UNIT-I

Why study programming languages?

♦ Toimprove the ability to develop effective algorithms.

♦ Toimprove the use of familiar languages.

♦ Toincrease the vocabulary of useful programming constructs.

♦ Toallow a better choice of programming language.

♦ Tomakeit easier to learn a new language.

♦ Tomakeit easier to design a new language.

♦ Tosimulate useful features in languages that lack them.

♦ Tomakebetter use of language technology wherever it appears**.**

Programming Domains

- Scientific applications
  - Large number of floating point computations
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated
  - LISP
- Systems programming
  - Need efficiency because of continuous use
  - C
- Web Software
  - Eclectic collection of languages: markup (**e.g.**, XHTML), scripting (**e.g.**, PHP), general-purpose (**e.g.**, Java)

Language Evaluation Criteria –

- Readability : the ease with which programs can be read and understood
- Writability : the ease with which a language can be used to create programs
- Reliability : conformance to specifications (i.e., performs to its specifications)
- Cost : the ultimate total cost

Readability

- Overall simplicity
  - A manageable set of features and constructs
  - Few feature multiplicity (means of doing the same operation)
  - Minimal operator overloading
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
- Control statements
  - The presence of well-known control structures (**e.g.**, while statement)
- Data types and structures
  - The presence of adequate facilities for defining data structures
- Syntax considerations
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Writability

- Simplicity and Orthogonality
  - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
  - The ability to define and use complex structures or operations in ways that allow details to be ignored
- Expressivity
  - A set of relatively convenient ways of specifying operations
  - Example: the inclusion of for statement in many modern languages

Reliability

- Type checking
  - Testing for type errors
- Exception handling
  - Intercept run-time errors and take corrective measures
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
  - A language that does not support “natural” ways of expressing an algorithm will necessarily use “unnatural” approaches, and hence reduced reliability

Cost

- Training programmers to use language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs

- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

Others

- Portability
  - The ease with which programs can be moved from one implementation to another
- Generality
  - The applicability to a wide range of applications
- Well-definedness
  - The completeness and precision of the language‘s official definition

Influences on Language Design

- Computer Architecture
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- Programming Methodologies
  - New software development methodologies (**e.g.**, object-oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
- Variables model memory cells
- Assignment statements model piping
- Iteration is efficient

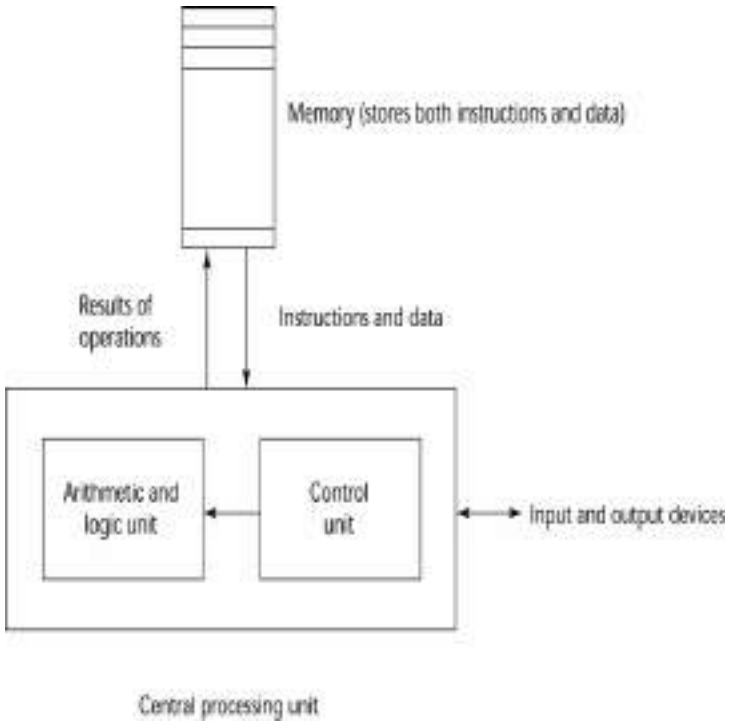


Figure 1.1 The von Neumann Computer Architecture

Programming Methodologies

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  - data abstraction
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

Language Categories –

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Examples: C, Pascal
- Functional
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme
- Logic
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog
- Object-oriented
  - Data abstraction, inheritance, late binding
  - Examples: Java, C++
- Markup
  - New; not a programming per se, but used to specify the layout of information in Web documents
  - Examples: XHTML, XML

Language Design Trade-Offs

- Reliability vs. cost of execution
  - Conflicting criteria
  - Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- Readability vs. writability
  - Another conflicting criteria
  - Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - Another conflicting criteria
  - Example: C++ pointers are powerful and very flexible but not reliably used

Implementation Methods -

- Compilation
  - Programs are translated into machine language
- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters

Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code

- code generation: machine code is generated

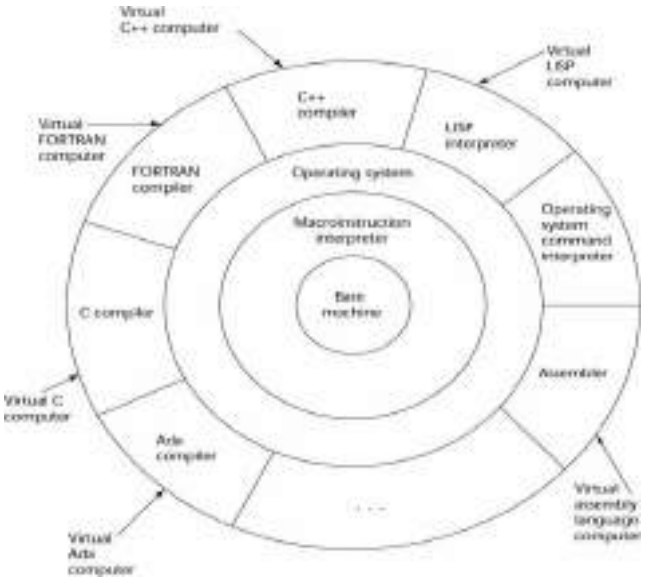


Figure 1.2 Layered View of Computer: The operating system and language implementation are layered over Machine interface of a computer

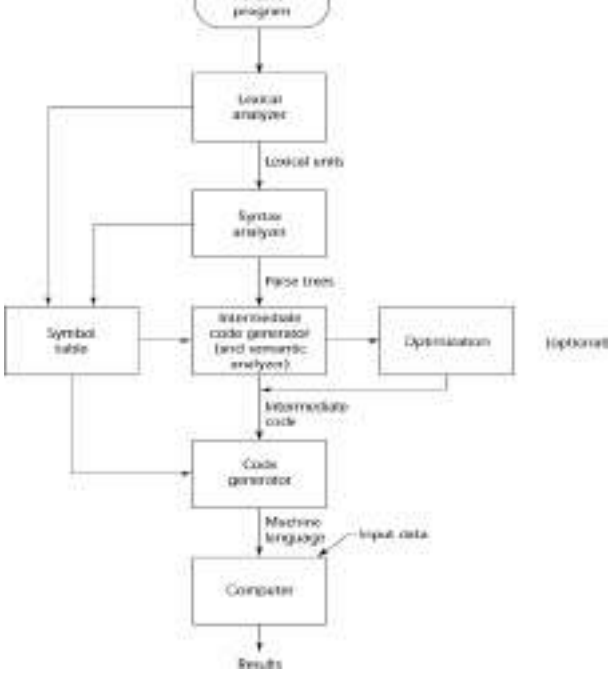


Figure 1.3 The Compilation Process

Additional Compilation Terminologies

- Load module (executable image): the user and system code together
- Linking and loading: the process of collecting system program and linking them to user program

Execution of Machine Code

- Fetch-execute-cycle (on a von Neumann architecture)

initialize the program counter
repeat forever
fetch the instruction pointed by the counter
increment the counter
decode the instruction
execute the instruction
end repeat

