UNIT-5

1. Basic concepts –
   (1) Exception – It is any unusual event, erroneous or not, that is detectible by either hardware or software & that may require special processing.
   (2) Exception Handling – The special processing that may be required when an exception is detected.
   (3) Exception Handler – Special processing done by a code unit or segment.

2. Practical examples of Exception Handling –

3. Advantages of Exception Handling –
   (1) Detects error
   (2) Exception propagation – It allows an exception raised in one program unit to be handled in some other unit in its dynamic or static ancestry.
   (3) Unusual situations can be simplified.
   (4) Encourage users to consider all the events that could occur during program execution and how they can be handled.

4. Design Issues of Exception Handling –
   (1) How and where are exception handlers specified, and what is their role?
   (2) How is an exception occurrence bound to an exception handler?
   (3) Can information about an exception be passed to the handler?
   (4) Where does execution continue if at all, after an exception handler completes its execution?
   (5) Is some form of finalization provided?
   (6) How are user-defined exceptions specified?
   (7) If there are predefined exceptions, should there be default exception handler for programs that do not provide their own?
   (8) Can predefined exceptions be explicitly raised?
   (9) Are hardware-detectable errors treated as exceptions that may be handled?
   (10) Are there any predefined exceptions?
   (11) Should it be possible to disable predefined exceptions?
Exception Handling in C++

In C++, exceptions are user or library-defined and explicitly raised.

→ Exception Handles:
  try clause:
  
  ```
  try {
      // /* Code that might raise an exception.
  }
  catch (formal parameter) { // Each catch function is an exception handler.
      // /* A handler body
  }
  catch (formal parameter) { // Catch function can have only one.
      // /* A handler body
  }
  ``

→ C++ exceptions are raised only by the explicit statement 'throw',
  `throw [expression];`

→ A C++ function can list the types of the exceptions (the type of
  the throw expression) that it could raise. This is done by attaching
  the reserved word 'throw', followed by a parenthegized list of these
  types, to the function header:

  ```
  typedecl | throw (int | char * ) & ...
  ```

Exception Handling in Java:

Java includes a collection of predefined exceptions that are
implicitly raised by the Java Virtual Machine (JVM).

→ Classes of exceptions:

All Java exceptions are objects of classes that are descendants
of the Throwable class.

Subclasses of Throwable class are Error and Exception.

Subclass of Exception is any user-defined exceptions, and the
Exception class has two system-defined direct descendants that
are RuntimeException and IOException.
Exception Handlers -

- Some form as those of C++, except that every catch must have a parameter and the class of the parameter must be a descendant of the predefined class Throwable.
- Exceptions of class Error and RuntimeException and their descendants are called unchecked exceptions. All other exceptions are called checked exceptions.
- There is no default exception handler, and it is not possible to disable exceptions.
- finally clause - It is used when some situations in which a process must be executed regardless of whether a try clause throws an exception and regardless of whether a thrown exception is caught in a method.

