of how it is done.

Ex SortInt (List, ListLen);

This call is an abstraction of actual sorting process whose algorithm is not specified. This call is independent of the algorithm implemented in the called subprogram.

Data Abstraction- An abstract data type is an enclosure that includes only the data representation of one specific data type and the subprograms that provide the operations for that type.

Ex- C++, Java made it easier for programmer to use ADTs, each ADTs correspond to a class and the operations on the ADT's are public methods. The user of the ADT only needs to know about the method interfaces not the actual implementation.

Benefits-

1. Code is easier to understand.
2. Implementation of the ATDs can be changed without requiring changes to the program that uses ADTs. ADTs support abstraction, encapsulation and information hiding.

Common Examples of ADTs-

1. Built in types: Boolean, integer, real.
2. User defined: stack, queue, tree list, class, and structure.

Data abstraction refers to the data that can be used without taking into account how the data are stored.

An ADT is defined as

1. An ADT is externally defined data type that holds some kind of data

2. An ADT has built in operations that can be performed on it or by it.
3. Users of an ADT don't need to have any detail information about the internal representation of data storage or implementation of the operations.

- Floating Point as an ADT

A floating type provides a set of arithmetic operations for manipulating objects of that type. The actual format of the data value in floating point memory cell is hidden from the user and the only operations available are those provided by the language. The user is not allowed to create new operations on data of the type.

Implementations may use different representations for particular data types. Ex- IEEE 754 standard floating point.

- User Defined ADT-

ADTs are often called User Defined Data Type because they allow programmers to define new types that resemble primitive data types. Just like primitive data types Integer with operations +, -,*, / an ADT has a type domain whose representation is unknown to clents and a set of operations defined on the domain.

A user DT provides the same characteristics provided by built in abstract type.

i. A type definition that allows program units to declare variables of the type but hides the representation of these variables.
ii. A set of operations for manipulating objects of the type.

```
#include <iostream.h>

 class Add

{

private:

int x,y,r;

public:

int Addition(int x, int y)

{

r= x+y;
return r;

}

void show( )

{ cout << "The sum is::" << r << "\n";}

}s;

void main()

{

Add s;

s.Addition(10, 4);

s.show();

}
```

Unit-04/Lecture-02

ENCAPSULATION

When the size of a program reaches beyond a few thousand lines two practical problems arises-

i. Managing or organizing program is difficult.
ii. Recompilation is costlier. For smaller programs, recompiling the whole program is not costly. But for large programs the cost of recompilation is costlier. So there is a need to find ways to avoid recompilation of the parts of a program that are not affected by a change.

The obvious solution to both of these problems is to organize programs into both of these problems is to organize programs into collection of logically related code and data, each of which can be compiled without recompilation of the rest of the program.

It is the process of combining or packaging data and functions. Using the method of encapsulation programmer cannot directly access the data. The data is only accessible through functions present inside the class.

```
public class EncapsulationDemo{

  private int ssn;

  private String empName;

  private int empAge;

  public int getEmpSSN(){

    return ssn;

  }

  public String getEmpName(){

    return empName;

  }

  public int getEmpAge(){
        return empAge;

    }

    public void setEmpAge(int newValue){

        empAge = newValue;

    }

    public void setEmpName(String newValue){

        empName = newValue;

    }

    public void setEmpSSN(int newValue){

        ssn = newValue;

    }

}

public class EncapsTest{

    public static void main(String args[]){

         EncapsulationDemo obj = new EncapsulationDemo();

         obj.setEmpName("Mario");

         obj.setEmpAge(32);

         obj.setEmpSSN(112233);

         System.out.println("Employee Name: " + obj.getEmpName());

         System.out.println("Employee SSN: " + obj.getEmpSSN());

         System.out.println("Employee Age: " + obj.getEmpAge());

    }

}
```

Garbage Collection

Garbage collection is an automatic memory management feature in many modern programming languages, such as Java and languages in the .NET framework. Languages that use garbage collection are often interpreted or run within a virtual machine like the JVM. In each case, the environment that runs the code is also responsible for garbage collection.

In older programming languages, such as C and C++, allocating and freeing memory is done manually by the programmer. Memory for any data that can't be stored within a primitive data type, including objects, buffers and strings, is usually reserved on the heap. When the program no longer needs the data, the programmer frees that chunk of data with an API call. Because this process is manually controlled, human error can introduce bugs in the code. Memory leaks occur when the programmer forgets to free up memory after the program no longer needs it. Other times, a programmer may try to access a chunk of memory that has already been freed, leading to dangling pointers that can cause serious bugs or even crashes.

Programs with an automatic garbage collector (GC) try to eliminate these bugs by automatically detecting when a piece of data is no longer needed. A GC has two goals: any unused memory should be freed, and no memory should be freed unless the program will not use it anymore. Although some languages allow memory to be manually freed as well, many do not.

Benefits-

Garbage collection frees the programmer from manually dealing with memory allocation and deallocation. As a result certain categories of bugs are eliminated.

i. Dangling pointer bugs which occur when a piece of memory is freed while there are still pointers to it and one of these pointers is then used.
ii. Double free bugs which occur when the program attempts to free a region of memory that is already free.
iii. Certain kinds of memory leaks in which a program fails to free memory occupied by objects that will not be used again, leading overtime to memory exhaustion.

Unit-04/Lecture-03

Small Talk

Smalltalk was one of the earliest object-oriented (OO) languages and can be said to be extremely "pure" in an OO sense:

- Everything is an object and objects are only communicated with via the sending of messages
- No primitives (no ints, booleans etc)
- No control structures (no for, while, if etc).
- No statics

Message Sending Syntax

i. **Unary Messages**

The "unary" syntax is used when there are no arguments. In addition to showing the message-send in Smalltalk, we also show the equivalent in Java, as a contrast:

```
x reset       "a message-send in Smalltalk"

x.reset()     // a message-send in Java
```

In Smalltalk, comments are enclosed in double quotes. Here, "x" is a variable and "reset" is the message name

ii. Binary Messages

Binary Messages

Second, we look at "binary" messages. Here is an example:

```
x + y    "in Smalltalk"

x + y    // in Java
```

Even though the Smalltalk syntax looks like Java or C++, there is a very important difference. In Smalltalk, this is a first-class message sending expression. The name of the message is "+", and it is sent to the object named "x". There is a single argument. To compare it to Java, we really ought to compare it to the following, which has an equivalent semantics to the Smalltalk version.

```
x + y        "in Smalltalk"

x.plus(y)    // the Java equivalent
```

With binary messages, there is no presumption about the class of the receiver. It might be a SmallInteger or a Float or some user-defined class.

iii. Keyword Messages

Third, is "keyword" message. Let's start with this Java message-send:

```
x.addKeyValue(a, b)    // Java
```

Here, there are 2 arguments and the message name is "addKeyValue". Here is the Smalltalk equivalent:

```
x addKey: a value: b    "Smalltalk"
```

The message name in Smalltalk is "addKey:value:" and we will notice that it contains the colon character. The colon character is used in keyword message sending and is syntactically significant in understanding the keyword syntax.