Von Neumann Bottleneck

- Connection speed between a computer‘s memory and its processor determines the speed of a computer
- Program instructions often can be executed a lot faster than the above connection speed; the connection speed thus results in a *bottleneck*
- Known as von Neumann bottleneck; it is the primary limiting factor in the speed of computers

interpreter

The interpreter in the compiler checks the source code line-by-line and if an error is found on any line, it stops the execution until the error is resolved. Error correction is quite easy for the interpreter as the interpreter provides a line-by-line error. But the program takes more time to complete the execution successfully. I

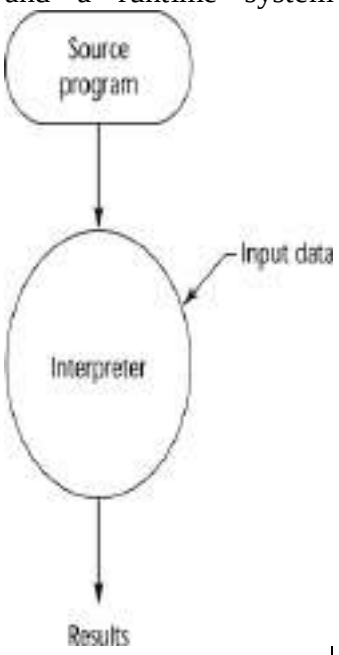
Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Becoming rare on high-level languages

- Significantly comeback with some latest web scripting languages (**e.g.**, JavaScript)

Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a runtime system (together, thesare called *Ja*



Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile intermediate language into machine code
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system

Preprocessors -

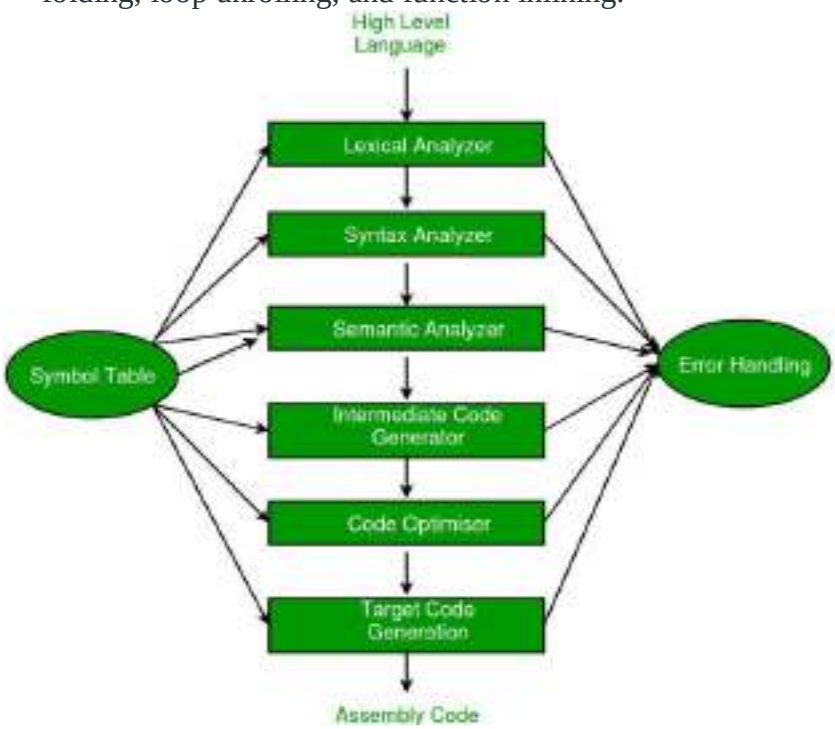
- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
  - expands #include, #define, and similar macros

Compiler

The compiler is software that converts a program written in a high-level language (Source Language) to a low-level language (Object/Target/Machine Language/0, 1's).

Phases of Compiler

1. **Lexical Analysis:** The first stage of compiler design is lexical analysis, also known as scanning. In this stage, the compiler reads the source code character by character and breaks it down into a series of tokens, such as keywords, identifiers, and operators. These tokens are then passed on to the next stage of the compilation process.
2. **Syntax Analysis:** The second stage of compiler design is syntax analysis, also known as parsing. In this stage, the compiler checks the syntax of the source code to ensure that it conforms to the rules of the programming language. The compiler builds a parse tree, which is a hierarchical representation of the program's structure, and uses it to check for syntax errors.
3. **Semantic Analysis:** The third stage of compiler design is semantic analysis. In this stage, the compiler checks the meaning of the source code to ensure that it makes sense. The compiler performs type checking, which ensures that variables are used correctly and that operations are performed on compatible data types. The compiler also checks for other semantic errors, such as undeclared variables and incorrect function calls.
4. **Code Generation:** The fourth stage of compiler design is code generation. In this stage, the compiler translates the parse tree into machine code that can be executed by the computer. The code generated by the compiler must be efficient and optimized for the target platform.
5. **Optimization:** The final stage of compiler design is optimization. In this stage, the compiler analyzes the generated code and makes optimizations to improve its performance. The compiler may perform optimizations such as constant folding, loop unrolling, and function inlining.



Syntax vs Semantics in Programming Languages

Syntax refers to the structure of the elements of a language based on its type. On the other hand, semantics are all about the meaning.

Something written syntactically correctly in a language can be completely meaningless. And no text can be meaningful if its syntax is incorrect.

The following code is syntactically correct but semantically wrong because it's not possible to reassign something to a constant variable:

```
const name = "Palash";


name = "Akash";
```

The following is syntactically incorrect and thus does not even have any chance to be semantically correct.

```
"Palash" = const name;"Akash" = name;
```

Terminals and Non-Terminals

BNF/EBNF is usually used to specify the grammar of a language. Grammar is a set of *rules* (also called *production rules*). Here language refers to nothing but a set of strings that are valid according to the rules of its grammar.

A BNF/EBNF grammar description is an unordered list of rules. *Rules* are used to define *symbols* with the help of other symbols.

You can think of *symbols* as the building blocks of grammar. There are two kinds of symbols:

- **Terminal (or Terminal symbol)**: Terminals are strings written within quotes. They are meant to be used as they are. Nothing is hidden behind them. For example "freeCodeCamp" or "firefly".
- **Non-terminal (or Non-terminal symbol)**: Sometimes we need a name to refer to something else. These are called *non-terminals*. In BNF, *non-terminal* names are written within angle brackets (for example <statement>), while in EBNF they don't usually use brackets (for example statement).

What is BNF?

BNF stands for **B**ackus–**N**aur **F**orm which resulted primarily from the contributions of John Backus and Peter Naur.

```
<something> ::= "content"
```

Each rule in BNF (also in ENBF) has three parts:

- **Left-hand side**: Here we write a non-terminal to define it. In the above example, it is <something>.
- ::=: This character group separates the **Left hand side** from **Right hand side**. Read this symbol as "is defined as".
- **Right-hand side**: The definition of the non-terminal specified on the right-hand side. In the above example, it's "content".

  The above <something> is just one thing fixed thing. Let's now see all the ways you can compose a *non-terminal*.

  How to compose a non-terminal
  BNF offers two methods to us:

- Sequencing
- Choice

  You can just write a combination of one or more terminals or non-terminals in a sequence and the result is their concatenation, with non-terminals being replaced by their content. For example, you can express your breakfast in the following ways:

  <breakfast> ::= <drink> " and biscuit"<drink> ::= "tea"

  It means the only option for breakfast for you is "tea and biscuit". Note that here, the order of symbols is important.

  <breakfast> ::= <drink> " and biscuit"<drink> ::= "tea" | "coffee"

  The | operator indicates that the parts separated by it are choices.

  What is EBNF?

  digits = digit { digit }digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

  The braces above mean that its inner part may be repeated 0 or more times. It frees your mind from getting lost in recursion.

  One interesting fact is that everything you can express in EBNF can also be expressed in BNF.

