CSC 307 STRUCTURED AND OBJECT ORIENTED PROGRAMMING

(LECTURE NOTES)

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines.

Examples of structured programming languages include; ALGOL, PASCAL, C, FORTRAN, COBOL, BASIC, Delphi, Simula, CPL, PL/M, Modula, Oberon, Ada.

Major concepts in Structured Programming
Structured programming uses three major concepts:
1. Top-down Analysis
2. Modular Programming
3. Structured Coding

- **Top-down Analysis**
  Software is always designed to perform some kind of analysis. In the world of software, this type of analysis is referred to as a problem. As a result, we must comprehend how to address the issue. The problem is broken down into small pieces with some significance in top-down analysis. Each problem is solved individually, and the steps to do so are clearly stated.

- **Modular Programming**
  While programming, the code is broken down into smaller groups of instructions. These groups are referred to as modules, subprograms, or subroutines. Modular programming is based on an understanding of top-down analysis. It discourages program jumps using 'goto statements, which render the program flow untraceable. Jumps are prohibited, and the modular format is encouraged in structured programming.

- **Coding Structure**
  Structured coding, in contrast to top-down analysis, subdivides modules into smaller units of code in the order of their execution. The control structure is used in structured programming to control the flow of the program, whereas structured coding uses control structure to organise its instructions in definable patterns.

Types of structured programming
Structured programming can be divided into three categories:

Procedural programming
It is a programming paradigm that is based on the idea that programs are sequences of instructions that must be executed. They place a strong emphasis on dividing programs into named sets of instructions known as procedures, which are analogous to functions. A procedure can access and modify global data variables and store local data that is not accessible from outside the procedure's scope. Fortran and ALGOL were two of the first procedural programming languages. The concepts developed in ALGOL are still very relevant and prevalent in modern programming languages.

Principles of procedural programming

→ Programs are made up of instruction sequences. There is only a thin abstraction layer between the code and the machine.

→ Procedures, which are logical blocks made up of groups of instructions, can be called from anywhere in the code.

→ A procedure can take arguments and return results to the caller. Functions can also access and modify variables in the global scope.

→ Procedural languages use block-based control flow rather than goto commands and adhere to structured programming practices.

Object-oriented programming (OOP)

It defines a program as a collection of objects or resources to which commands are sent. An object-oriented language will create a data resource and pass it to processing commands. For example, a procedural programmer may say "Print(object)," whereas an OOP programmer may say "Tell Object to Print."

Model-based programming

Database query languages are the most common example of this. Units of code in database programming are associated with steps in database access and are updated or run when those steps occur. The structure of the code will be determined by the database and database access structure. Reverse Polish Notation (RPN), a math-problem structure that lends itself to efficient solving of complex expressions, is another example of a model-based structure. Quantum computing, which is still in its early stages, is another example of model-based structured programming; the quantum computer requires a specific model to organise steps, and the language simply provides it.

Advantages of Structured Programming:

1. Easier to read and understand
2. User Friendly
3. Easier to Maintain
4. Mainly problem based instead of being machine based
5. Development is easier as it requires less effort and time
6. Easier to Debug
7. Machine-Independent, mostly.

Disadvantages of Structured Programming:

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. Whereas in the case of assembly language, the development takes lesser time as it is fixed for the machine.

Structured Analysis and Structured Design (SA/SD)

Is a diagrammatic notation that is designed to help people understand a system and how its components work? The basic goal of SA/SD is to improve quality and reduce the risk of

system failure. It establishes concrete management specifications and documentation. It focuses on the solidity, reliability, and maintainability of the system.
SA/SD is combined known as SAD and it mainly focuses on the following 3 points:

i. System
ii. Process
iii. Technology

SA/SD involves 2 phases:

i. **Analysis Phase:** It uses Data Flow Diagram, Data Dictionary, State Transition diagram and ER diagram.
ii. **Design Phase:** It uses Structure Chart and Pseudo Code.

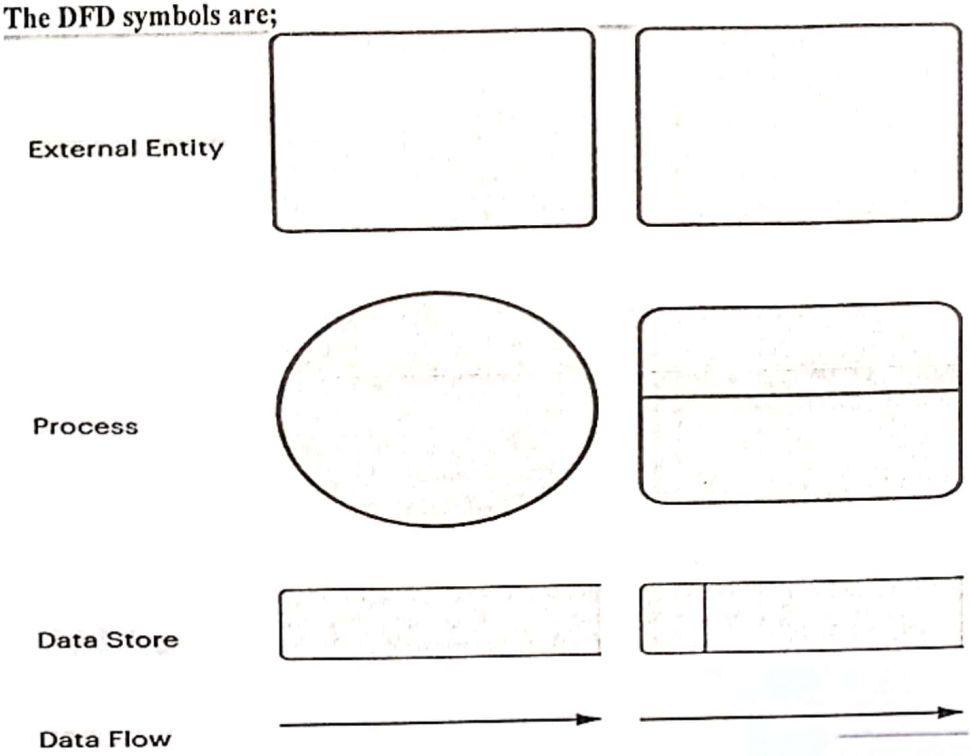
Analysis Phase:
Analysis Phase involves data flow diagram, data dictionary, state transition diagram, and entity-relationship diagram.

1. **Data Flow Diagram:**
   In the data flow diagram, the model describes how the data flows through the system. We can incorporate the Boolean operators and & or link data flow when more than one data flow may be input or output from a process.
   For example, if we have to choose between two paths of a process we can add an operator or and if two data flows are necessary for a process we can add an operator. The input of the process "check-order" needs the credit information and order information whereas the output of the process would be a cash-order or a good-credit-order.

Symbols and Notations Used in DFDs
Using any convention's DFD rules or guidelines, the symbols depict the four components of data flow diagrams.

1. **External entity:** an outside system that sends or receives data, communicating with the system being diagrammed. They are the sources and destinations of information entering or leaving the system. They might be an outside organization or person, a computer system or a business system. They are also known as terminators, sources and sinks or actors. They are typically drawn on the edges of the diagram.
2. **Process:** any process that changes the data, producing an output. It might perform computations, or sort data based on logic, or direct the data flow based on business rules. A short label is used to describe the process, such as "Submit payment."
3. **Data store:** files or repositories that hold information for later use, such as a database table or a membership form. Each data store receives a simple label, such as "Orders."
4. **Data flow:** the route that data takes between the external entities, processes and data stores. It portrays the interface between the other components and is shown with arrows, typically labeled with a short data name, like "Billing details."

The DFD symbols are;



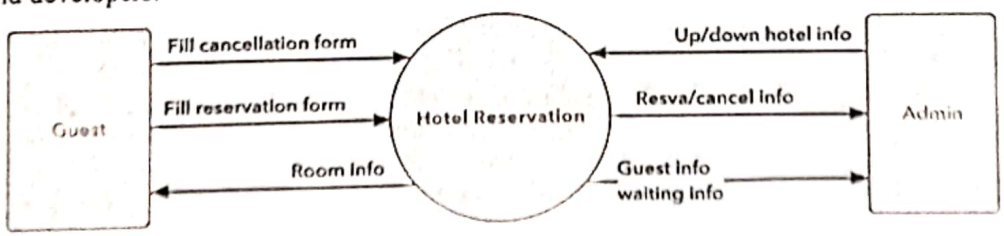
Rules for DFD Design

- Each process should have at least one input and an output.
- Each data store should have at least one data flow in and one data flow out.
- Data stored in a system must go through a process.
- All processes in a DFD go to another process or a data store.