Assignment Statements

The assignment statement is familiar. In general, its syntax is:

```
variable := expression
```

Here are a some examples:

```
i := j+1

num := phoneBook at: name

x := y := z := 0
```

In Smalltalk we always have a simple variable name to the left of the assignment operator.

Expressions

An expression includes multiple message sends. In this case expressions are parsed according to a simple order of precedence. Unary messages have highest precedence followed by binary messages followed by keyword messages.

Ex- 3 factorial + 4 factorial between: 10 and: 100

Blocks

A block is a sequence of executable code. Syntactically, a block has a number of statements, surrounded by brackets. For example:

```
[
  x := x + 1.
  y := x * y.
  z := x + y
]
```

Input/ Output

The user’s console is represented as class/ object. Transcripts are available for writing output.

Ex- Transcript. showLine: (‘Here comes an object’, myObjectdescription)

The "If" Statement

Consider the following "if" statement in Java/C++:

```
if (x < y) {      // in Java
  max = y;
  i = j;
} else {
  max = x;
  i = k;
}
```

Here is how the same thing would look in Smalltalk. The "if" statement begins with the boolean expression.

```
   (x < y) ifTrue: [   "in Smalltalk"

     max := y.

     i := j

   ] ifFalse: [

     max := x.

     i := k

   ]
```

The "While" Statement

Here is an example of a "while" loop:

```
   [i < 100] whileTrue: [

     sum := sum + i.

     i := i + 1

   ]
```

The conditional expression must be in brackets here (unlike "if") for the reason that it must be re- evaluated each time around the loop.

Unit-04/Lecture-04

Overview of C++

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Object:

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class:

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

```
class Box

{

  public:

    double length;   // Length of a box

    double breadth;  // Breadth of a box

    double height;   // Height of a box

};
```

Following statements declare two objects of class Box:

```
Box Box1;          // Declare Box1 of type Box

Box Box2;          // Declare Box2 of type Box
```

Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear:

```
#include <iostream.h>

class Box

{

  public:

    double length;   // Length of a box

    double breadth;  // Breadth of a box

    double height;   // Height of a box

};

void main( )

{

  Box Box1;       // Declare Box1 of type Box

  Box Box2;       // Declare Box2 of type Box

  double volume = 0.0;     // Store the volume of a box here

  // box 1 specification

  Box1.height = 5.0;

  Box1.length = 6.0;

  Box1.breadth = 7.0;

  // box 2 specification

  Box2.height = 10.0;

  Box2.length = 12.0;
  Box2.breadth = 13.0;

  // volume of box 1

  volume = Box1.height * Box1.length * Box1.breadth;

  cout << "Volume of Box1 : " << volume <<endl;

  // volume of box 2

  volume = Box2.height * Box2.length * Box2.breadth;

  cout << "Volume of Box2 : " << volume <<endl;

  return 0;

}
```

Constructors

A class can include a special function called its *constructor*, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.
This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void.

```
#include <iostream.h>

class Rectangle {

    int width, height;

  public:

    Rectangle (int,int);

    int area () {return (width*height);}

};

Rectangle::Rectangle (int a, int b) {
```

```
rect area: 12

rectb area: 30
```

```
  width = a;

  height = b;

}

int main () {

  Rectangle rect (3,4);

  Rectangle rectb (5,6);

  cout << "rect area: " << rect.area() << endl;

  cout << "rectb area: " << rectb.area() << endl;

  return 0;

}
```

Destructors

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor:

```
#include <iostream.h>

 class Line

{

  public:

     void setLength( double len );

     double getLength( void );
```

```
      Line();   // This is the constructor declaration

      ~Line();  // This is the destructor: declaration

   private:

      double length;

};

 Line::Line (void)

{ cout << "Object is being created" << endl;

}

Line::~Line(void)

{

    cout << "Object is being deleted" << endl;

}

void Line::setLength( double len )

{

    length = len;

}

double Line::getLength( void )

{

    return length;

}

// Main function for the program

int main( )

{
   Line line;

   // set line length

   line.setLength(6.0);

   cout << "Length of line : " << line.getLength() <<endl;

  return 0;

}
```

Abstraction:

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation:

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance:

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Single Inheritance is method in which a derived class has only one base class.

Example:

```
#include <iostream.h>

ClassValue
{
protected:
intval;
public:
void  set_values  (int  a)
{                      val=a;}
};

class Cube: publicValue
{
public:
intcube()
{ return  (val*val*val); }
};      int      main      ()
{
Cubecub;
cub.set_values(5);
cout << "The Cube of 5
is::"  <<  cub.cube()  <<
endl;
return0;
}
```

Polymorphism:

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading:

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

```
#include <iostream>

class printData
```

```
{
  public:
    void print(int i) {
      cout << "Printing int: " << i << endl;
    }
     void print(double  f) {
      cout << "Printing float: " << f << endl;
    }
     void print(char* c) {
      cout << "Printing character: " << c << endl;
    }
};
int main(void)
{
  printData pd;
  pd.print(5);
  pd.print(500.263);
  pd.print("Hello C++");
  return 0;
}
```

Unit-04/Lecture-05

Overview of JAVA

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995.