  EBNF usually uses a slightly different notation than BNF. For example:

- ::= becomes just =.
- There are no angle brackets around non-terminals.

  For concatenation, instead of juxtaposition, some prefer `,` to be more explicit. However, I will not use it here.

  Don't assume that these styles to be universal. There are several variants of them and they are usually clear from the context. The more important thing to focus on is the new operations it offers like the braces we've seen above.

  EBNF extends BNF by adding the following 3 operations:

- Option
- Repetition
- Grouping

  Option
  Option uses square brackets to make the inner content optional. Example:

  ```
  thing = "water" [ "melon" ]
  ```

  So the above thing is either water or watermelon.

  Repetition
  Curly braces indicate the inner content may be repeated 0 or more times. You have already seen a good example of it above. Below is a very simple one just to make the idea solid in your mind:

  ```
  long_google = "Goo" { "o" } "gle"
  ```

  So "Google", "Gooogle", "Gooooooogle" are all valid long_google non-terminal.

  Grouping
  Parentheses can be used to indicate grouping. It means everything they wrap can be replaced with any of the valid strings that the contents of the group represent according to the rules of EBNF. For example:

  ```
  fly = ("fire" | "fruit") "fly"
  ```

  Here fly is either "firefly" or "fruitfly".

With BNF we could not do that in one line. It would look like the following in BNF:

<fly> ::= <type> "fly"

<type> ::= "fire" | "fruit"

Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

1. G= (V, T, P, S)

   Where,

   G describes the grammar

   T describes a finite set of terminal symbols.

   V describes a finite set of non-terminal symbols

   P describes a set of production rules

   S is the start symbol.

   Production rules:

1. S → aSa
2. S → bSb
3. S → c

   Now check that abbcbba string can be derived from the given CFG.

1. S aSa
2. S abSba
3. S abbSbba

4. S abbcbba

   By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

   ### Derivation

   Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

   - We have to decide the non-terminal which is to be replaced.
   - We have to decide the production rule by which the non-terminal will be replaced.

   We have two options to decide which non-terminal to be replaced with production rule.

   ### Left-most Derivation

   In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

   ### Example:

   Production rules:

1. S = S + S
2. S = S - S
3. S = a | b |c

   Input:

   a - b + c

   The left-most derivation is:

1. S = S + S
2. S = S - S + S
3. S = a - S + S
4. S = a - b + S
5. S = a - b + c

Right-most Derivation

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

Example:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

The right-most derivation is:

1. S = S - S
2. S = S - S + S
3. S = S - S + c
4. S = S - b + c
5. S = a - b + c

Parse tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Example:

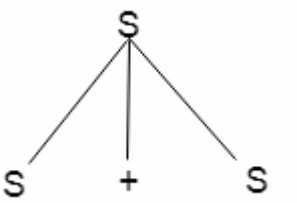
Production rules:

1. T= T + T | T * T
2. T = a|b|c

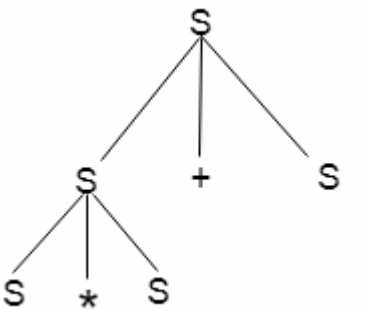
Input:

a * b + c

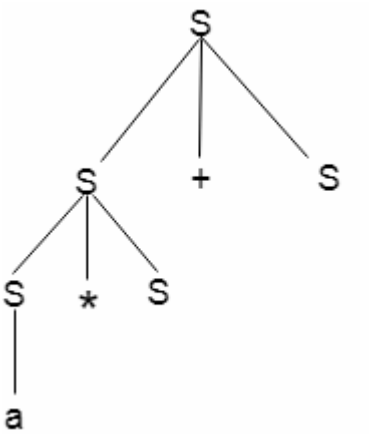
Step 1:



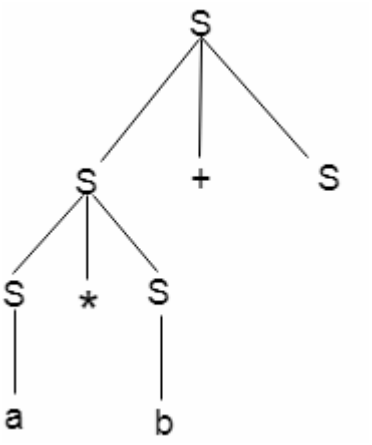
Step 2:



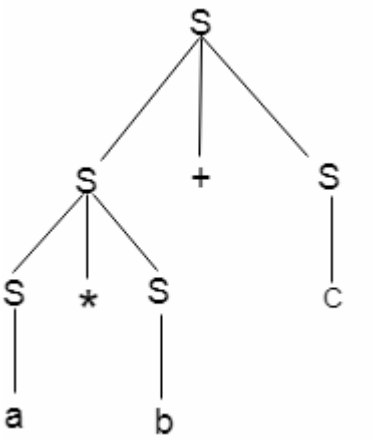
Step 3:



Step 4:



Step 5:



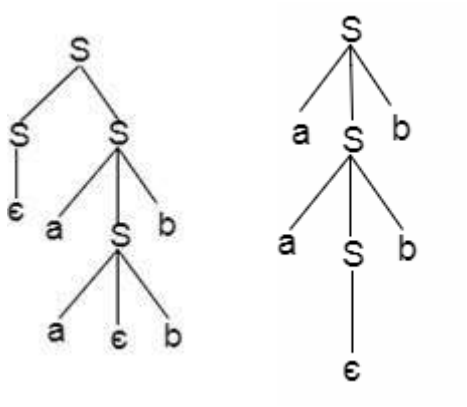
Ambiguity

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:

1. S = aSb | SS
2. S =

For the string aabb, the above grammar generates two parse trees:



If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

What is BNF?

BNF stands for **B**ackus–**N**aur **F**orm which resulted primarily from the contributions of John Backus and Peter Naur.

Below is an example of a simple *production rule* in BNF:

<something> ::= "content"

Each rule in BNF (also in ENBF) has three parts:

- **Left-hand side**: Here we write a non-terminal to define it. In the above example, it is <something>.
- ::=: This character group separates the **Left hand side** from **Right hand side**. Read this symbol as "is defined as".
- **Right-hand side**: The definition of the non-terminal specified on the right-hand side. In the above example, it's "content".

  The above <something> is just one thing fixed thing. Let's now see all the ways you can compose a *non-terminal*.

How to compose a non-terminal

BNF offers two methods to us:

- Sequencing
- Choice

  You can just write a combination of one or more terminals or non-terminals in a sequence and the result is their concatenation, with non-terminals being replaced by their content. For example, you can express your breakfast in the following ways:

<breakfast> ::= <drink> " and biscuit"<drink> ::= "tea"

It means the only option for breakfast for you is "tea and biscuit". Note that here, the order of symbols is important.

Let's say someday you want to drink coffee instead of tea. In this case, you can express your possible breakfast items like below:

<breakfast> ::= <drink> " and biscuit"<drink> ::= "tea" | "coffee"

The | operator indicates that the parts separated by it are choices. there is no difference between "tea" | "coffee and "coffee" | "tea".

As a simple example let's see how you express one or more digits in BNF:

<digits> ::= <digit> | <digit> <digits><digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

What is EBNF?
BNF is fine, but sometimes it can become verbose and hard to interpret. EBNF (which stands for **E**xtended **B**ackus–**N**aur **F**orm) may help you in those cases. For example, the previous example can be written in EBNF like below:

digits = digit { digit }digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

The braces above mean that its inner part may be repeated 0 or more times. It frees your mind from getting lost in recursion.

One interesting fact is that everything you can express in EBNF can also be expressed in BNF.

EBNF usually uses a slightly different notation than BNF. For example:

- ::= becomes just =.
- There are no angle brackets around non-terminals.