```java
try {
    // code
}
catch (...) {
    // code
} // more handlers
finally {
    // code
}
```

- Assertions - Two possible forms:
  - `assert condition;` // condition is tested. If true, nothing happens.
  - `assert condition: expression;` // If false, AssertionError exception is thrown.
- `assert condition: expression;` // Same as above, except that the value of the expression is passed to the AssertionError constructor as a string and becomes debugging output.
LOGIC PROGRAMMING LANGUAGE -

1. Programming that uses a form of symbolic logic as a programming language is often called logic programming, and languages based on symbolic logic are called logic programming languages, or declarative languages. Eg - Prolog

2. Overview of logic programming -
   - The basic concept of declarative semantics is that there is a simple way to determine the meaning of each statement, and it does not depend on how the statement might be used to solve a problem.
   - Programming in a logic programming language is non-structured.

3. Basic elements of Prolog -
   - Term: A Prolog term is a constant, a variable, or a structure.
   - A constant is either an atom or an integer. Atoms are symbolic values of Prolog.
   - Instantiation: The binding of a value, and thus a type, to a variable.
   - Uninstantiated: A variable that has not been assigned a value.
   - A variable is any string of letters, digits, and underscores that begins with an uppercase letter or an underscore (_).
   - Structures represent the atomic propositions of predicate calculus, and their general form is the home: functor (parameter list).
   - The functor is any atom and is used to identify the structure.
   - A parameter list can be any list of atoms, variables, or other structures.

Fact statements -

Two basic statement forms - Headless and Headed horn clause.

Headless horn clause - single structure, which is interpreted as an unconditional assertion, e.g., male (jake)
Rule statement

Headsed Horn clause - It can be related to a known theorem in mathematics from which a conclusion can be drawn if the set of given conditions is satisfied. They called rules.

Example - ancestor (mary, shelley) -> mother (mary, shelley).

If Mary is the mother of Shelley, then Mary is an ancestor of Shelley.

Headsed Horn clauses are called rules, because they state rules of implication between propositions.

Goal statement

The theorem is in the form of a proposition that we want the system to either prove or disprove. In Prolog, these propositions are called goals, or queries. Example - man (Fred).

Conjunctive propositions and propositions with variables are also legal goals.

Inferencing Pattern of Prolog

When a goal is a compound proposition, each of the facts (stating) is called a subgoal.

Matching - Proposition matching focused.

Satisfying - Proving a subgoal.

Forward Chaining (bottom-up resolution)

System begins with the facts and rules of the database and attempt to find a sequence of matches that lead to a goal.

Backward Chaining (top-down resolution)

System begins with the goal and attempt to find a sequence of matching propositions that lead to some set of original facts in the database. Query:

Example - man (bob), database contains: father (bob),

man (X); - father (X).
A depth-first search finds a complete sequence of propositions for a proof for the first subgoal before working on the others. A breadth-first search works on all subgoals of a given goal in parallel.

Backtracking: Backing up in the goal to the recombination of a previously found subgoal.

- Simple arithmetic:
  - \(+(7, x)\) → sum of 7 and the variable \(x\).
  - \(A = \frac{8}{17} + C\) → \(A = \frac{8}{17} + C\).

- Prolog example:
  - speed (ford, 100).
  - speed (chevy, 125).
  - time (ford, 20).
  - time (chevy, 24).

- distance \((X, Y)\) = speed \((X, speed)\) * time \((X, Time)\), \(Y\) is Speed * Time.

- Tracing model:
  - model for execution of Prolog programs. The tracing model describes Prolog execution in terms of four events:
  1. call, which occurs at the beginning of an attempt to satisfy a goal
  2. fail, which occurs when a goal has been satisfiable
  3. redo, which occurs when backtracking causes an attempt to satisfy a goal
  4. fail, which occurs when a goal fail.

- list structure:
  - [apple, prune, grape, kumquat]
  - \([X \mid Y]\) → list with head \(X\) and tail \(Y\)
  - creating \(new\ list\) = new list ([apple, prune, grape, kumquat]).

4. Applications of logic programming:
   1. RDBMS (Relational Database Management Systems)
   2. Expert systems → designed to emulate human expertise, e.g., APES
   3. Natural Language Processing
Functional Programming Languages

1. The functional programming paradigm, which is based on mathematical functions, is the design basis of the most important non-imperative styles of languages. This style of programming is supported by functional programming languages.
   → One characteristic feature of a pure functional programming language is that neither expansions nor functions have side effects. 
   \[\text{e.g. - LISP developed in 1959}\]

2. Fundamentals of functional programming language -
   → Objective is to mimic mathematical functions to the greatest extent possible.
   → A purely functional programming language does not use variables or assignment statements. Therefore, programs are function definitions and function application specifications, and executions consist of evaluating function applications.
   → The execution of a function always produces the same result when given the same parameters. This feature is called referential transparency.

3. Introduction to 4GL - (Fourth-Generation Languages)
   They are closer to human language. \[\text{e.g.- Oracle, VB, VC++, SQL}\]
   Most 4GL are used to access databases. They allow the programmer to define 'what' is required without telling the computer 'how' to implement it.

   → Features of 4GL languages are:
   1. Ease of use
   2. Extended range of functions
   3. Availability of options
   4. Default options