Java is:

- **Object Oriented:** In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform independent:** Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
- **Simple:** Java is designed to be easy to learn.
- **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architectural-neutral: Java** compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.
- **Portable:** Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable.
- **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
- **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
- **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed:** Java is designed for the distributed environment of the internet.
- **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Constructor in JAVA

Constructor is a special type of method that is used to initialize the object. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

```
  class Student{

    int id;

    String name;



    Student(int i,String n){

    id = i;

    name = n;

    }

    void display(){System.out.println(id+" "+name);}



    public static void main(String args[]){

    Student4 s1 = new Student4(111,"Karan");

    Student4 s2 = new Student4(222,"Aryan");

    s1.display();

    s2.display();

   }

  }
```

Inheritance in Java

In Java, classes can be derived from other classes, in order to create more complex relationships. A class that is derived from another class is called *subclass* and inherits all fields and methods of its *super class*. In Java, only single inheritance is allowed and thus, every class can have at most one direct super class. A class can be derived from another class that is derived from another class and so on.

```
class A {

                    public A() {
```

```
                    System.out.println("New A");
                  }
                }
class B extends A {
                  public B() {
                    super();
                    System.out.println("New B");
                  }
                }
```

Method Overloading

1. To call an overloaded method in Java, it is must to use the type and/or number of arguments to determine which version of the overloaded method to actually call.
2. Overloaded methods may have different return types; the return type alone is insufficient to distinguish two versions of a method. .
3. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
4. It allows the user to achieve compile time polymorphism.
5. An overloaded method can throw different exceptions.
6. It can have different access modifiers.

```
class Overload
{
  void demo (int a)
  {
    System.out.println ("a: " + a);
  }
```

```
   void demo (int a, int b)

   {

     System.out.println ("a and b: " + a + "," + b);

   }

   double demo(double a) {

     System.out.println("double a: " + a);

     return a*a;

   }

}

class MethodOverloading

{

   public static void main (String args [])

   {

      Overload Obj = new Overload();

      double result;

      Obj .demo(10);

      Obj .demo(10, 20);

      result = Obj .demo(5.5);

      System.out.println("O/P : " + result);

   }

}
```

C

MAIN FEATURES OF C#

1. SIMPLE

1. Pointers are missing in C#.
2. Unsafe operations such as direct memory manipulation are not allowed.
3. In C# there is no usage of "::" or "->" operators.
4. Since it's on .NET, it inherits the features of automatic memory management and garbage collection.
5. Varying ranges of the primitive types like Integer, Floats etc.
6. Integer values of 0 and 1 are no longer accepted as Boolean values. Boolean values are pure true or false values in C# so no more errors of "="operator and "=="operator.
7. "==" is used for comparison operation and "=" is used for assignment operation.

2. MODERN

1. C# has been based according to the current trend and is very powerful and simple for building interoperable, scalable, robust applications.
2. C# includes built in support to turn any component into a web service that can be invoked over the internet from any application running on any platform.

3. OBJECT ORIENTED

1. C# supports Data Encapsulation, inheritance, polymorphism, interfaces.
2. (int,float, double) are not objects in java but C# has introduces structures(structs) which enable the primitive types to become objects.

int i=1;
string a=i Tostring(); //conversion (or) Boxing

4. TYPE SAFE

1. In C# we cannot perform unsafe casts like convert double to a Boolean.
2. Value types (primitive types) are initialized to zeros and reference types (objects and classes) are initialized to null by the compiler automatically.
3. Arrays are zero bases indexed and are bound checked.
4. Overflow of types can be checked.

5. INTEROPERABILITY

1. C# includes native support for the COM and windows based applications.
2. Allowing restricted use of native pointers.
3. Users no longer have to explicitly implement the unknown and other COM interfaces, those features are built in.
4. C# allows the users to use pointers as unsafe code blocks to manipulate your old code.
5. Components from VB NET and other managed code languages and directly be used in C#.

Objects

Objects are basic building blocks of a C# OOP program. An object is a combination of data and methods. The data and the methods are called *members* of an object. In an OOP program, we create objects. These objects communicate together through methods. Each object can receive messages, send messages and process data.

There are two steps in creating an object. First, we define a class. A *class* is a template for an object. It is a blueprint which describes the state and behavior that the objects of the class all share. A class can be used to create many objects. Objects created at runtime from a class are called *instances* of that particular class.

```
using System;
public class Being {}
public class Objects
{
   static void Main()
   {
      Being b = new Being();
      Console.WriteLine(b);
   }
}
```

Example of C# program

```
using System;

using System.Collections.Generic;

using System.Text;

namespace ConsoleApplication1

{
    class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("Hello, world!");

            Console.ReadLine();

        }

    }

}
```

Access Specifiers

Access modifiers set the visibility of methods and member fields. C# has four access modifiers: public, protected, private and internal. The public members can be accessed from anywhere. The protected members can be accessed only within the class itself and by inherited and parent classes. The private members are limited to the containing type, e.g. only within its class or interface. The internal members may be accessed from within the same assembly (exe or DLL).

Keyword	**Description**
public	Public class is visible in the current and referencing assembly.
private	Visible inside current class.
protected	Visible inside current and derived class.
Internal	Visible inside containing assembly.
Internal protected	Visible inside containing assembly and descendent of the current class.

```
using System;

public class Person

{

    public string name;
    private int age;

    public int GetAge()

    {

        return this.age;

    }

    public void SetAge(int age)

    {

        this.age = age;

    }

}

public class AccessModifiers

{

    static void Main()

    {

        Person p = new Person();

        p.name = "Jane";


        p.SetAge(17);

        Console.WriteLine("{0} is {1} years old",

            p.name, p.GetAge());

    }

}
```

The constructor

A constructor is a special kind of a method. It is automatically called when the object is created. Constructors do not return values. The purpose of the constructor is to initiate the state of an object. Constructors have the same name as the class. The constructors are methods, so they can be overloaded too.

Constructors cannot be inherited. They are called in the order of inheritance. If we do not write any constructor for a class, C# provides an implicit default constructor. If we provide any kind of a constructor, then a default is not supplied.

```
using System;

public class Being

{

   public Being()

   {

      Console.WriteLine("Being is created");

   }

 public Being(string being)

   {

      Console.WriteLine("Being {0} is created", being);

  }

}



public class Constructor

{

   static void Main()
```

```
    {
        new Being();
        new Being("Tom");
    }
}
```

Unit-04/Lecture-07

PHP

PHP is an acronym for "PHP: Hypertext Pre-processor". PHP is a widely-used, open source scripting language. PHP scripts are executed on the server. PHP is free to download and use. PHP files can contain text, HTML, CSS, JavaScript, and PHP code. PHP code is executed on the server, and the result is returned to the browser as plain HTML. PHP files have extension ".php" PHP can generate dynamic page content. PHP runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.) PHP is compatible with almost all servers used today (Apache, IIS, etc.) .PHP supports a wide range of databases

Defining PHP Classes:

The general form for defining a new class in PHP is as follows:

```
<?php

class phpClass{

  var $var1;

  var $var2 = "constant string";

  function myfunc ($arg1, $arg2) {

    [..]

  }

  [..]

}

?>
```

Here is the description of each line:

- The special form **class** followed by the name of the class that you want to define.
- A set of braces enclosing any number of variable declarations and function definitions.
- Variable declarations start with the special form **var**, which is followed by a

conventional $ variable name; they may also have an initial assignment to a constant value.