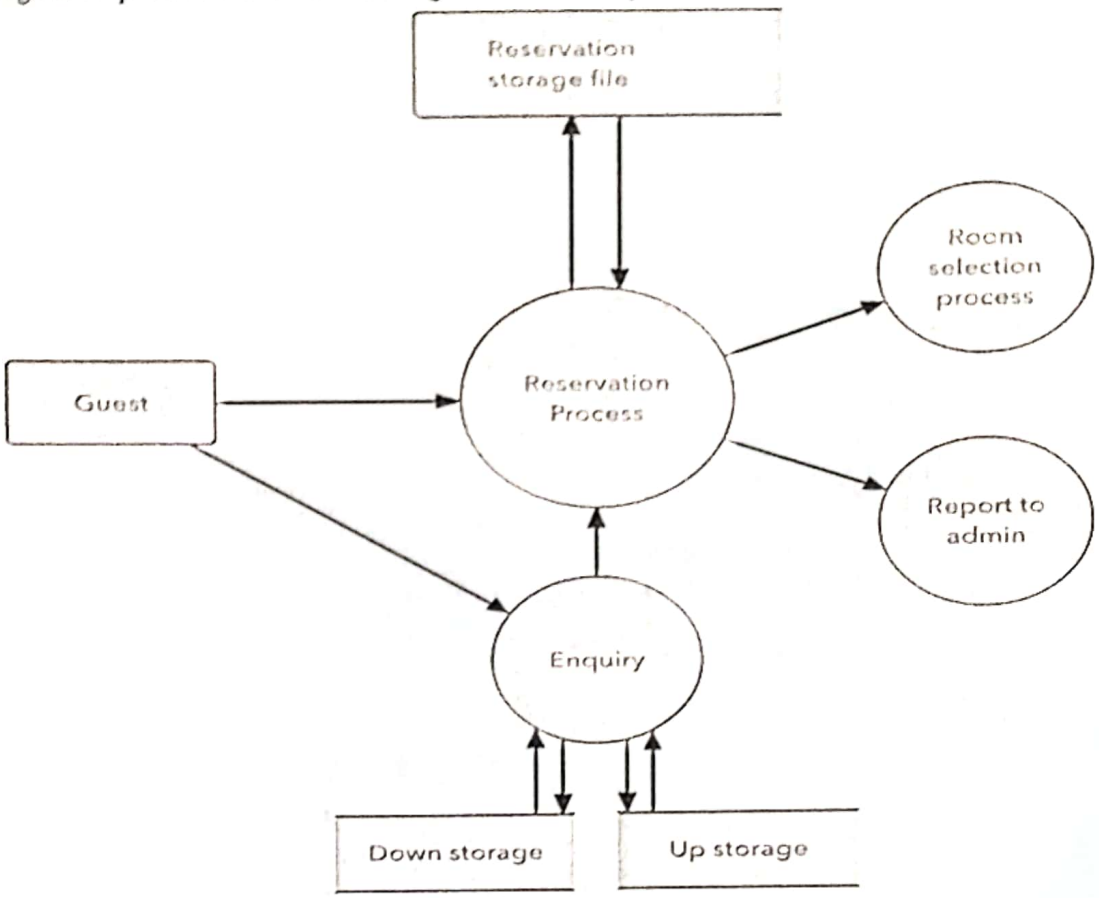
DFD levels and layers

A data flow diagram can dive into progressively more detail by using levels and layers, zeroing in on a particular piece. DFD levels are numbered 0, 1 or 2, and occasionally go to even Level 3 or beyond. The necessary level of detail depends on the scope of what you are trying to accomplish.

DFD Level 0: is also called a Context Diagram. It's a basic overview of the whole system or process being analyzed or modeled. It's designed to be an at-a-glance view, showing the system as a single high-level process, with its relationship to external entities. It should be easily understood by a wide audience, including stakeholders, business analysts, data analysts and developers.



DFD Level 1: provides a more detailed breakout of pieces of the Context Level Diagram. You will highlight the main functions carried out by the system, as you break down the high-level process of the Context Diagram into its sub processes.



2. **Data Dictionary:**
   The content that is not described in the DFD is described in the data dictionary. It defines the data store and relevant meaning. A physical data dictionary for data elements that flow between processes, between entities, and between processes and entities may be included. This would also include descriptions of data elements that flow external to the data stores.
   A logical data dictionary may also be included for each such data element. All system names, whether they are names of entities, types, relations, attributes, or services, should be entered in the dictionary.

Typical attributes of Data Dictionary

Here is a non-exhaustive list of typical items found in a data dictionary for columns or fields:

- Entity or form name or their ID (EntityID or FormID). The group this field belongs to.
- Field name, such as RDBMS field name
- Displayed field title. May default to field name if blank.
- Field type (string, integer, date, etc.)

- Measures such as min and max values, display width, or number of decimal places. Different field types may interpret this differently. An alternative is to have different attributes depending on field type.
- Prompt type, such as drop-down list, combo-box, check-boxes, range, etc.
- Is-required (Boolean) - If 'true', the value cannot be blank, null, or only white-spaces
- Reference table name, if a foreign key. Can be used for validation or selection lists.
- Various event handlers or references to. Example: "on-click", "on-validate", etc.
- Format code.
- Description or synopsis
- Database index characteristics or specification

Data

client_id	name	dob	gender	marital_status	current_address	description
1	Ki Ding	03/02/01	M	Single	Osaka, Japan	-
2	Gu Fing	30/08/99	M	Single	Tokyo, Japan	Certificate for proof of date of birth is yet to be submitted
3	Joe King	02/11/99	M	Married	Nagoya, Japan	-

Data dictionary (Metadata)

	Column	Data type	Field size	Description
1	client_id	int	5	Client's ID
2	name	nvarchar	30	Client's fullname
3	dob	date	8	Date of birth as per client's documents
4	gender	char	2	M – Male, F – Female, NB – Non-binary
5	marital_status	char	30	Marital Status as described by the client
6	current_address	char	300	Current residential address as described by client
7	description	nvarchar	300	Notes

Sample Layout of a Data dictionary

3. State Transition Diagram:

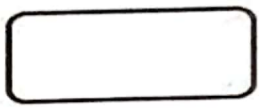
State-transition diagrams describe all of the states that an object can have, the events under which an object changes state (transitions), the conditions that must be fulfilled before the transition will occur (guards), and the activities undertaken during the life of an object (actions). State-transition diagrams are very useful for describing the behavior of individual objects over the full set of use cases that affect those objects. State-transition diagrams are not useful for describing the collaboration between objects that cause the transitions.

You can see the symbols and their description given below;

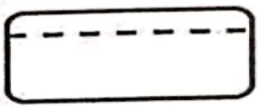
1. **Initial State –**

2. **Final State –**

3. **Simple State –**



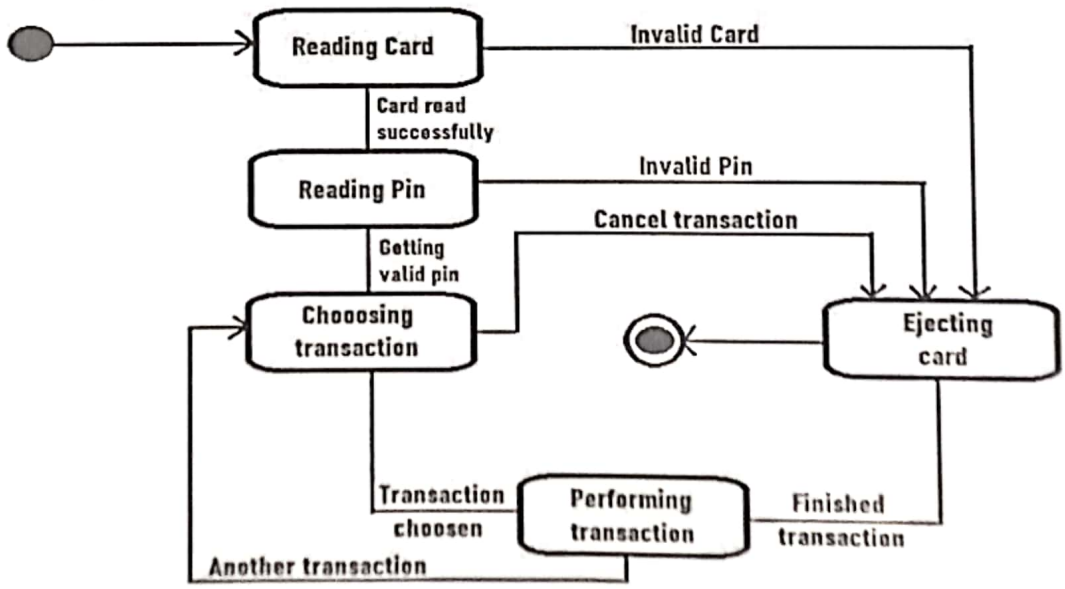
4. **Composite State –**



Type of State	Description
Initial State	In a System, it represents Starting state.
Final State	In a System, it represents Ending state.
Simple State	In a System, it represents a Simple state with no substructure.
Composite State	In a System, it represents a Composite state with two or more parallel or concurrent states out of which only one state will be active at a time and other states will be inactive.