For concatenation, instead of juxtaposition, some prefer `,` to be more explicit. However, I will not use it here.

Don't assume that these styles to be universal. There are several variants of them and they are usually clear from the context. The more important thing to focus on is the new operations it offers like the braces we've seen above.

EBNF extends BNF by adding the following 3 operations:

- Option
- Repetition
- Grouping

Option
Option uses square brackets to make the inner content optional. Example:

```
thing = "water" [ "melon" ]
```

So the above thing is either water or watermelon.

Repetition
Curly braces indicate the inner content may be repeated 0 or more times. You have already seen a good example of it above. Below is a very simple one just to make the idea solid in your mind:

```
long_google = "Goo" { "o" } "gle"
```

So "Google", "Gooogle", "Gooooooogle" are all valid long_google non-terminal.

Grouping
Parentheses can be used to indicate grouping. It means everything they wrap can be replaced with any of the valid strings that the contents of the group represent according to the rules of EBNF. For example:

```
fly = ("fire" | "fruit") "fly"
```

Here fly is either "firefly" or "fruitfly".

With BNF we could not do that in one line. It would look like the following in BNF:

```
<fly> ::= <type> "fly"
```

```
<type> ::= "fire" | "fruit"
```

Attribute Grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.

Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Example:

```
E → E + T { E.value = E.value + T.value }
```

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

```
S → ABC
```

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

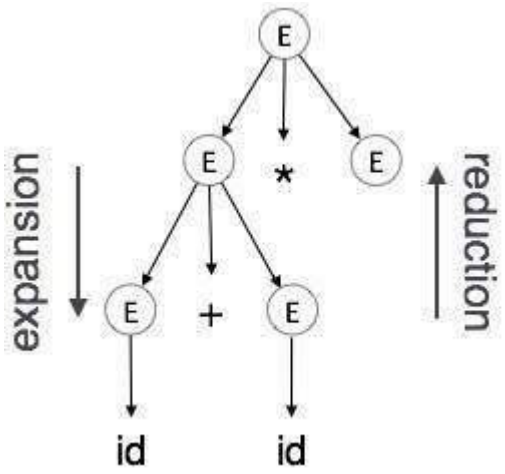
Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

```
S → ABC
```

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Expansion : When a non-terminal is expanded to terminals as per a grammatical rule



UNIT-II

Basic Data Types

The data type specifies the size and type of information the variable will store.

In this tutorial, we will focus on the most basic ones:

Data Type	Size	Description
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient f 6-7 decimal digits
double	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient f 15 decimal digits
char	1 byte	Stores a single character/letter/number, or ASCII values

---

Basic Format Specifiers

There are different format specifiers for each data type. Here are some of them:

Format Specifier	Data Type
%d or %i	int
%f or %F	float

%lf	double
%c	char
%s	Used for **strings** **(text)**, which you will learn more about in a later

Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,64
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 deci
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

---

Union Types

A union is a type whose variables may store different type values at different times during program execution.

Record vs Union

```
struct sample

{

int x;
```

```
float y;
char z;
};

union sample
{
int x;
float y;
char z;
};

union sample x;
```

Free Unions vs Discriminated Unions

C and C++ provide union constructs in which there is no language support for type checking.

The unions in these languages are called free unions, because programmers are allowed complete freedom from type checking in their use.

```
union sample
{
int a;
float b;
};
```

```
union sample myunion;

float x;



myunion.a = 27;

x = myunion.b;
```

Type checking of unions requires that each union construct include a type indicator. Such an indicator is called a **tag**, or **discriminant**, and a union with a discriminant is called a **discriminated union**.

The first language to provide discriminated unions was ALGOL 68. They are now supported by Ada, ML, Haskell, and F#.

Unions in Ada

e.g.

```
type Shape is (Circle, Triangle, Rectangle);

type Colors is (Red, Green, Blue);


type Figure (Form : Shape) is
    record
        Filled : Boolean;
        Color : Colors;
        case Form is
            when Circle =>
                Diameter : Float;
            when Triangle =>
                Left_Side : Integer;
```

```
                Right_Side : Integer;
                Angle : Float;
            when Rectangle =>
                Side_1 : Integer;
                Side_2 : Integer;
        end case;
    end record;
```

```
Figure_1 : Figure;
Figure_2 : Figure(Form => Triangle);
```

Here,

Figure_1 is unconstrained variant record.

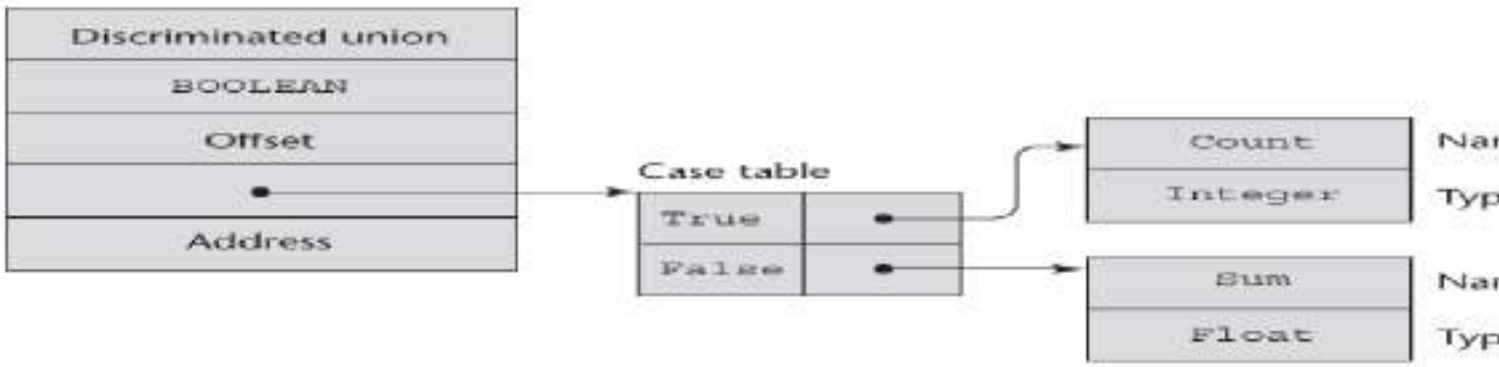
Figure_2 is constrained variant record.

Unions in F#

```
type intReal =
| IntValue of int
| RealValue of float;;
```

A compile-time descriptor for a discriminated union

```
type Node (Tag : Boolean) is
    record
        case Tag is
            when True => Count : Integer;
            when False => Sum : Float;
        end case;
    end record;
```



Creating Pointers

You learned from the previous chapter, that we can get the **memory address** of a variable with the reference operator &:

Example

```
int myAge = 43; // an int variable

printf("%d", myAge);  // Outputs the value of myAge (43)
printf("%p", &myAge); // Outputs the memory address of myAge (0x7ffe5367e044)
```

A **pointer** is a variable that **stores** the **memory address** of another variable as its value.

A **pointer variable points** to a **data type** (like int) of the same type, and is created with the * operator.

The address of the variable you are working with is assigned to the pointer:

Example

```
int myAge = 43;     // An int variable
int* ptr = &myAge;  // A pointer variable, with the name ptr, that stores the address of myAge

// Output the value of myAge (43)
printf("%d\n", myAge);

// Output the memory address of myAge (0x7ffe5367e044)
printf("%p\n", &myAge);

// Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);
```

Dereference

In the example above, we used the pointer variable to get the memory address of a variable (used together with the & **reference** operator).

You can also get the value of the variable the pointer points to, by using the * operator (the **dereference** operator):

Example

```
int myAge = 43;     // Variable declaration
int* ptr = &myAge;  // Pointer declaration

// Reference: Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);

// Dereference: Output the value of myAge with the pointer (43)
printf("%d\n", *ptr);
```

- When used in declaration (int* ptr), it creates a **pointer variable**.
- When not used in declaration, it act as a **dereference operator**.