- Function definitions look much like standalone PHP functions but are local to the class and will be used to set and access object data.

Example:

Here is an example which defines a class of Books type:

```
<?php

class  Books{

   /* Member variables */

   var $price;

   var $title;

   /* Member functions */

   function setPrice($par){

     $this->price = $par;

   }

   function getPrice(){

     echo $this->price ."<br/>";

   }

   function setTitle($par){

     $this->title = $par;

   }

   function getTitle(){

     echo $this->title ." <br/>";
```

```
    }

 }

 ?>
```

The variable **$this** is a special variable and it refers to the same object i.e. itself.

Creating Objects in PHP

Once you defined your class, then you can create as many objects as you like of that class type. Following is an example of how to create object using **new** operator.

```
    $physics = new Books;

    $maths = new Books;

    $chemistry = new Books;
```

Here we have created three objects and these objects are independent of each other and they will have their existence separately.

Calling Member Functions

After creating your objects, you will be able to call member functions related to that object. One member function will be able to process member variable of related object only.

Following example shows how to set title and prices for the three books by calling member functions.

```
    $physics->setTitle( "Physics for High School" );

    $chemistry->setTitle( "Advanced Chemistry" );

    $maths->setTitle( "Algebra" );

   $physics->setPrice( 10 );

    $chemistry->setPrice( 15 );

    $maths->setPrice( 7 );
```

Now we call another member functions to get the values set by in above example:

```
$physics->getTitle();

$chemistry->getTitle();

$maths->getTitle();

$physics->getPrice();

$chemistry->getPrice();

$maths->getPrice();
```

This will produce follwoing result:

```
Physics for High School

Advanced Chemistry

Algebra

10

15

7
```

Constructor Functions:

Constructor Functions are special type of functions which are called automatically whenever an object is created. So we take full advantage of this behaviour, by initializing many things through constructor functions.

PHP provides a special function called **__construct ()** to define a constructor. You can pass as many as arguments you like into the constructor function.

Following example will create one constructor for Books class and it will initialize price and title for the book at the time of object creation.

```
 function __construct( $par1, $par2 ){

   $this->price = $par1;

   $this->title = $par2;

 }
```

Now we don't need to call set function separately to set price and title. We can initialize these two member variables at the time of object creation only. Check following example below:

```
   $physics = new Books( "Physics for High School", 10 );

   $maths = new Books ( "Advanced Chemistry", 15 );

   $chemistry = new Books ("Algebra", 7 );

   /* Get those set values */

   $physics->getTitle();

   $chemistry->getTitle();

   $maths->getTitle();

   $physics->getPrice();

   $chemistry->getPrice();

   $maths->getPrice();
```

This will produce following result:

```
  Physics for High School

  Advanced Chemistry

  Algebra

  10
```

```
15

7
```

Destructor:

Like a constructor function we can define a destructor function using function **__destruct ()**. We can release all the resources with-in a destructor.

Inheritance:

PHP class definitions can optionally inherit from a parent class definition by using the extends clause. The syntax is as follows:

```
  class Child extends Parent {

     <definition body>

  }
 class Novel extends Books{

   var publisher;

   function setPublisher($par){

     $this->publisher = $par;

   }

   function getPublisher(){

     echo $this->publisher. "<br />";

   }

 }
```

Unit-04/Lecture-08

Perl

Perl is a stable, cross platform programming language. Though Perl is not officially an acronym but few people used it as **Practical Extraction and Report Language**. It is used for acts in the public and private sectors. Perl is Open *Source* software, licensed under its *Artistic License*, or the *GNU General Public License (GPL)*. Perl was created by Larry Wall.

Perl is a script language, which is compiled each time before running. That Unix knows that it is a Perl script there must be the following header at the top line of every Perl script: **#!/usr/bin/perl** where the path to Perl has to be correct and the line must not exceed 32 characters.

An **object** is a data structure that bundles together data and subroutines which operate on that data. An object's data is called **attributes**, and its subroutines are called **methods**

- A class within Perl is a package that contains the corresponding methods required to create and manipulate objects.
- A method within Perl is a subroutine, defined with the package. The first argument to the method is an object reference or a package name, depending on whether the method affects the current object or the class.

Perl provides a **bless( )** function which is used to return a reference and which becomes an object.

Defining a Class

In Perl, a class is corresponds to a Package. To create a class in Perl, we first build a package. A package is a self-contained unit of user-defined variables and subroutines, which can be re-used over and over again. They provide a separate namespace within a Perl program that keeps subroutines and variables from conflicting with those in other packages.

To declare a class named Person in Perl we do:

```
package Person;
```

When creating an object, you need to supply a constructor. This is a subroutine within a package that returns an object reference. The object reference is created by blessing a reference to the package's class. For example:

```
package Person;

sub new

{

   my $class = shift;

   my $self = {

      _firstName => shift,

      _lastName  => shift,

      _ssn       => shift,

   };

   # Print all the values just for clarification.

   print "First Name is $self->{_firstName}\n";

   print "Last Name is $self->{_lastName}\n";

   print "SSN is $self->{_ssn}\n";

   bless $self, $class;

   return $self;

}
```

Every method of a class passes first argument as class name. So in the above example class name would be "Person".

To create an Object

```
 $object = new Person( "Mohammad", "Saleem", 23234345);
```

If we don't want to assign any value to any class variable. For example

```
  package Person;

  sub new

  {

      my $class = shift;

      my $self = {};

      bless $self, $class;

      return $self;

  }
```

Lets define a helper method to get person first name:

```
 sub getFirstName {

     return $self->{_firstName};

 }
```

Another helper function to set person first name:

```
 sub setFirstName {

     my ( $self, $firstName ) = @_;

     $self->{_firstName} = $firstName if defined($firstName);

     return $self->{_firstName};

 }

#!/usr/bin/perl

package Person;
```

```
sub new

{

   my $class = shift;

   my $self = {

      _firstName => shift,

      _lastName  => shift,

      _ssn       => shift,

   };

   # Print all the values just for clarification.

   print "First Name is $self->{_firstName}\n";

   print "Last Name is $self->{_lastName}\n";

   print "SSN is $self->{_ssn}\n";

   bless $self, $class;

   return $self;

}

sub setFirstName {

   my ( $self, $firstName ) = @_;

   $self->{_firstName} = $firstName if defined($firstName);

   return $self->{_firstName};

}

sub getFirstName {

   my( $self ) = @_;

   return $self->{_firstName};
};
```

Unit-04/Lecture-09

Semaphores

Semaphore is a synchronization tool. semaphore is a value that indicates the status of common resourses. A semaphore, in its most basic form, is a protected integer variable that can facilitate and restrict access to shared resources in a multi-processing environment. The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphores represent multiple resources, while binary semaphores, as the name implies, represents two possible states (generally 0 or 1; locked or unlocked). Semaphores were invented by the late Edsger Dijkstra.