The UML notation for state-transition diagrams is shown below:

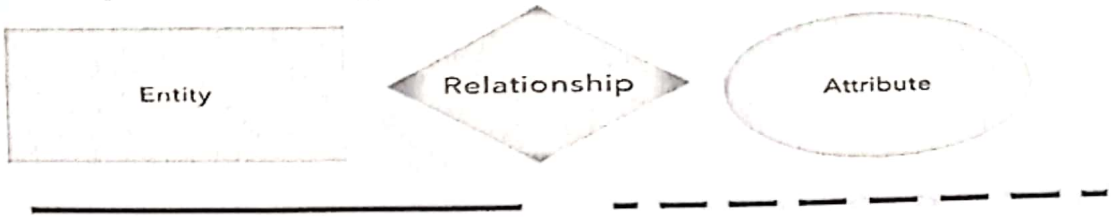
Now let us see the State Transition Diagram of Automated Teller Machine (ATM) System. In this you will see the processing when the customer performs transactions using ATM card



State Transition Diagram for ATM System

4. **Entity Relational Diagram:**
ER diagram specifies the relationship between data store. It is basically used in database design. It basically describes the relationship between different entities.

The components of an ER diagram



mandatory relationship optional relationship

Entity

A definable thing—such as a person, object, concept or event—that can have data stored about it. Think of entities as nouns. Examples: a customer, student, car or product. Typically shown as a rectangle.

Entity type: A group of definable things, such as students or athletes, whereas the entity would be the specific student or athlete. Other examples: customers, cars or products.

Entity set: Same as an entity type, but defined at a particular point in time, such as students enrolled in a class on the first day. Other examples: Customers who purchased last month, cars currently registered in Florida. A related term is instance, in which the specific person or car would be an instance of the entity set.

Entity categories: Entities are categorized as strong, weak or associative. A strong entity can be defined solely by its own attributes, while a weak entity cannot. An associative entity associates entities (or elements) within an entity set.

Entity keys

Refers to an attribute that uniquely defines an entity in an entity set. Entity keys can be super, candidate or primary.

Super key: A set of attributes (one or more) that together define an entity in an entity set.

Candidate key: A minimal super key, meaning it has the least possible number of attributes to still be a super key. An entity set may have more than one candidate key.

Primary key: A candidate key chosen by the database designer to uniquely identify the entity set.

Foreign key: Identifies the relationship between entities.

Relationship

How entities act upon each other or are associated with each other. Think of relationships as verbs. For example, the named student might register for a course. The two entities would be the student and the course, and the relationship depicted is the act of enrolling, connecting the two entities in that way. Relationships are typically shown as diamonds or labels directly on the connecting lines.

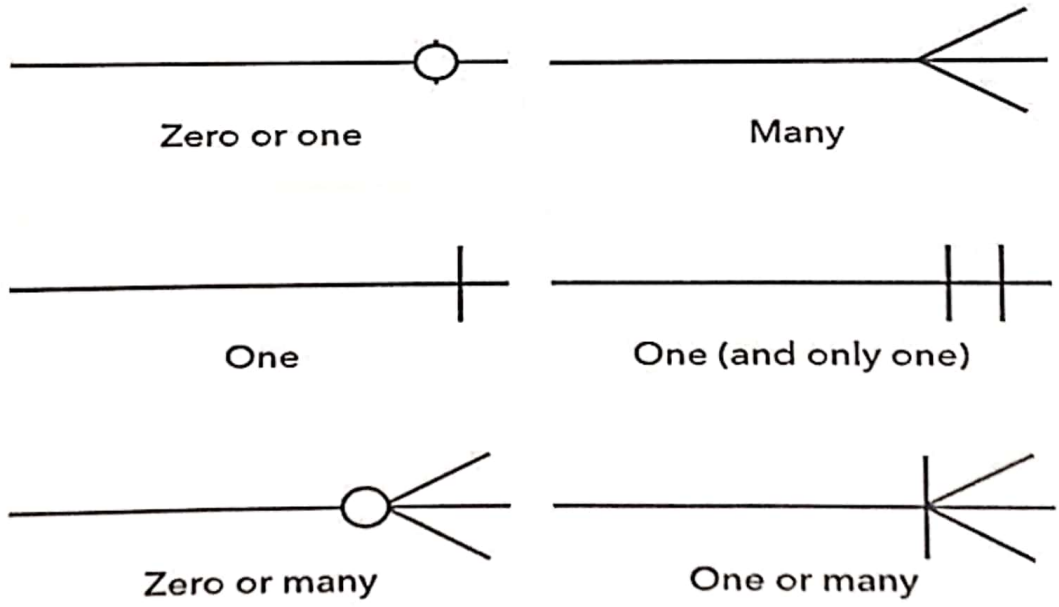
Attribute

A property or characteristic of an entity. Often shown as an oval or circle.
Attribute categories: Attributes are categorized as simple, composite, derived, as well as single-value or multi-value.
Simple: Means the attribute value is atomic and can't be further divided, such as a phone number.
Composite: Sub-attributes spring from an attribute.
Derived: Attributed is calculated or otherwise derived from another attribute, such as age from a birthdate.

Multi-value: More than one attribute value is denoted, such as multiple phone numbers for a person.

Cardinality

Defines the numerical attributes of the relationship between two entities or entity sets. The three main cardinal relationships are one-to-one, one-to-many, and many-many. A one-to-one example would be one student associated with one mailing address. A one-to-many example (or many-to-one, depending on the relationship direction): One student registers for multiple courses, but all those courses have a single line back to that one student. Many-to-many example: Students as a group are associated with multiple faculty members, and faculty members in turn are associated with multiple students.

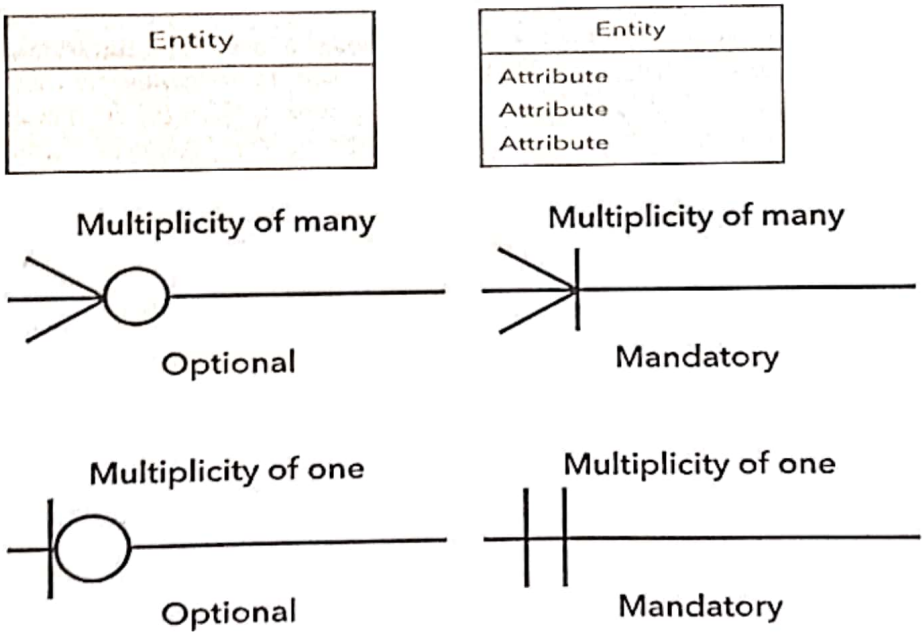


Uses of entity relationship diagrams

Database design: ER diagrams are used to model and design relational databases, in terms of logic, business rules and the specific technology to be implemented. In software engineering, an ER diagram is often an initial step in determining requirements for an information systems project.
Database troubleshooting: ER diagrams are used to analyze existing databases to find and resolve problems in logic or deployment. Drawing the diagram should reveal where it's going wrong.
Business information systems: Any business process that uses fielded data involving entities, actions and interplay can potentially benefit from a relational database. It can streamline processes, uncover information more easily and improve results.
Business process re-engineering (BPR): ER diagrams help in analyzing databases used in business process re-engineering and in modeling a new database setup.
Education: Databases are today's method of storing relational information for educational purposes and later retrieval, so ER Diagrams can be valuable in planning those data structures.
Research: Since so much research focuses on structured data, ER diagrams can play a key role in setting up useful databases to analyze the data.