List

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

Lists are created using square brackets:

Example

Create a List:

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

List Items

List items are ordered, changeable, and allow duplicate values.

List items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

When we say that lists are ordered, it means that the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

Changeable

The list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

Allow Duplicates

Since lists are indexed, lists can have items with the same value:

Example

Lists allow duplicate values:

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

List Length

To determine how many items a list has, use the len() function:

Example

Print the number of items in the list:

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

Try it Yourself »

List Items - Data Types

List items can be of any data type:

Example

String, int and boolean data types:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

A list can contain different data types:

Example

A list with strings, integers and boolean values:

```
list1 = ["abc", 34, True, 40, "male"]
```

Tuple

Tuples are used to store multiple items in a single variable.

Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.

A tuple is a collection which is ordered and **unchangeable**.

Tuples are written with round brackets.

Example

Create a Tuple:

```
thistuple =
("apple",
"banana",
"cherry")
print(thistuple)
```

Tuple Items

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

Since tuples are indexed, they can have items with the same value:

Example

Tuples allow duplicate values:

```
thistuple = ("apple",
"banana", "cherry", "apple",
"cherry") print(thistuple)
```

UNIT III

Generic Programming in C++

Using C++ templates, the generic programming pattern generalizes the approach so that it may be used with a variety of data types. Instead of specifying the actual data type, we supply a placeholder in templates, and that placeholder is substituted by the data type that was utilized during compilation. As a result, the compiler would generate 3 copies of the function if the template function were called for an integer, character, and

float. Because C++ uses static types, this is feasible.
The process to calculate the distance between two coordinates will stay the same regardless of whether the coordinates are provided as integer or floating-point integers, as we are aware that the technique may be the same for multiple forms of data. The int data type cannot store float values, and using the float data type for int values is a memory waste, hence in C++ programming we must write separate methods for each data type. Therefore, templates in C++ offer a generic solution to this problem.
Functions having generic types that can change their behavior depending on the data type provided during the function call are written using function templates. This makes it simpler for us to carry out the same action without code duplication on several data kinds.
Functions having generic types that can change their behavior depending on the data type provided during the function call are written using function templates. This makes it simpler for us to carry out the same action without code duplication on several data kinds.

```
# include < iostream >
// Template Function with a Type T
// This T will be changed to the data type of the argument during instantia tion.
template < class T >
T maxNum ( T x , T y ) {
```

```
  return ( a > y ? a : y ) ;

 }
int main ( )

 {
    int a = 5 , b = 2 ; float i

    = 4.5 , j = 1.3 ;

    std :: cout < < maxNum < int > ( a , b ) < < " \ n " ; std :: cout

    < < maxNum < float > ( c , d ) ;

    return 0 ;

 }
```

OUTPUT:

5

4.5

???..

Process executed in 0.11 seconds

Press any key to continue.

Generic Data Type

Classes or functions that have parameterization over a type are known as generic types. Using C++ templates, this is accomplished. A generic data type is built using the template parameters. We supply some template arguments, and the function gets them as type values, which is a similar idea to function parameters.

Operator Overloading in C++

in C++, Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning. In this article, we will further discuss about operator overloading in C++ with examples and see which operators we can or cannot overload in C++.

C++ Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile-time polymorphism. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

Example:

```
int a;
float b,sum;
sum = a + b;
```

Here, variables "a" and "b" are of types "int" and "float", which are built- in data types. Hence the addition operator '+' can easily add the contents of "a" and "b". This is because the addition operator "+" is predefined to add variables of built-in data type only.

```
1
3
// Overloading 4

// C++ Program to Demonstrate the
// working/Logic behind Operator
// Overloading

class A {

    statements;
```

```
};


int main()

{
    A a1, a2, a3;


    a3 = a1 + a2;


    return 0;

}
```

In this example, we have 3 variables “a1”, “a2” and “a3” of type “class A”. Here we are trying to add two objects “a1” and “a2”, which are of user-defined type i.e. of type “class A” using the “+” operator. This is not allowed, because the addition operator “+” is predefined to operate only on built-in data types. But here, “class A” is a user-defined type, so the compiler generates an error. This is where the concept of “Operator overloading” comes in.

Now, if the user wants to make the operator “+” add two class objects, the user has to redefine the meaning of the “+” operator such that it adds two class objects. This is done by using the concept of “Operator overloading”. So the main idea behind “Operator overloading” is to use C++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

example of Operator Overloading in C++ C++

```
1
// C++ Program to Demonstrate
```

```
2
// Operator Overloading 3

#include <iostream> 4

using namespace std; 5



6
class Complex { 7

private:

8
    int real, imag;
9



10
public:
11
    Complex(int r = 0, int i = 0)
12

    {
13
        real = r;
14
```

```
    imag = i;
15
}
16


17
// This is automatically called when '+' is used with
18
// between two Complex objects
19
Complex operator+(Complex const& obj)
20
{
21
    Complex res;
22
    res.real = real + obj.real;
23
    res.imag = imag + obj.imag;
24
    return res;
25
}
26
void print() { cout << real << " + i" << imag << '\n'; }
```

```
27
};
28



29
int main() 30
{ 31
    Complex c1(10, 5), c2(2, 4);
32
    Complex c3 = c1 + c2;
33
    c3.print();
34
 }
```

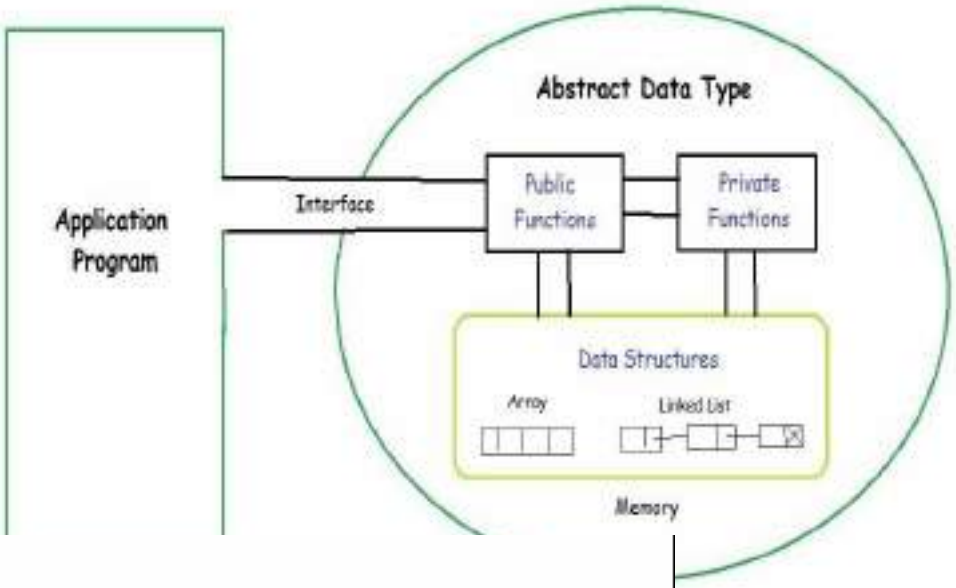
Output

```
12 + i9
```

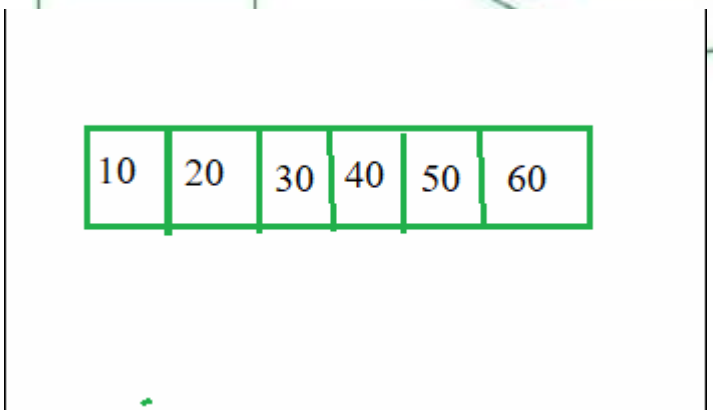
Abstract Data Types

Data types such as int, float, double, long, etc. are considered to be in- built data types and we can perform basic operations with them such as addition, subtraction, division, multiplication, etc. Now there might be a situation when we need operations for our user-defined data type which have to be defined. These operations can be defined only as and when we require them. So, in order to simplify the process of solving problems, we can create data structures along with their operations, and such data structures that are not in-built are known as Abstract Data Type (ADT).