Semaphores can be looked at as a representation of a limited number of resources, like seating capacity at a restaurant. If a restaurant has a capacity of 50 people and nobody is there, the semaphore would be initialized to 50. As each person arrives at the restaurant, they cause the seating capacity to decrease, so the semaphore in turn is decremented. When the maximum capacity is reached, the semaphore will be at zero, and nobody else will be able to enter the restaurant. Instead the hopeful restaurant goers must wait until someone is done with the resource, or in this analogy, done eating. When a patron leaves, the semaphore is incremented and the resource becomes available again.

A semaphore can only be accessed using the following operations: **wait()** and **signal()**. **wait()** is called when a process wants access to a resource. This would be equivalent to the arriving customer trying to get an open table. If there is an open table, or the semaphore is greater than zero, then he can take that resource and sit at the table. If there is no open table and the semaphore is zero, that process must wait until it becomes available. **signal()** is called when a process is done using a resource, or when the patron is finished with his meal. The following is an implementation of this **counting semaphore** (where the value can be greater than 1):

```
wait(Semaphore s){

   while (s==0);    /* wait until s>0 */

   s=s-1;

}
```

```
signal(Semaphore s){

   s=s+1;

}

Init(Semaphore s , Int v){

   s=v;

}
```

If there is only one count of a resource, a **binary semaphore** is used which can only have the values of 0 or 1. They are often used as **mutex locks**. Here is an implementation of mutual-exclusion using binary semaphores:

```
do

{

   Wait;

   // critical section

   Signal;

   // remainder section

} while (1);
```

In the Producer-Consumer problem, semaphores are used for two purposes:

- mutual exclusion and
- Synchronization.

In the following example there are three semaphores. ***Full***, used for counting the number of slots that are full; ***empty***, used for counting the number of slots that are empty; and ***mutex***, used to enforce mutual exclusion.

```
 BufferSize = 3;

 semaphore mutex = 1;          // Controls access to critical section

 semaphore empty = BufferSize;    // counts number of empty buffer slots
```

```
semaphore full = 0;            // counts number of full buffer slots


Producer()

{

 int widget;

 while (TRUE) {              // loop forever

   make_new(widget);           // create a new widget to put in the buffer

   down(&empty);              // decrement the empty semaphore

   down(&mutex);              // enter critical section

   put_item(widget);           // put widget in buffer

   up(&mutex);                // leave critical section

   up(&full);               // increment the full semaphore

   }

}

Consumer() {

 int widget;

 while (TRUE) {              // loop forever

   down(&full);               // decrement the full semaphore

   down(&mutex);              // enter critical section

   remove_item(widget);        // take a widget from the buffer

   up(&mutex);                // leave critical section

   up(&empty);                // increment the empty semaphore

   consume_item(widget);       // consume the item
```

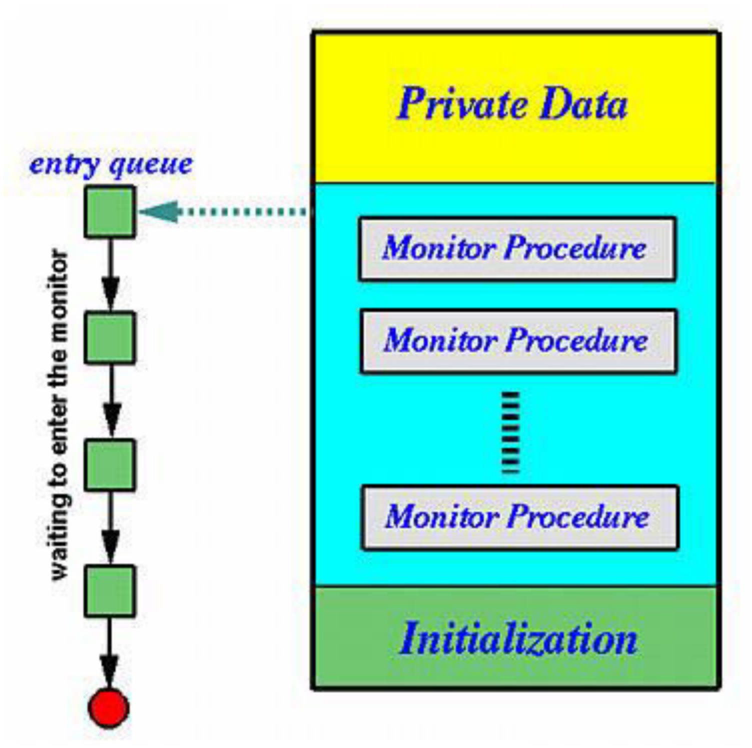
```
    }

}
```

Unit-04/Lecture-10

Monitors

A monitor has *four* components as shown below: initialization, private data, monitor procedures, and monitor entry queue. The *initialization* component contains the code that is used exactly once when the monitor is created, the *private data* section contains all private data, including private procedures that can only be used *within* the monitor. Thus, these private items are not visible from outside of the monitor. The *monitor procedures* are procedures that can be called from outside of the monitor. The *monitor entry queue* contains all threads that called monitor procedures but have not been granted permissions.



Mutual Exclusion is automatically provided by the monitor implementation. If a process calls monitor methods, but another process is already executing inside the monitor, the calling thread must wait outside the monitor.