ER diagram Formation Style

Crow's Foot/Martin/Information Engineering style

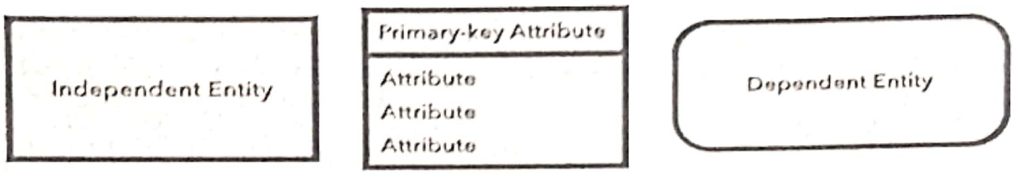


Bachman style

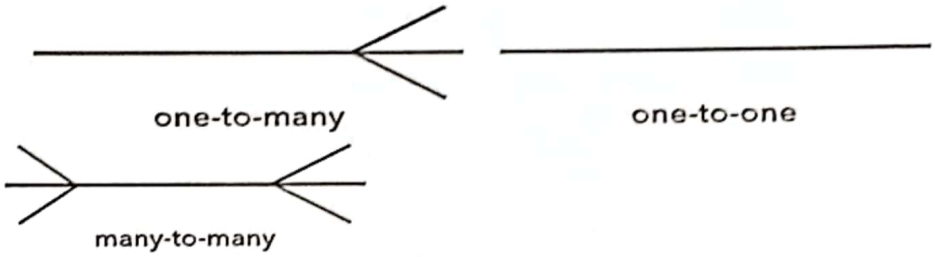
One to One

One to Many

IDEF1X style



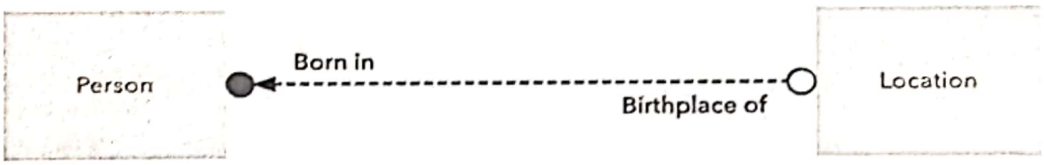
Barker style



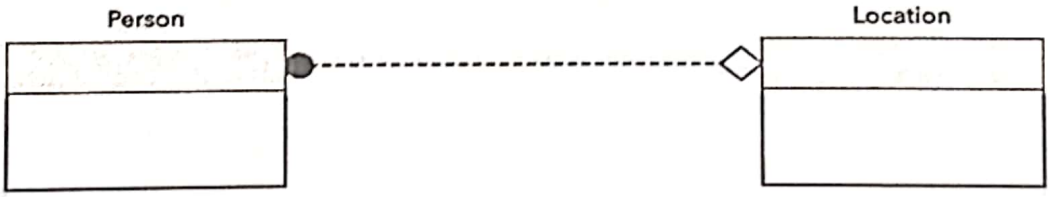
Examples

Following are examples of ERD diagrams made in each system.

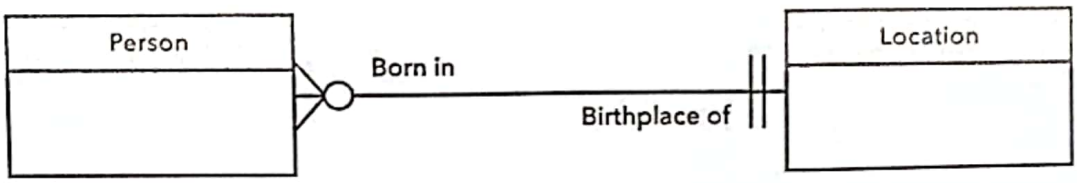
Bachman



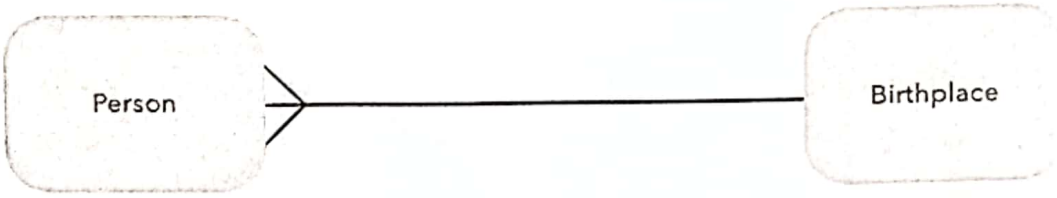
IDEF1X



Crow's Foot



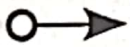
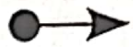
Barker's



Design Phase
Design Phase involves structure chart and pseudocode.

1. **Structure Chart:**

A Structure Chart in software engineering is a chart which shows the breakdown of a system to its lowest manageable parts. They are used in structured programming to arrange program modules into a tree. Each module is represented by a box, which contains the module's name. The tree structure visualizes the relationships between modules, showing data transfer between modules using arrows. Structured Charts are an example of a **top-down** design where a problem (the program) is broken into its components. The tree shows the relationship between modules, showing data transfer between the models.

Symbol	Name	Meaning
Module Name	Process	Each Box represents a programming module, this might be something that calculates the average of some figures, or prints out some pay slips
	Data Couple	Data being passed from module to module that needs to be processed.
	Flag	Check data sent to process to stop or start processes. For example when the End of a File that is being read is reached, or a flag to say whether data sent was in the correct format

Let's take a look at a simple example of how this might be executed when representing the following code:

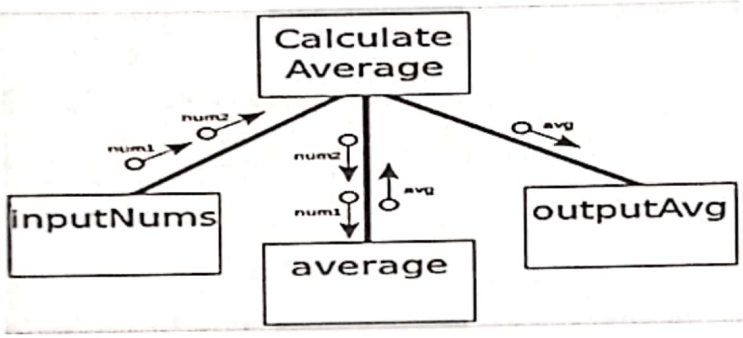
```
dim num1, num2 as integer
sub calculateAverage()
  dim avg as integer
  inputNums()
  avg = average(num1, num2)
```

```
 outputAvg(avg)
end sub

function average(a,b)
 return (a + b) / 2
end function

sub inputNums()
 num1 = console.readline()
 num2 = console.readline()
end sub

sub outputAvg(x)
 console.writeline("average = " & x)
end sub
```



Example

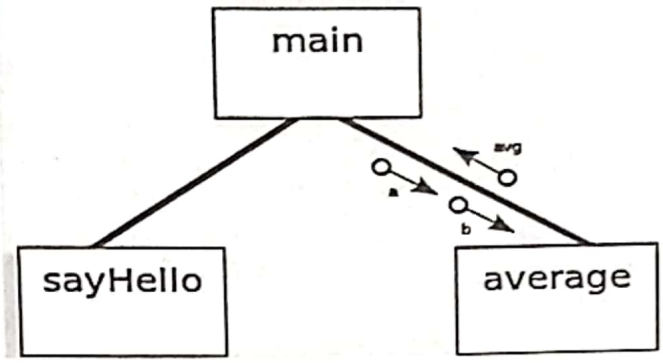
Create structure charts for the following code:

```
sub main()
 dim num1 as integer
 dim num2 as integer
 dim avg as integer
 sayHello()
 num1 = 34
 num2 = 89
 avg = average(num1, num2)
end sub

function average(a,b)
 return (a + b) / 2
end function

sub sayHello()
 console.writeline("hello")
end sub
```

Answer:



2. **Pseudo Code:**
   It is the actual implementation of the system. It is an informal way of programming that doesn't require any specific programming language or technology.