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view.



1. Lis



Vies of list

The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.

The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list.

The List ADT Functions is given below:
get() – Return an element from the list at any given position. insert() – Insert an

element at any position of the list.

remove() – Remove the first occurrence of any element from a non- empty list.

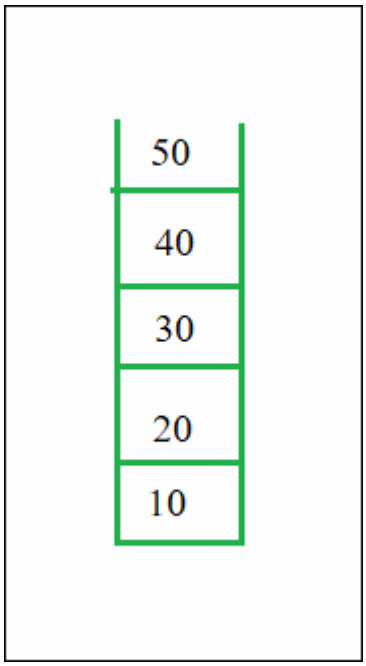
removeAt() – Remove the element at a specified location from a non- empty list.
replace() – Replace an element at any position by another element. size() – Return

the number of elements in the list.

isEmpty() – Return true if the list is empty, otherwise return false. isFull() – Return

true if the list is full, otherwise return false.

2. Stack ADT

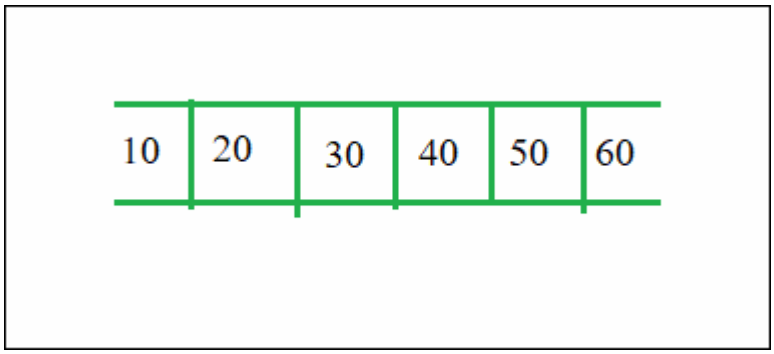


View of stack

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
The program allocates memory for the *data* and *address* is passed to the stack ADT.

The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.
push() – Insert an element at one end of the stack called top.
pop() – Remove and return the element at the top of the stack, if it is not empty.
peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
size() – Return the number of elements in the stack.
isEmpty() – Return true if the stack is empty, otherwise return false. isFull() – Return true if the stack is full, otherwise return false.

3. Queue ADT



View of Queue

The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue.
The program's responsibility is to allocate memory for storing the data.
enqueue() – Insert an element at the end of the queue.
dequeue() – Remove and return the first element of the queue, if the queue is not empty.
peek() – Return the element of the queue without removing it, if the queue is not empty.

UNIT-IV

CONCURRENCY

❖ **Concurrency:**

- It refers to the execution of multiple instruction sequences at the same time.
- It occurs in an operating system when multiple process threads are executing concurrently.
- These threads can interact with one another via shared memory or message passing.
- Concurrency results in resource sharing, which causes issues like deadlocks and resource scarcity.
- It aids with techniques such as process coordination, memory allocation,

and execution schedule to maximize throughput.

Levels of concurrency:

Concurrency is naturally divided into instruction level, statement level(executing two or more statements simultaneously), program unit level(execute two or more subprogram units simultaneously) and program level(executing two or more programs simultaneously).

1) sub program level concurrency:

- concurrency, at the subprogram level, is a programming technique that enables multiple tasks to be executed concurrently within a single program.
- Unlike traditional sequential execution, where statements are executed one after the other, subprogram level concurrency allows different parts of a program to run independently and simultaneously.
- This article explores the concept of subprogram level concurrency, its benefits, implementation methods, and common use cases.

- Subprogram level concurrency involves breaking down a program into smaller subprograms or tasks that can run concurrently.
- Each subprogram operates independently of the others and may share data or communicate with each other as needed.
- This approach allows a program to perform multiple tasks concurrently, leveraging the available computing resources more effectively

Parallel Processing

Benefits of Subprogram Level Concurrency:

1. **Improved Performanc**

ReResponsiveness:

Implementation of Subprogram Level Concurrency:

Subprogram level concurrency can be achieved through various mechanisms, depending on the programming language and the underlying platform. Some common approaches include:

1. Threads: Threads are lightweight execution units that run within the same process and share the same memory space. Each thread can execute a separate subprogram concurrently.

Example of Using Threads in C++:

```cpp
#include <iostream>
#include <thread>

void task1() {
    std::cout << "Task 1 executed concurrently" << std::endl;
}

void task2() {
    std::cout << "Task 2 executed concurrently" << std::endl;
}

int main() {
    std::thread t1(task1);
    std::thread t2(task2);

    t1.join();
    t2.join();

    return 0;
}
```

2. Statement level concurrency:

- Two or more high-level statements running simultaneously
- Statement level concurrency is largely a matter of specifying how data should be distributed over multiple memories and which statements can be executed concurrently.
- HPF(High Performance Fortran) is a collection of extensions to Fortran90 that are meant to allow programmers to specify information to the compiler to help it optimize the execution of programs on multiprocessor computers.
- User can specify the number of processors, the distribution of data over the memories of those processors, these can be used to inform the compiler of ways it can map the program onto a multiprocessor architecture.

3) Instruction level – 2 or more machine instructions running simultaneously

4) program level – 2 or more programs running simultaneously instruction and program level concurrency involve no language issues so we won't consider them and instead concentrate on the other two levels instruction and program level concurrency typically require parallel processing while statement and unit level merely require multiprocessing

Synchronization is a mechanism that controls the order in which tasks execute. Two kinds of synchronization are required when tasks share data, cooperation and competition. Cooperation synchronization is required between task A and B when task A must wait for task B to complete some specific activity before task A can continue its execution. Competition synchronization is required between two tasks when both require the use of some resource that can't be simultaneously used.

synchronization methods:

1. semaphores

2. monitors
3. message passing

1. semaphores:

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

The definitions of wait and signal are as follows −

- **Wait**

  The wait operation decrements the value of its argument **S**, if it is positive. If **S** is negative or zero, then no operation is performed.

```
wait(S)
{
   while (S<=0);

   S--;
}
```

- **Signal**

  The signal operation increments the value of its argument **S**.

```
signal(S)
{
   S++;
}
```

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows −

- **Counting Semaphores**

These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

- **Binary Semaphores**

  The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores

Some of the advantages of semaphores are as follows −

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

2. MONITORS:

Monitors are a synchronization tool used in process synchronization to manage access to shared resources and coordinate the actions of numerous threads or processes. When opposed to low-level primitives like locks or semaphores, they offer a higher-level abstraction for managing concurrency. Let's examine monitors to see what they are, why they are utilized, and how process synchronization uses them −

Monitors Implementation:

- Locks, semaphores, or atomic operations are examples of synchronization primitives that can be used to build monitors.
- A lock or mutex connected to a monitor ensures mutual exclusion. The monitor can only be accessed by the thread or process holding the lock.
- Condition variables are employed within the monitor to control synchronization and communication.
- Using condition variables, threads or processes examine conditions and wait for the conditions to become true.
- When a modification to the shared resource transforms the condition into true and frees up waiting threads or processes, signalling or notification takes place.