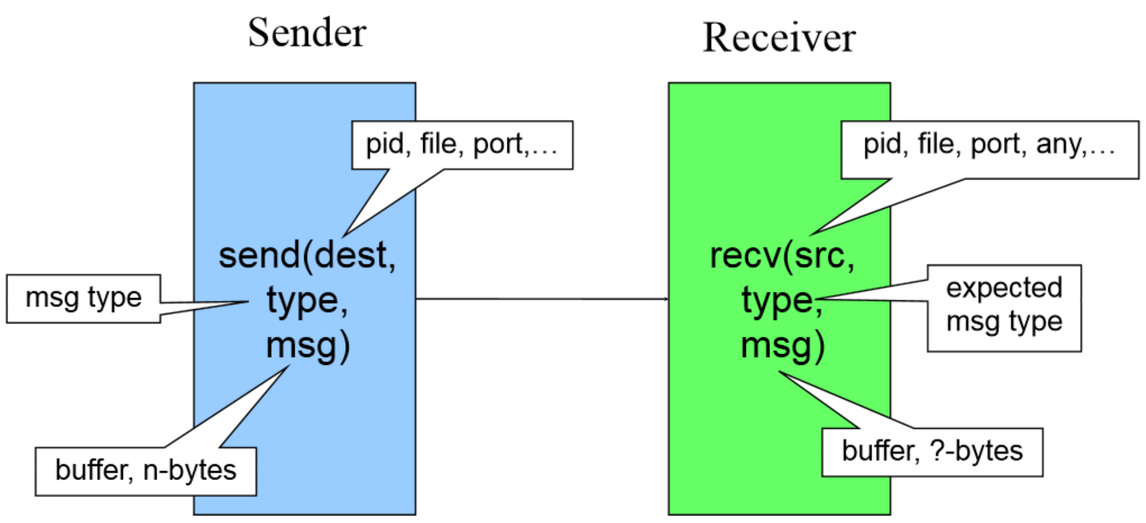
Synchronous Message Passing

Move data between processes

- Sender: when data is ready, send it to the receiver process
- Receiver: when the data has arrived and when the receive process is ready to take the data, move the data to the destination data structure

Synchronization

- Sender: signal the receiver process that a particular event happens
- Receiver: block until the event has happened.



Asynchronous Message Passing

If a process sends a message and continue executing without waiting for the message to be received then the communication is termed asynchronous.

- Send operations are non blocking.
- A sending process can get arbitrarily for ahead of a receiving process.
- Message delivery is not guaranteed if failures can occur.
- Since channels can contain an unbounded number of messages, messages have to be buffered.
- The sending process has no ways of knowing if the message was ever received unless

the receiving process sends a reply.

- It is hard to detect when failures have occurred.
- Buffer space is finite- if too many messages are sent either the program will crash; the buffer will overflow with loss of messages or send operation will block.

Unit-04/Lecture-11

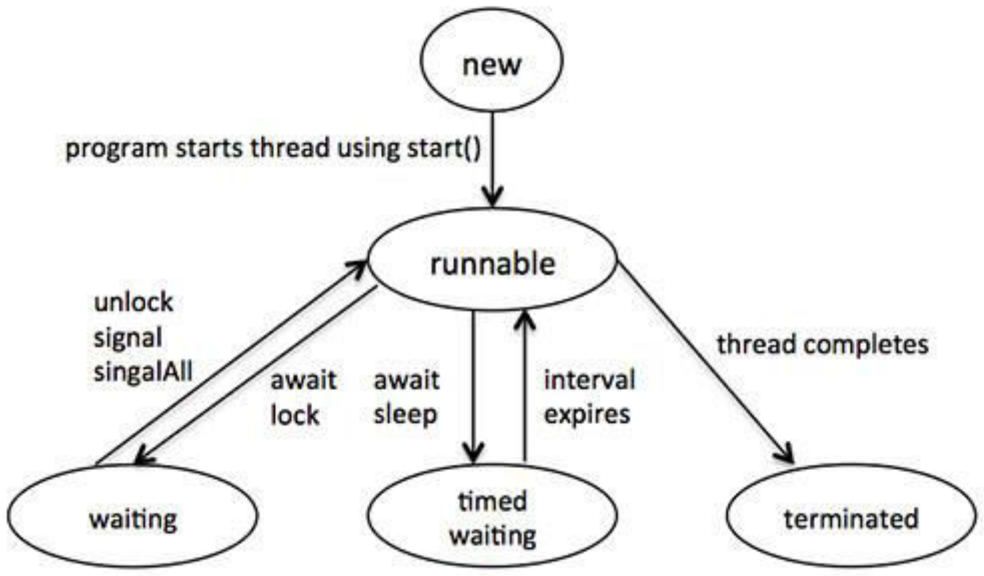
Java Threads

Java is m*ultithreaded programming language* which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.

Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Life Cycle of a Thread:A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in these state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing **Runnable** interface. You will need to follow three basic steps:

Step 1:

As a first step we need to implement a run() method provided by **Runnable** interface. This method provides entry point for the thread and we will put our complete business logic inside this method. Following is simple syntax of run() method:

public void run( )

Step 2:

At second step you will instantiate a **Thread** object using the following constructor:

Thread(Runnable threadObj, String threadName);

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method. Following is simple syntax of start() method:

void start( );

Example:

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable {

  private Thread t;

  private String threadName;

   RunnableDemo( String name){

     threadName = name;

     System.out.println("Creating " +  threadName );

  }

  public void run() {

    System.out.println("Running " +  threadName );

    try {

      for(int i = 4; i > 0; i--) {

        System.out.println("Thread: " + threadName + ", " + i);

        // Let the thread sleep for a while.

        Thread.sleep(50);

      }

   } catch (InterruptedException e) {

      System.out.println("Thread " +  threadName + " interrupted.");

   }

   System.out.println("Thread " +  threadName + " exiting.");

  }

  public void start ()

  {

    System.out.println("Starting " +  threadName );
    if (t == null)

    {

      t = new Thread (this, threadName);

      t.start ();

    }

  }

}

public class TestThread

public static void main(String args[]) {

    RunnableDemo R1 = new RunnableDemo( "Thread-1");

    R1.start();

    RunnableDemo R2 = new RunnableDemo( "Thread-2");

    R2.start();

  }

}
```

This would produce the following result:

```
Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-1

Thread: Thread-1, 4

Running Thread-2
```

```
Thread: Thread-2, 4

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.
```

Unit-05/Lecture-01

Exception Handling

Exception Handling is a programming language construct designed to handle the occurrence of exceptions.

Exceptions are runtime unusual conditions that a program may encounter while executing.

Types of Exceptions

i. Synchronous Exception- The errors that are caused by the events in the control of program are called as synchronous exceptions. Examples are: out of index, overflow.
ii. Asynchronous exception- The errors that are caused by the events beyond the control of program are called as asynchronous exception. Examples are: keyboard interrupts, out of memory.

Exception Propagation:-

If an exception is not handled in the subprogram in which it was raised, this exception is propagated to the subprogram that called it. A handler for the exception is searched for in the calling subprogram. If no handler is found there then the exception is propagated again. This continues until either an exception handler is found, or the highest level of the current task is reached, in which case the task is aborted.

Exception Handler in C++

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

- throw: A program throws an exception when a problem shows up. This is done using a throw keyword.
- catch: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- try: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)

{

  if( b == 0 )

  {

    throw "Division by zero condition!";

  }

  return (a/b);

}
```

Catching Exceptions:

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try

{

  // protected code

}catch( ExceptionName e )

{

 // code to handle ExceptionName exception

}
```

```
#include <iostream>

double division(int a, int b)

{

  if( b == 0 )

  {

    throw "Division by zero condition!";

  }

  return (a/b);

}

int main ()

{

  int x = 50;

  int y = 0;

  double z = 0;

  try {

   z = division(x, y);

   cout << z << endl;

  }catch (const char* msg) {

   cout << msg << endl;

  }

  return 0;

}
```

Multiple Catch Statements

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is excuted. After executing the handler; the control goes to the first statement after the last catch block for that try. When no match is found, the program is terminated.