Elements of Structured Programming

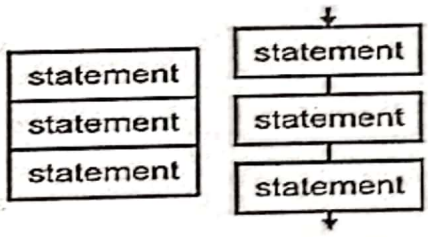
The three fundamental elements of Structured Programming languages are;

1. Control Structures.
2. Subroutine.
3. Blocks.

Control structures

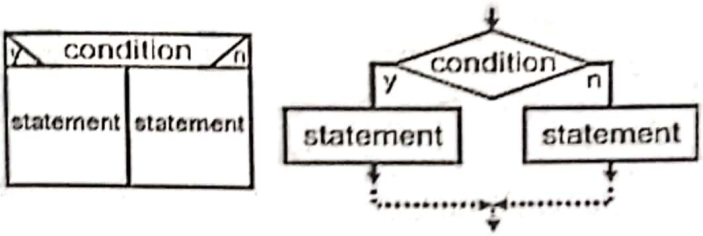
Following the structured program theorem, all programs are seen as composed of control structures:

Sequence: ordered statements or subroutines executed in sequence.



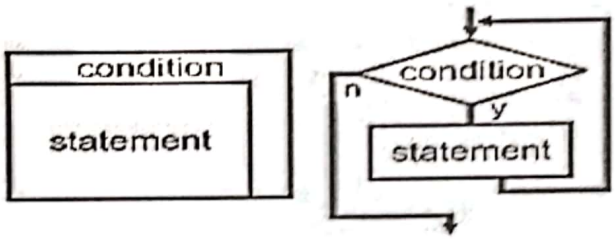
Nassi–Shneiderman diagram (NSD) for a Sequence

Selection: one or a number of statements is executed depending on the state of the program. This is usually expressed with keywords such as if..then..else..endif. The conditional statement should have at least one true condition and each condition should have one exit point at max.



Nassi–Shneiderman diagram (NSD) for a Selection

Iteration: a statement or block is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as while, repeat, for or do..until. Often it is recommended that each loop should only have one entry point (and in the original structural programming, also only one exit point, and a few languages enforce this).



Nassi–Shneiderman diagram (NSD) Repetition

Recursion: a statement is executed by repeatedly calling itself until termination conditions are met. While similar in practice to iterative loops, recursive loops may be more computationally efficient and are implemented differently as a cascading stack.

Subroutines

Subroutines are callable units such as procedures, functions, methods, or subprograms are used to allow a sequence to be referred to by a single statement.

They are sequence of program instructions that performs a specific task, packaged as a unit. This unit can then be used in programs wherever that particular task should be performed.

Basic example

```
Example()                    ' Calls the subroutine
Sub Example                  ' Begins the subroutine
  TextWindow.WriteLine("This is an example of a subroutine in Microsoft Small Basic.") '
What the subroutine does
EndSub                       ' Ends the subroutine
```

Python example

Program to print "Hello world!" followed by "Malgwi is Kind" on the next line.

```
def simple_function():
  print('Hello world!')
  print('Malgwi is Kind')
simple_function()
```

Subroutines may be defined within programs, or separately in libraries that can be used by many programs. In different programming languages, a subroutine may be called a **routine, subprogram, function, method,** or **procedure.**

Different programming languages may use different conventions for passing arguments:

Convention	Description	Usage
Call by value	Argument is evaluated and copy of the value is passed to subroutine	Default in most Algol-like languages after Algol 60, such as Pascal, Delphi, Simula, CPL, PL/M, Modula, Oberon, Ada, and many others. C, C++, Java (References to objects and arrays are also passed by value)
Call by reference	Reference to an argument, typically its address is passed	Selectable in most Algol-like languages after Algol 60, such as Algol 68, Pascal, Delphi, Simula, CPL, PL/M, Modula, Oberon, Ada, and many others. C++, Fortran, PL/I
Call by result	Parameter value is copied back to argument on return from the subroutine	Ada OUT parameters
Call by value-result	Parameter value is copied back on entry to the subroutine and again on return	Algol, Swift in-out parameters
Call by name	Like a macro – replace the parameters with the unevaluated argument expressions	Algol, Scala
Call by constant value	Like call by value except that the parameter is treated as a constant	PL/I, Ada

Call by value: Is a parameter-passing mechanism in which the value of an argument is passed to the subroutine, and the subroutine works with a copy of that value. The original value in the calling program is not affected by any changes made to the parameter inside the subroutine.

When a subroutine is called by value, the actual value of the argument is passed to the subroutine, and a copy of that value is made. The subroutine works with the copy, and any changes made to the copy do not affect the original value in the calling program. This is because the subroutine only has access to the copy of the value, not the original variable.

For example, consider the following code in Python:

```
def add_numbers (a, b):
a= a+1
b= b+1
return a+b
x= 2
y= 3
print (add_numbers (x, y) )
print (x, y)
```

The output of the above subroutine is:
```
7
2, 3
```

Call by reference: instead of passing the actual value of a variable to the subroutine, the memory address of the variable is passed.

When a subroutine is called by reference, any changes made to the parameter inside the subroutine will be reflected in the original variable outside the subroutine. This is because the subroutine has direct access to the memory location of the original variable.

For example, consider the following code in Python:

```
def change_list (lst) :
lst.append(4)
my_list = [1, 2, 3]
change_list (my_list)
print (my_list)
```

Output of the above program is: [1, 2, 3, 4]

In this example, the **change_list()** subroutine takes a list **lst** as a parameter by reference. The **append()** method is used to add the value **4** to the list inside the subroutine. Since the list is passed by reference, the changes made to the list inside the subroutine are reflected in the original list **my_list** outside the subroutine.

Advantages of Subroutines

The advantages of breaking a program into subroutines include:

i. Decomposing a complex programming task into simpler steps: this is one of the two main tools of structured programming, along with data structures
ii. Reducing duplicate code within a program.
iii. Enabling reuse of code across multiple programs.
iv. Dividing a large programming task among various programmers or various stages of a project.

v. Hiding implementation details from users of the subroutine.
vi. Improving readability of code by replacing a block of code with a function call where a descriptive function name serves to describe the block of code. This makes the calling code concise and readable even if the function is not meant to be reused.
vii. Improving traceability.

Blocks

A **block** is a lexical structure of source code which is grouped together. Blocks consist of one or more declarations and statements. Blocks are used to enable groups of statements to be treated as if they were one statement.

Blocks use different syntax in different languages. The broad families are:

- the ALGOL family in which blocks are delimited by the keywords "begin" and "end"
- The C, blocks are delimited by curly braces - "{" and "}".
- The MS-DOS batch language uses parentheses - "(" and ")"
- indentation, as in Python
- s-expressions with a syntactic keyword such as prog or let (as in the Lisp family)

Importance of Blocks

i. Provides the programmer with a way for creating arbitrarily large and complex structures that can be treated as units.
ii. Enables the programmer to limit the scope of variables and sometimes other objects that have been declared.

Examples of Blocks

1. PASCAL

```
if wages > supertax then
        begin
        pays_supertax := true;
        supertax := (wages - supertax) * supertax_rate
        end
     else begin
        pays_supertax := false;
        supertax := 0
        end
```

2. Ada

```
function E(x: real): real;
  function F(y: real): real;
  begin
    F := x + y
  end;
```

3. C

```
#include <stdio.h>
int main()
{
  printf << "Hello, world!\n";
}
```

Principle of Structured Programming Design

As we have seen much that structured programming is an essential aspect of any of the programming languages. It basically makes our program in a form of a linear structure in which the execution of statement of a program starts form the beginning of the program and then linearly flows through the program, executing the statements one by one in a sequence.