3. Message Passing:

It provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.

For example − chat programs on World Wide Web. Now let us
discuss the message passing step by step.

Step 1 − Message passing provides two operations which are as follows −

- Send message
- Receive message

Messages sent by a process can be either fixed or variable size.

Step 2 − For fixed size messages the system level implementation is straight forward. It makes the task of programming more difficult.

Step 3 − The variable sized messages require a more system level implementation but the programming task becomes simpler.

Step 4 − If process P1 and P2 want to communicate they need to send a message to and receive a message from each other that means here a communication link exists between them.

Step 5 − Methods for logically implementing a link and the send() and receive() operations.

- **Exception Handling** :
  - Exception is a run time error. An exception is an error event that can happen during the execution of a program and disrupts its normal flow.
  - The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

Types of Exceptions: Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions. Exceptions can be categorized in two ways:

Exceptions can be categorized in two ways:

1. **Built-in Exceptions**

a).Checked Exception b). Unchecked Exception

2. **User-Defined Exceptions**

1. **Built-in Exceptions**: • Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

a**). Checked Exceptions:** • Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the comp

b). **Unchecked Exceptions**: • unchecked exceptions are called run time exception. The compiler will not check these exceptions at compile time

2. **User-Defined Exceptions**: • Sometimes, the built-in exceptions in Java are not able to describe a certain situation. • In such cases, users can also create exceptions, which are called 'userdefined Exceptions'.

Advantages of Exception Handling:

• Provision to Complete Program Execution

• Easy Identification of Program Code and Error-Handling Code

• Propagation of Errors

• Meaningful Error Reporting

• Identifying Error

Java Exception Handling Key words

. 1. try...catch block

2. finally block

3. throw and throws keyword

1. **Java try...catch block:** The try-catch block is used to handle exceptions in Java.

Syntax of try...catch block: try {

```
// code
}
catch(Exception e)
```

{ // code

```
}
```

Example:

```
class Main
{
public static void main(String[] args)
{
```

```
Try
{
intdivideByZero = 5 / 0; System.out.println("Rest of code
in try block");
}
catch (ArithmeticException e)
{
System.out.println("ArithmeticException => " + e.getMessage());
}
}
}
```

Output: ArithmeticException => / by zero

2. Java finally block:

- The finally block is always executed no matter whether there is an exception or not. is optional.
- And, for each try block, there can be only one finally block.

Syntax:

```
Try
{
//code }
catch (ExceptionType1 e1)
{
// catch block
}
finally
{
// finally block always executes
}
```

Example:

```
class Main {
public static void main(String[] args) { try {
intdivideByZero = 5 / 0;
    }
catch (ArithmeticException e) { System.out.println("ArithmeticException => " +
e.getMessage());
    }
finally {
System.out.println("This is the finally block");
    }
    }
    }
```

Output:

```
ArithmeticException => / by zero This is the
finally block
```

3. Java throw and throws keyword:

- The Java throw keyword is used to explicitly throw a single exception.
- When we throw an exception, the flow of the program moves from the try block to the catch block.

Example:

```
class Main {
public static void divideByZero() {
throw new ArithmeticException("Trying to divide by 0");
    }
public static void main(String[] args) { divideByZero();
    }
```

```
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0
atMain.divideByZero(Main.java:5) atMain.main(Main.java:9)
```

In the above example, we are explicitly throwing the ArithmeticException using the throw keyword.

Throws keyword:

- The throws keyword is used to declare the type of exceptions that might occur within the method.
- It is used in the method declaration.

Example: Java throws keyword

```
import java.io.*;
class Main {
public static void findFile() throws IOException { File newFile =
new File("test.txt");
FileInputStream stream = new FileInputStream(newFile);
}
public static void main(String[] args) { try {
findFile();
}
catch (IOException e) {
System.out.println(e);
}
}
}
```

Output:

java.io.FileNotFoundException: test.txt (The system cannot find the file specified)

C++ Exception Handling:

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.

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

You
block raises more than one exception in different situations.

Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement 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.

Example

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

Catching Exceptions:

```
double division(int a, int b) { if( b
   == 0 ) {
      throw "Division by zero condition!";
try {
   // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

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.

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

```
try {
   // protected code
```

```
} catch(...) {
  // code to handle any exception
}
```

Example

The following is an e

which throws a division by zero exception and we catch it in catch block.

Open Compiler #include

```
tream> using namespace


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) { cerr <<
     msg << endl;
   }
```

```
   return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result –

Exception Handling in Ada:

- Ada is a high-level programming language designed for safety-critical systems, and it provides a robust exception handling mechanism for dealing with errors and exceptions.
- Exception handling in Ada is based on the principle of raising and handling exceptions, and it provides a mechanism for gracefully recovering from errors and returning to a safe state.
- In Ada, exceptions can be raised explicitly using the raise statement or implicitly by the language or the runtime environment.
- When an exception is raised, the control is transferred to the nearest handler that matches the exception type. I
- f there is no matching handler, the exception is propagated up the call stack until it is caught or the program terminates.
- Exception handling in Ada consists of three main components: raising an exception, handling an exception, and defining an exception.

Raising an exception: In Ada, you can raise an exception explicitly using the raise statement. The raise statement takes an exception object or a predefined exception and transfers control to the nearest handler that matches the exception type.

The syntax for raising an exception is as follows: python

raise exception_name;

Handling an exception: In Ada, you can handle exceptions using the try-catch block. The try block contains the code that may raise an exception, and the catch block contains the code that handles the exception. The catch block is executed only if an exception is raised in the try block. The syntax for handling an exception is as follows:

begin

-- Code that may raise an exception exception

when exception_name =>

-- Code that handles the exception end;

Defining an exception: In Ada, you can define your own exceptions using the exception declaration. An exception declaration specifies the name and the type of the exception. You can also specify a message associated with the exception that can be displayed when the exception is raised.

The syntax for defining an exception is as follows:

exception exception_name is

exception_type;

In Ada, predefined exceptions are available for common errors such as arithmetic overflow, range check failure, and program termination. These exceptions can be caught and handled in the same way as user-defined exceptions.

Event:

- Change in the state of an object is known as event i.e. event describes the change in state of source.
- Events are generated as result of user interaction with the graphical user interface components.

- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event:

The events can be broadly classified into two categories:

1. Foreground Events: Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

3. **Background Events**: Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

Event Handling:

- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- This mechanism have the code which is known as event handler that is executed when an event occurs.
- Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view

the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

Event Handling Example:

Create the following java program using any editor of your choice in say **D:/ > AWT > com > tutorialspoint > gui >**