```
#include<iostream.h>

#include<conio.h>

void test(int x)

{

  try

  {

         if(x>0)

           throw x;

     else

           throw 'x';

  }

  catch(int x)

  {

         cout<<"Catch a integer and that integer is:"<<x;

  }

  catch(char x)

  {
```

```
            cout<<"Catch a character and that character is:"<<x;
  }
}
void main()
{
  clrscr();
  cout<<"Testing multiple catches\n:";
  test(10);
  test(0);
  getch();
}
```

Output:

```
Testing multiple catches
Catch a integer and that integer is: 10
Catch a character and that character is: x
```

Unit-05/Lecture-03

Exception Handling in Java

Exceptions are errors which occur when the program is executing. Consider the Java program below which divides two integers.

```
import java.util.Scanner;

 class Division {

  public static void main(String[] args) {

  int a, b, result;

  Scanner input = new Scanner(System.in);

  System.out.println("Input two integers");

  a = input.nextInt();

  b = input.nextInt();

  result = a / b;

  System.out.println("Result = " + result);

  }

}
```

Java provides a powerful way to handle such exceptions which is known as exception handling. In it we write vulnerable code i.e. code which can throw exception in a separate block called as **try** block and exception handling code in another block called **catch** block. Following modified code handles the exception.

```
class Division {

  public static void main(String[] args) {

  int a, b, result;
  Scanner input = new Scanner(System.in);

  System.out.println("Input two integers");

  a = input.nextInt();

  b = input.nextInt();

  // try block

  try {

    result  = a / b;

    System.out.println("Result = " + result);

  }

   // catch block

  catch (ArithmeticException e) {

    System.out.println("Exception caught: Division by zero.");

  }

  }

}
```

Whenever an exception is caught corresponding catch block is executed, For example above code catches Arithmetic Exception only. If some other kind of exception is thrown it will not be caught so it's the programmer work to take care of all exceptions as in our try block we are performing arithmetic so we are capturing only arithmetic exceptions. A simple way to capture any exception is to use an object of Exception class as other classes inherit Exception class, see another example below:

```
class Exceptions {

  public static void main(String[] args) {

 String languages[] = { "C", "C++", "Java", "Perl", "Python" };

  try {
   for (int c = 1; c <= 5; c++) {

     System.out.println(languages[c]);

   }

  }

  catch (Exception e) {

   System.out.println(e);

  }

  }

}
```

Output of program:

```
C++

Java

Perl

Python

java.lang.ArrayIndexOutOfBoundsException: 5
```

Unit-05/Lecture-04

LOGIC PROGRAMMING

A logic program consists of a set of axioms and a goal statement. The rules of inference are applied to determine whether the axioms are sufficient to ensure the truth of the goal statement. The execution of a logic program corresponds to the construction of a proof of the goal statement from the axioms.

In the logic programming model the programmer is responsible for specifying the basic logical relationships and does not specify the manner in which the inference rules are applied. Thus

Logic + Control = Algorithms

Logic programming is based on tuples. Predicates are abstractions and generalization of the data type of tuples.

Languages used for logic programming are called Declarative languages becaue programs written using them consists of declarations rather than assignments and control flow statements. These declarations are actually statements or propositions in symbolic logic.

Declarative semantics is simpler than the semantics of imperative languages. For example, the meaning of a given proposition in a logic programming language can be determined from the statement itself. In an imperative language, the semantics of a simple assignment statement requires examination of local declarations, knowledge of scoping rule of the language, data type of variable etc.

Facts, Predicates and Atoms

Facts are a means of stating that a relationship holds between objects.

father(bill,mary).

plus(2,3,5).

...

This fact states that the relation father holds between bill and mary. Another name for a relationship is predicate.

Queries

A query is the means of retrieving information from a logic program.

?- father (bill,mary).

?- father (bill, jim).

Introduction to Prolog

A prolog **term** is a constantan, a variable, or a structure. A **fact statement** is simply a proposition that is assumed to be true.

Ex. female(Shelly). – note: every statement is terminated by a period.

Rule statements state rules of implication between propositions.

Ex. parent(X, Y): - mother(X, Y). This means that if x is a mother of y then x is the parent of y.

A goal statement is one that which requests an answer; the syntactic form of fact statements and goal statements are identical

Ex. father(X, mike). This asks the question “who is the father of mike?”

parent(sue, bill).

parent(sue, james).

parent(sue, edith).

parent(fred, bill).

parent(fred, james).

parent(arthur, edith).

parent(mary, kylie).

parent(mary, jason).

parent(mary, matilda).

parent(james, kylie).

parent(james, jason).

parent(james, matilda).

parent(edith, david).

parent(william, david).

ancestor(X, Y) :-parent(X, Y).

ancestor(X, Y) :-parent(X, Z), ancestor(Z, Y).

An example query would be? - ancestor (fred, david), *"Is fred the ancestor of david"* which would return a **no**.

Unit-05/Lecture-05

INTRODUCTION TO FUNCTIONAL PROGRAMMING

The functional programming paradigm is based on mathematical functions. LISP is a purely functional language. ML is a strongly typed functional language with more conventional syntax than LISP.

Mathematical Functions

A mathematical function is a mapping of members of one set called the domain set to another set called the range set.

A function definition specifies the domain and range sets along with the mapping. The mapping is described by an expression; functions are always applied to a particular element of the domain set and a function return an element of the range set.

1. Simple Functions- Function definitions are often written as a function name, followed by a list of parameters in parenthesis followed by the mapping expression

   Ex- Cube (x) = x*x*x

   Where x is real number.

In this definition the domain and range sets are real numbers. The parameter 'x' can represent any member of the domain set. The range element is obtained by evaluating the function mapping expression with the domain element substituted for the occurrences of the parameter.

Ex- Cube (2.0) returns 8.0

2. Lambda Notation- In functional programming lambda notation provides a method for defining nameless functions. A lambda expression specifies the parameter and the mapping of a function. Ex- (ƛ(x) x * x * x)(2) which results in the value 8.
3. Functional Forms- A higher order function or function form is one that either takes functions as parameters or returns a function as its result or both.

Types of Functional Forms

1. Functional composition- It has two functional parameters and returns a

function whose value is the first actual parameter function applied to the result of the second. Functional composition is written as an expression1 using o as an operator.

H= f o g

Ex- if f(x) = x+2

g(x)= 3*x

then h is defined as h(x)= f(g(x)) or

h(x)= (3 * x) + 2

2. Construction- Construction is a functional form that takes a list of functions as parameters. When applied to an argument, construction applies to each of its functional parameters to that argument and collects the result in a list or sequence. A construction is denoted by placing the functions in brackets as in [f, g].

Ex- g(x) = x*x

h(x) = 2 * x

f(x) = x / 2

Then [g, h, f] (4) yields (16, 8, 2)

3. Apply to all- Apply to all is a functional form that takes a single function as a parameter. If applied to a list of arguments apply to all applies to its functional parameter to each of the values in the list argument and collects the result in a list or sequence. Apply to all is denoted by α.

Ex- h(x) = x * x

Then α (h (2, 3, 4)) yields (4, 9, 16).

Unit-05/Lecture-06

Fundamentals of Functional Programming

1. TYPES:- Values and Operations

“A type consists of set of elements called values together with a set of functions called operations”.

Basic Types- A type is basic if its values are atomic- this is, if the values are treated as whole elements with no internal structure. For Ex- the Boolean values in the set {true, false} are basic values.

Operation on basic values- Basic values have no internal structure so the only operation defined for all basic types is a comparison for equality; for example the equality 2 = 2 is true and the equality 2 != 2 is false.

Product of types- The product A * B of two types A and B consists of ordered pairs written as (a, b) where a is value of type A and b is value of type B. Thus

- (1, “one”) is a pair consisting of the integer 1 and the string “one”
- A product of n types A1 * A2.... * An consists of tuples written as (a1, a2... an)

Operations on Pairs- Associated with pairs are operations called projection functions to extract the first and second elements from a pair.

Projection can be defined as