So the main principle of the structured programming is to linearize the control flow through a computer program so that the execution follows the sequence in which the code has been written.

This principle makes to the following characteristics clear:

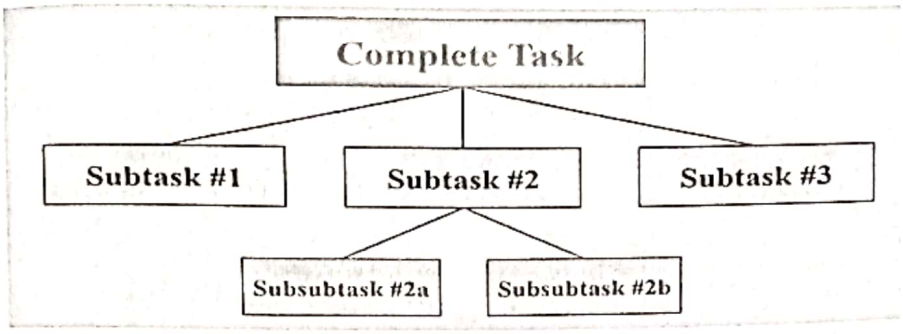
- Only one entry and exit is to be used.
- None of the constructs, sequence, selection and repetition consists of more than three boxes.
- If we visualize any one of the three constructs as they are used, then a third characteristic is evident.
- The entry is the start and the exit is at the end.

Stepwise Refinement

Stepwise Refinement: It was invented by Niklaus Wirth in 1971, it may be the oldest systematic approach to software design still in use. But it remains quite relevant to modern programming.

Stepwise Refinement is the process of breaking down a programming problem into a series of steps. You start with a general set of steps to solve the problem, defining each in turn. Once you have defined each of the steps you then break the problem down into a series of smaller sub-steps. You keep on going until you have described the problem in such a level of detail that you can code a solution to the problem.

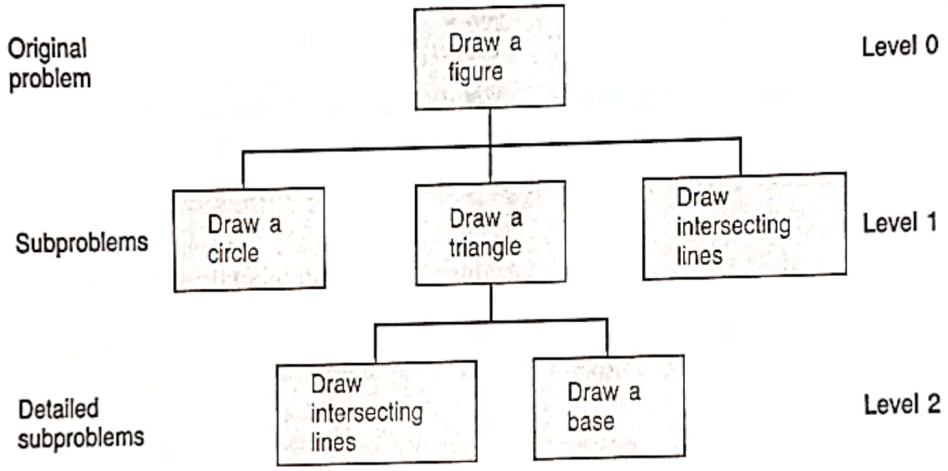
The most effective way to solve a complex problem is to break it down into successively simpler sub problems. • You start by breaking the whole task down into simpler parts. • Some of those tasks may themselves need subdivision. • This process is called stepwise refinement or decomposition.



Stepwise Refinement (Example)

- Start with the initial problem statement
- Break it into a few general steps
- Take each "step", and break it further into more detailed steps
- Keep repeating the process on each "step", until you get a breakdown that is pretty specific, and can be written more or...
- Translate the pseudocode into real code

Example: design a program for drawing a cone.



Structured Programming Design

Operator

Operator is a symbol use to perform some operation on variables, operands or with the constant. Some operator required 2 operands to perform operation while other require single operation.

An operator acts a function which is when applied to data produce a result.

An operation is a defined action upon operand (data item). For example 3 + 5 is an arithmetic operation on two integers. The result of operation 3 + 5 is 8. The operator used to signify this operation has the symbol "+". The operands are the data items operated upon, i n this example the operands are 3 and 5.

Common Operators of Structured Programming Languages

- Arithmetic Operators

- Unary Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Conditional Operator (Ternary Operator)
- Bitwise Operators

Arithmetic Operators: are used for numeric calculation. They are;
+ Addition
- Subtraction
* Multiplication
/ Division
% Modulus (Reminder after integer division)

Unary Operators: act upon a single operand to produce a new value. They include;
++ Increment
-- Decrement
& Address of a value
type Force typed of conversion
! Negation
~ Bitwise complement of

Relational Operators: are used to compare value of two expressions depending o n their relation. Expression that contain relational operator is called relational expression.
< Less than
<= Less than or equal to
> Greater than
>= Greater than or equal to

Equality Operators: are used to check weather two operands are equal or not. There are following two equality Operators.
== Equal to
!= Not equal to

Logical Operators: can act upon operands that are themselves logical expressions. The net effect is to combine the individual logical expressions into more complex conditions that are either true or false. Logical operator is used with one or more operand and return either value zero (for false) or one (for true). The operand may be constant, variables or expressions. And the expression that combines two or more expressions using logical operator is termed as logical expression.

&& Logical AND
II Logical OR
! Logical NOT

Assignment Operators: evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression. There are Several

Assignment operators in C which is used to assign the value of an expression to an identifier. They are implemented as follows;

= Assign right hand side(RHS) value to the left hand side(L HS)
+= Value of LHS variable will be added to the value of RHS and assign it back to the variable in LHS
-= Value of RHS variable will be subtracted from the value of LHS and assign it back to the variable in LHS.
*= Value of LHS variable will be multiplied by the value of R HS and assign it back to the variable in LHS.
/= Value of LHS variable will be divided by the value of RHS and assign it back to the variable in LHS.
%= The Remainder will be stored back to the LHS after integer division is carried out between the LHS variable and the RHS

Expression	Equivalent Expression
i += 5	i = i + 5
f -= g	f = f – g
j *= (i - 3)	j = j * (i - 3)
f /= 3	f = f / 3
i %= (j - 2)	i = i % (j - 2)

Conditional Operator (Ternary Operator ?:) Conditional operator (?:) is used to check and process simple conditions. An expression that makes use of the conditional operator is called a conditional expression.

A conditional expression is written in the form;
Expression 1? expression 2: expression 3

In this case, Expression 1 is evaluated first. If expression 1 is true, then expression 2 is evaluated and becomes the value of the conditional expression. Otherwise expression 3 is evaluated and becomes the value of the conditional expression.

Example;
min = (x1 < x2) ? x1 : x2;
This statement causes the value of the smaller of x1 and x2 to be assigned to min.

Bitwise Operator: permit programmer to access and manipulate of data at bit level.
Various bitwise operator enlisted are
1. one's complement (~)
2. bitwise AND (&)
3. bitwise OR (|)
4. bitwise XOR (^)
5. left shift (<<)
6. Right shift (>>)

These operator can operate on integer and character value but not on float and double

Programming examples;

Example1:

```
int main(void)
 { int i,j,k;
 i=5; j=10; k=15;
printf("\n%d %d %d",i,j,k);
++i;
 i++;
 j--;
 k--;
 k--;
 printf("\n%d %d %d",++i,j++,++k);
 printf("\n%d %d %d",i,j,k);
  return 0;
  }
```

Example2:

```
 int main(void)
  { int i,j,larger;
  printf ("Input 2 integers : ");
 scanf("%d %d",&i, &j);
  larger = i > j ? i : j;
 printf("The largest of two numbers is %d \n", larger);
 return 0;
  }
```

Output : Input 2 integers: 34 45 The largest of two numbers is 45

Data types

Data type are generally classified into four namely;

1. **Basic Types**

   They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.

2. **Enumerated types**

   They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.

3. **The type void**

   The type specifier *void* indicates that no value is available.

4. **Derived types**

   They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges –

Type	Storage size	Value range
Char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
Int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
Long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision

Type	Storage size	Value range	Precision
Float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The void Type

The void type specifies that no value is available. It is used in three kinds of situations –

S/No.	Types & Description

1	**Function returns as void** There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, **int rand(void);**
3	**Pointers to void** A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

Variable

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. There will be the following basic variable types –

S/No.	Type & Description
1	**Char**: Typically a single octet (one byte), a single character.
2	**Int**: The most natural size of integer for the machine.
3	**Float**: A single-precision floating point value.
4	**Double**: A double-precision floating point value.
5	**Void**: Represents the absence of type.

Variable Definition

A variable definition tells the compiler where and how much storage to create for the variable. It specifies a data type and contains a list of one or more variables of that type.

Some valid declarations are shown here –

```
int    i, j, k;
```

```
char   c, ch;
float  f, salary;
double d;
```

Variable Declaration

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only, the compiler needs actual variable definition at the time of linking the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program

Example:

```
Variable declaration:
 int a, b;
int c;
float f;
int main () {

   /* variable definition: */
   int a, b;
   int c;
   float f;

   /* actual initialization */
   a = 10;
   b = 20;

   c = a + b;
   printf("value of c : %d \n", c);

   f = 70.0/3.0;
   printf("value of f : %f \n", f);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
value of c : 30
value of f : 23.333334
```

Object Oriented Programming (OOP)

Object-oriented programming (OOP) is a programming paradigm that revolves around the concept of objects, which are instances of classes that contain data and methods that operate on that data. Here are some basic OOP concepts:

1. **Class:** A blueprint or a template for creating objects. It defines the attributes (data) and methods (functions) that objects of that class can have.
2. **Object:** An instance of a class that contains its own unique data and methods. Each object can have its own values for the attributes defined by its class.
3. **Encapsulation:** The practice of hiding the internal details of an object and exposing only what is necessary to interact with it. This helps prevent unwanted external access to an object's data and methods.
4. **Inheritance:** A mechanism that allows a new class to be based on an existing class, inheriting its attributes and methods. This promotes code reuse and can help organize code into a hierarchy of related classes.
5. **Polymorphism:** The ability of objects to take on multiple forms. In OOP, polymorphism can be achieved through method overloading and method overriding.
6. **Abstraction:** The practice of reducing complex systems to their most essential parts. In OOP, abstraction is achieved by defining classes at various levels of abstraction, from general to specific.

OOP is a powerful programming paradigm that can help organize code, promote code reuse, and create more maintainable software.

First Java Program:
Let us look at a simple code that would print the words *Hello World.*

```
public class MyFirstJavaProgram{

public static void main(String[]args){

System.out.println("Hello World");
}
}
```

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

Open notepad and add the code as above.

Save the file as: MyFirstJavaProgram.java.

Open a command prompt window and go to the directory where you saved the class. Assume it's C:\.

Type ' javac MyFirstJavaProgram.java '

Now, type ' java MyFirstJavaProgram ' to run your program.

You will be able to see ' Hello World ' printed on the window.

```
C :> javac MyFirstJavaProgram.java
C :> java MyFirstJavaProgram
HelloWorld
```

Basic Syntax of Java Program:

About Java programs, it is very important to keep in mind the following points.

Case Sensitivity - Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.

Class Names - For all class names, the first letter should be in Upper Case. If several words are used to form a name of the class, each inner word's first letter should be in Upper Case. Example *class MyFirstJavaClass*

Method Names - All method names should start with a Lower Case letter. If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case. Example *public void myMethodName()*

Program File Name - Name of the program file should exactly match the class name. When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match your program will not compile). Example : Assume 'MyFirstJavaProgram' is the class name, then the file should be saved as *'MyFirstJavaProgram.java'*

public static void main(String args[]) - Java program processing starts from the main() method, which is a mandatory part of every Java program.

Java Identifiers:

All Java components require names. Names used for classes, variables and methods are called identifiers.

In Java, there are several points to remember about identifiers. They are as follows;

i. All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
ii. After the first character, identifiers can have any combination of characters.
iii. A keyword cannot be used as an identifier.
iv. Most importantly identifiers are case sensitive.
v. Examples of legal identifiers: age, $salary, _value, __1_value
vi. Examples of illegal identifiers: 123abc, -salary

Java Modifiers:

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers:

Access Modifiers: default, public, protected, private
Non-access Modifiers: final, abstract, strictfp

For **classes**, you can use either `public` or *default*:

Modifier	Description
`public`	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the declared class
default	The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter
`protected`	The code is accessible in the same package and **subclasses.** You will learn

Java Variables Types:
We would see following type of variables in Java:

i. **Local Variables**
ii. **Class Variables** (Static Variables)
iii. **Instance Variables** (Non-static variables)

Local variables: Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
Instance variables: Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
Class variables: Class variables are variables declared within a class, outside any method, with the static keyword.

Class

A class can be thought of as a user-defined data type that encapsulates data and methods that operate on that data. Each object created from a class has its own set of values for the attributes defined by its class.

For example, you can define a class called "Car" that has attributes such as make, model, and color, and methods such as start(), stop(), and accelerate(). Each object created from the "Car" class will have its own unique set of values for these attributes, and can call these methods to perform various actions on the car object.

Classes provide a way to organize and encapsulate related data and methods, making code more modular and easier to understand and maintain. They also enable code reuse, as you can create multiple objects from the same class.

Object

an object is an instance of a class that contains its own unique data and methods. Each object created from a class has its own set of values for the attributes defined by its class, and can call its methods to perform various actions on the object.

For example, if you have a class called "Person" with attributes such as name, age, and address, you can create objects of this class such as "John", "Mary", and "Bob", each with their own unique set of values for the attributes.

Objects in OOP provide a way to represent real-world entities in software. For example, you can model a bank account as an object with attributes such as account number, balance, and account holder name, and methods such as deposit() and withdraw().

Objects can also interact with each other by calling each other's methods. This enables you to create more complex behavior by composing objects together.

Objects are a fundamental concept in OOP that provide a powerful way to represent and manipulate data and behavior in software

Encapsulation

Encapsulation is achieved by defining the attributes and methods of an object as private or protected. Private attributes and methods can only be accessed within the class that defines them, while protected attributes and methods can be accessed within the class and its subclasses.

To interact with an object, you typically define public methods that provide a controlled interface to the object's internal data and behavior. This way, external code can only access the object through the methods that you've explicitly defined, rather than by directly manipulating its internal state.

Encapsulation provides several benefits in OOP, including:

1. **Security**: By hiding the internal details of an object, you can prevent unwanted access and modification of its data and behavior.
2. **Abstraction**: By providing a controlled interface to an object's data and behavior, you can abstract away the implementation details and focus on the object's essential properties and actions.
3. **Modularity**: By encapsulating related data and behavior within an object, you can create more modular and reusable code.

Encapsulation is a key principle in OOP that helps promote code security, abstraction, and modularity.

```
public class Student {
    private String name;
    private int age;
    private double gpa;

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setGpa(double gpa) {
        this.gpa = gpa;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public double getGpa() {
        return gpa;
    }
}

public class Main {
    public static void main(String[] args) {
        Student student = new Student();
        student.setName("John Smith");
        student.setAge(18);
        student.setGpa(3.5);
```

```
        System.out.println("Name: " + student.getName());
        System.out.println("Age: " + student.getAge());
        System.out.println("GPA: " + student.getGpa());
    }
}
```

In this program, we have a class called **Student** that has three private instance variables: **name**, **age**, and **gpa**. These variables are encapsulated within the class, meaning they cannot be accessed directly from outside the class.

To set and get the values of these variables, we have defined public setter and getter methods. The setter methods (**setName**, **setAge**, and **setGpa**) set the values of the private variables, while the getter methods (**getName**, **getAge**, and **getGpa**) return the values of the private variables.

In the **Main** class, we create a **Student** object and set its name, age, and GPA using the setter methods. We then use the getter methods to retrieve and print out the values of these variables.

By encapsulating the private variables and providing public setter and getter methods, we ensure that the values of the variables can only be accessed and modified in a controlled way, preventing unintended modifications and promoting data integrity.

Inheritance

inheritance is a mechanism that allows a new class (called the subclass or derived class) to be based on an existing class (called the superclass or base class), inheriting its attributes and methods.

The subclass can add its own attributes and methods or override the ones inherited from the superclass. This promotes code reuse and can help organize code into a hierarchy of related classes.

For example, you can define a superclass called "Vehicle" that has attributes such as make, model, and year, and methods such as start() and stop(). You can then define a subclass called "Car" that inherits from "Vehicle" and adds its own attributes such as color and number of doors, as well as methods such as accelerate() and brake().