```
fun first (x, y) = x;

fun second( x, y) = y;
```

List of Elements- A list is a finite length sequence of elements. The type A list consists of all list of elements, where each element belongs to type A.

For Ex- int list

consists of all lists of integers.

List elements is written between brackets [and] separated by commas

- The list [1, 2, 3] is a list of three integers 1, 2, 3.
- The list [“red”, “white”, “blue”] is a list of strings.

2. FUNCTION DECLARATION

A function declaration has three parts:

I. The name of the declared function
II. The parameters of the function.
III. A rule for computing a result from the parameters.

- Syntax of Function Declaration

fun <name> <formal- parameter> = <body>;

Paranthesis around the formal parameter is optional.

Ex- fun successor n = n+1;

The keyword fun marks the beginning of function declaration, <name> is the function name, <formal parameter> is a parameter name and <body> is an expression to be evaluated.

The function is called as follows:

<name><actual parameter>

Where <name> is the name of function, <actual- parameter> is an expression corresponding to the parameter name in the declaration of the function.

Thus, successor (2 + 3)

Is the application of successor function to the actual parameter (2 + 3).

- Recursive Functions

A function f is recursive if its body contains an application of f.

Ex- The following function len counts the number of elements in a list.

Fun len(x) = if null (x) then 0 else 1 + len(tl(x))

The function is recursive because the body contains an application of len; it is len (tl(x))

Unit-04/Lecture-07

FOURTH GENERATION LANGUAGES

1 GL was machine language or the level of instructions and data that the processor is actually given to work on (string of 0s and 1s)

2 GL is assembly language. An assembler converts the assembly language statements into machine language.

Ex- ADD 12, 8

3 GL is a high level language such as C, C++, and Java. A compiler converts the statements of a specific high level programming language into machine language. A 3GL requires a considerable amount of programming knowledge.

4 GL is designed to be closer to natural language than a 3GL language. Languages for accessing databases are often described as 4GLs. The advantage of using 4GL is that we can write faster code than a 3GL.

For Ex- If we want to access records of employee having name 'Smith 'from emp table then we have to write just the query

Select * from emp where name= 'Smith'

4 GL

The languages like Oracle, VB++, VC++, and SQL etc are called as 4 GL languages. Most 4G languages are used to access databases. They allow the programmer to define 'what' is required without telling the computer 'how' to implement it.

Features of 4 GL

- Ease of Use- As the syntax of 4GL is closer to human languages, it is easy to learn. Additionally due to the nonprocedural nature of many of the languages the techniques for accomplishing things are also simple, while the results are fast.
- Limited range of Functions- 4GLs are typically designed for a limited set of functions or specific applications. Because of this, the product becomes easier to use than a full programming language. For Ex- Oracle is used to

design database while VB is used to design front end.

- Default Options- A user of a 4GL is not required to specify all the parameters. Instead a compiler or interpreter is capable of making intelligent assumptions. 4 GLs provides default options is user does not make a selection.

Ex- if 101 > 100 then 101 – 10 else f (f (101 + 11))

= 101 – 10

=91

- Outermost Evaluation-

f (100) = if 100 > 100 then 100 -10 else f (f (100 + 11))

=f(f(100 + 11)) ---1

= if f(100 + 11) > 100 then f(100 + 11)   10

Else f (f(f(100 +11) + 11)

Evaluation of f (100 + 11)

f(100 + 11) = if 100 + 11 > 100 then 100 +11 -10

else f(f(100+11)+11)

=if 111 > 100 then 100 + 11 – 10

else f(f(100 + 11) +11)

= 100 + 11 – 10

= 101

By putting the value of f(100 + 11) in eq 1

f(100) = if 101 > 100 then 101 – 10

else f(f(101 + 11))

= 101 – 10

=91

- Short Circuit Evaluation- The operator “andalso” and “orelse” in ML performs short circuit evaluation of Boolean expressions in which the right hand operator is evaluated only if it has to be

Ex- E andalso F

Is false if E is false, it is true if both E and F are true. The evaluation of E andalso F proceeds from left to right with F being evaluated only if E is true. Similarly the value of expression E orelse F is true if E evaluates to true. If E is false then E is evaluated. In C operators & and | performs short circuit evaluation of Boolean expression.