Inheritance allows you to create specialized classes that share common attributes and methods with their parent classes, while also adding their own unique features. This can save time and reduce code duplication, as you don't have to redefine common attributes and methods in every subclass.

Inheritance also enables polymorphism, which allows objects of different classes to be treated as if they were objects of a common superclass. This can simplify code by allowing you to write code that works with a general type of object, without having to know the specific subclass.

Inheritance is a powerful concept in OOP that allows you to create hierarchies of related classes and promotes code reuse and modularity.

```
public class Vehicle {
   protected String make;
   protected String model;
   protected int year;

   public Vehicle(String make, String model, int year) {
      this.make = make;
      this.model = model;
      this.year = year;
   }
   public void start() {
      System.out.println("Starting the vehicle");
   }
   public void stop() {
      System.out.println("Stopping the vehicle");
   }
}
public class Car extends Vehicle {
   private int numDoors;

   public Car(String make, String model, int year, int numDoors) {
      super(make, model, year);
      this.numDoors = numDoors;
   }
   public void drive() {
      System.out.println("Driving the car");
   }
}
public class Motorcycle extends Vehicle {
   private boolean hasSidecar;

   public Motorcycle(String make, String model, int year, boolean hasSidecar) {
      super(make, model, year);
      this.hasSidecar = hasSidecar;
   }

   public void wheelie() {
      System.out.println("Popping a wheelie on the motorcycle");
   }
}

public class Main {
   public static void main(String[] args) {
      Car myCar = new Car("Honda", "Civic", 2021, 4);
      myCar.start();
      myCar.drive();
      myCar.stop();
```

```
    Motorcycle myMotorcycle = new Motorcycle("Harley-Davidson", "Sportster",
2022, false);
    myMotorcycle.start();
    myMotorcycle.wheelie();
    myMotorcycle.stop();
  }
}
```

In this program, we have a base class called **Vehicle** that defines three instance variables (**make**, **model**, and **year**) and two methods (**start** and **stop**) that print out messages indicating that the vehicle is starting or stopping.

We then define two subclasses of **Vehicle**: **Car** and **Motorcycle**. Each subclass extends the **Vehicle** class and adds its own instance variables and methods. The **Car** class adds an instance variable called **numDoors** and a method called **drive** that prints out a message indicating that the car is being driven. The **Motorcycle** class adds an instance variable called **hasSidecar** and a method called **wheelie** that prints out a message indicating that the motorcycle is performing a wheelie.

In the **Main** class, we create an instance of **Car** and an instance of **Motorcycle** and call various methods on each object. Since each object is actually an instance of a different subclass (**Car** or **Motorcycle**), it has access to all of the methods and instance variables defined in both its own class and its parent class (**Vehicle**), demonstrating inheritance.

The key concept of inheritance here is that we are able to define a base class (**Vehicle**) with common functionality and instance variables that are shared by all subclasses (**Car** and **Motorcycle**). Each subclass can then add its own instance variables and methods that are specific to its own type of vehicle, while still inheriting the common functionality from the parent class. This allows us to reuse code and avoid duplicating code between similar classes.

Polymorphism

Polymorphism enables you to write code that works with a general type of object, without having to know the specific subclass. This promotes code flexibility and can simplify code by allowing you to write generic algorithms that work with a variety of objects.

There are two main types of polymorphism in OOP:

1. **Static Polymorphism**: This is also known as compile-time polymorphism, and is achieved through function overloading or operator overloading. Function overloading is when you define multiple functions with the same name but different parameter types or numbers, allowing the compiler to choose the correct function based on the arguments passed. Operator overloading is when you redefine the behavior of an operator for a class, allowing you to use the operator with objects of that class.
2. **Dynamic Polymorphism**: This is also known as runtime polymorphism, and is achieved through inheritance and method overriding. Method overriding is when a subclass defines a method with the same name and signature as a method in its superclass, allowing the

subclass to override the behavior of the superclass method. When you call the overridden method on an object of the subclass, the subclass method is executed instead of the superclass method.

Polymorphism is a powerful concept in OOP that promotes code flexibility and simplifies code by allowing you to write generic algorithms that work with a variety of objects.

```
public class Animal {
  public void makeSound() {
    System.out.println("The animal makes a sound");
  }
}
public class Cat extends Animal {
  public void makeSound() {
    System.out.println("The cat meows");
  }
}
public class Dog extends Animal {
  public void makeSound() {
    System.out.println("The dog barks");
  }
}
public class Main {
  public static void main(String[] args) {
    Animal[] animals = new Animal[2];
    animals[0] = new Cat();
    animals[1] = new Dog();

    for (Animal animal : animals) {
      animal.makeSound();
    }
  }
}
```

In this program, we have a base class called **Animal** that defines a single method called **makeSound**, which simply prints out a message saying that the animal makes a sound.

We then define two subclasses of **Animal**: **Cat** and **Dog**. Each subclass overrides the **makeSound** method to produce a different message, representing the sound that the animal makes.

In the **Main** class, we create an array of **Animal** objects and populate it with one **Cat** object and one **Dog** object. We then loop through the array and call the **makeSound** method on each object. Since each object is actually an instance of a different subclass (**Cat** or **Dog**), the appropriate **makeSound** method is called for each object, demonstrating polymorphism.

The key concept of polymorphism here is that we are able to treat objects of different classes as if they are all objects of the same base class (**Animal**). This allows us to write code that can

work with any object that extends the **Animal** class, without having to know the specific details of each subclass.

Abstraction

Abstraction is a key principle in OOP that allows you to create models of complex systems that are easier to understand and manipulate. It promotes code flexibility, modularity, and maintainability by allowing you to focus on essential properties and actions while ignoring unnecessary details.

Abstraction is achieved by defining classes and objects that encapsulate the essential properties and actions of a system, while hiding the implementation details. This helps reduce the complexity of the system and makes it easier to understand and use.

For example, if you were creating a banking application, you might define an abstract class called "Account" that encapsulates the essential properties and actions of a bank account, such as balance, deposit, and withdrawal. You could then define more specific account types such as "CheckingAccount" and "SavingsAccount" that inherit from the "Account" class and add their own unique features.

Abstraction also enables you to define interfaces that specify the essential properties and actions of an object without specifying how they are implemented. This allows different objects to provide different implementations of the same interface, promoting code flexibility and modularity.

```
public abstract class Account {
    private double balance;
    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) {
        balance -= amount;
    }
    public double getBalance() {
        return balance;
    }
    public abstract void calculateInterest();
}
public class SavingsAccount extends Account {
    public void calculateInterest() {
        double balance = getBalance();
        double interestRate = 0.05;
        double interest = balance * interestRate;
        deposit(interest);
    }
}
public class CheckingAccount extends Account {
    public void calculateInterest() {
        // Checking accounts do not earn interest
    }
}
```

```
public class Main {
  public static void main(String[] args) {
    Account savings = new SavingsAccount();
    savings.deposit(1000);
    savings.calculateInterest();
    System.out.println("Savings Balance: " + savings.getBalance());

    Account checking = new CheckingAccount();
    checking.deposit(500);
    checking.withdraw(100);
    System.out.println("Checking Balance: " + checking.getBalance());
  }
}
```

In this program, we have an abstract class called **Account** that defines three methods: **deposit**, **withdraw**, and **getBalance**. The **deposit** and **withdraw** methods update the balance of the account, while the **getBalance** method returns the current balance. The **Account** class also defines an abstract method called **calculateInterest**, which we will implement differently in each subclass.

We then define two concrete classes that extend the **Account** class: **SavingsAccount** and **CheckingAccount**. The **SavingsAccount** class overrides the **calculateInterest** method to calculate and deposit interest into the account, while the **CheckingAccount** class leaves the **calculateInterest** method empty since checking accounts do not earn interest.

Finally, in the **Main** class, we create two **Account** objects, one of type **SavingsAccount** and one of type **CheckingAccount**. We deposit and withdraw money from these accounts, and call the **calculateInterest** method on the **SavingsAccount** object to earn interest. We then print out the current balance of each account.

The key concept of abstraction here is that we are able to define a general **Account** interface that defines common methods and behavior for all types of accounts. We can then create specific subclasses (**SavingsAccount** and **CheckingAccount**) that implement the interface in their own way, allowing us to write code that works with accounts in general, without having to know the specific details of each type of account.