AwtControlDemo.java

```
package com.tutorialspoint.gui;
import java.awt.*;
import java.awt.event.*;
public class AwtControlDemo { private
   Frame mainFrame; private Label
   headerLabel; private Label
   statusLabel; private Panel
   controlPanel; public
   AwtControlDemo(){
      prepareGUI();
   }
   public static void main(String[] args){
      AwtControlDemo  awtControlDemo = new AwtControlDemo();
      awtControlDemo.showEventDemo();
   }
```

```
private void prepareGUI(){
   mainFrame = new Frame("Java AWT Examples");
   mainFrame.setSize(400,400); mainFrame.setLayout(new
   GridLayout(3, 1));
   mainFrame.addWindowListener(new WindowAdapter() { public void
      windowClosing(WindowEvent windowEvent){
         System.exit(0);
      }
   });
   headerLabel = new Label();
   headerLabel.setAlignment(Label.CENTER); statusLabel
   = new Label();
   statusLabel.setAlignment(Label.CENTER);
   statusLabel.setSize(350,100);
   controlPanel = new Panel(); controlPanel.setLayout(new
  FlowLayout()); mainFrame.add(headerLabel);
  mainFrame.add(controlPanel);
  mainFrame.add(statusLabel); mainFrame.setVisible(true);
}
private void showEventDemo(){ headerLabel.setText("Control in
   action: Button"); Button okButton = new Button("OK");
   Button submitButton = new Button("Submit"); Button
   cancelButton = new Button("Cancel");

   okButton.setActionCommand("OK"); submitButton.setActionCommand("Submit");
```

```
      cancelButton.setActionCommand("Cancel");
      okButton.addActionListener(new ButtonClickListener());
      submitButton.addActionListener(new ButtonClickListener());
      cancelButton.addActionListener(new ButtonClickListener());
      controlPanel.add(okButton);
      controlPanel.add(submitButton);
      controlPanel.add(cancelButton);
      mainFrame.setVisible(true);
   }
   private class ButtonClickListener implements ActionListener{ public void
      actionPerformed(ActionEvent e) {
         String command = e.getActionCommand(); if(
         command.equals( "OK" ))  {
            statusLabel.setText("Ok Button clicked.");
         }
         else if( command.equals( "Submit" ) )  { statusLabel.setText("Submit
            Button clicked.");
         }
         else  {
            statusLabel.setText("Cancel Button clicked.");
         }
      }
   }
}
```

Compile the program using command prompt. Go to **D:/ > AWT** and type the following command.

```
D:\AWT>javac com\tutorialspoint\gui\AwtControlDemo.java
```

If no error comes that means compilation is successful. Run the program using following command.

```
D:\AWT>java com.tutorialspoint.gui.AwtControlDemo
```

Verify the following output

C# - Events:

- An event is a notification sent by an object to signal the occurrence of an action.
- Events in .NET follow the observer design pattern.
- The class who raises events is called Publisher, and the class who receives the notification is called Subscriber.
- There can be multiple subscribers of a single event.
- Typically, a publisher raises an event when some action occurred.
- The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it.

Declare an Event:

An event can be declared in two steps:

1. Declare a delegate.
2. Declare a variable of the delegate with event keyword.

The following example shows how to declare an event in publisher class.

Example: Declaring an Event

```
public delegate void Notify();  // delegate
```

```
public class ProcessBusinessLogic
{
    public event Notify ProcessCompleted; // event
}
```

- In the above example, we declared a delegate Notify and then declared an event ProcessCompleted of delegate type Notify using "event" keyword in the ProcessBusinessLogic class.
- Thus, the ProcessBusinessLogic class is called the publisher. The Notify delegate specifies the signature for the ProcessCompleted event handler.
- It specifies that the event handler method in subscriber class must have a void return type and no parameters.

Example: Raising an Event

```
public delegate void Notify();  // delegate public class
ProcessBusinessLogic
{
    public event Notify ProcessCompleted; // event public void
    StartProcess()
    {
        Console.WriteLine("Process Started!");
        // some code here.. OnProcessCompleted();
    }
    protected virtual void OnProcessCompleted() //protected virtual method
    {
        //if ProcessCompleted is not null then call delegate ProcessCompleted?.Invoke();
    }
}
```

Above, the StartProcess() method calls the method onProcessCompleted() at the end, which raises an event.

- Typically, to raise an event, protected and virtual method should be defined with the name On<EventName>.
- Protected and virtual enable derived classes to override the logic for raising the event.
- However, A derived class should always call the On<EventName> method of the base class to ensure that registered delegates receive the event.
- The OnProcessCompleted() method invokes the delegate using ProcessCompleted?.Invoke();.
- This will call all the event handler methods registered with the ProcessCompleted event.
- The subscriber class must register to ProcessCompleted event and handle it with the method whose signature matches Notify delegate, as shown below.

Example: Consume an Event

```
class Program
{
    public static void Main()
    {
        ProcessBusinessLogic bl = new ProcessBusinessLogic(); bl.ProcessCompleted +=
        bl_ProcessCompleted; // register with an event bl.StartProcess();
    }
    // event handler
    public static void bl_ProcessCompleted()
    {
```

```
        Console.WriteLine("Process Completed!");
    }
}
```

Above, the Program class is a subscriber of the ProcessCompleted event. It registers with the event using += operator. Remember, this is the same way we add methods in the invocation list of multicast delegate. The bl_ProcessCompleted() method handles the event because it matches the signature of the Notify delegate.

❖ **Thread:**

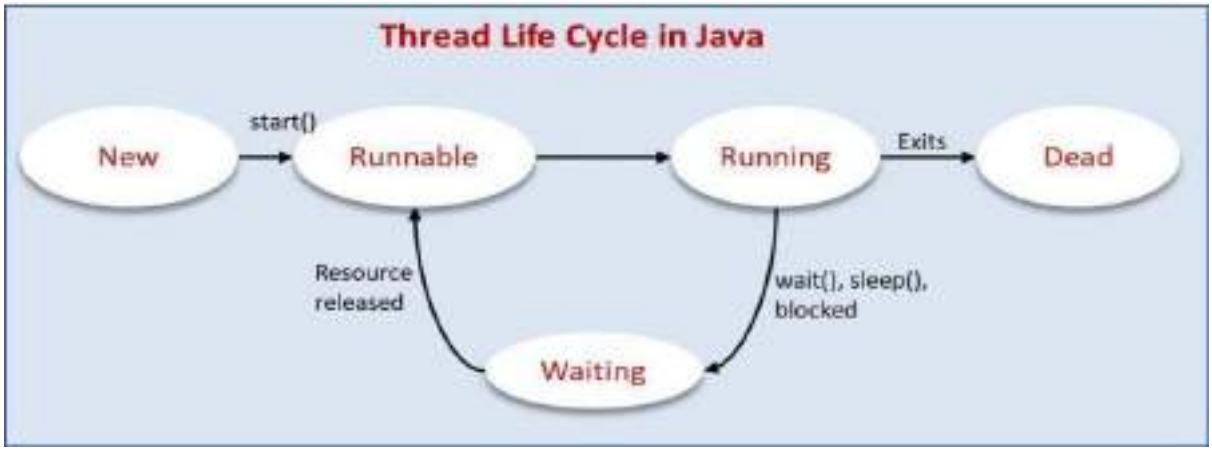
• A thread is a lightweight sub process, the smallest unit of processing. It is a separate path of execution.

• Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

Life cycle of a Thread:

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated



1. New:

• Whenever a new thread is created, it is always in the new state.

• For a thread in the new state, the code has not been run yet and thus has not begun its execution.

2. Runnable:

• A thread, that is ready to run is then moved to the runnable state.

• In the runnable state, the thread may be running or may be ready to run at any given instant of time.

3. Running:

• When the thread gets the CPU, it moves from the runnable to the running state.

• Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

4. Blocked or Waiting:

• Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

5. Terminated or Dead:

A thread reaches the termination state because of the following reasons:

1. When a thread has finished its job, then it exists or terminates normally.

2. It occurs when some unusual events such as an unhandled exception or segmentation fault.

There are the following two ways to create a thread:

1. By Extending Thread Class
2. By Implementing Runnable Interface

1. By Extending Thread Class:

- The simplest way to create a thread in Java is by extending the Thread class and overriding its run() method.
- Thread class provide constructors and methods to create and perform operations on a thread.

- Thread class extends Object class and implements Runnable interface.

2. By Implementing Runnable Interface:

- Another approach to creating threads in Java is by implementing the Runnable interface.
- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
- Runnable interface have only one method named run(). This approach is preferred when we want to separate the task from the thread itself, promoting better encapsulation and flexibility.

public void run(): is used to perform action for a thread.

Starting a Thread

The **start() method** of the Thread class is used to start a newly created thread. It performs the following tasks:

- A new thread starts (with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Thread Creation

1) Creating Thread by Extending Thread Class

```
class Multi extends Thread{
public void run(){ System.out.println("thread is
running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

OUTPUT:

thread is running...

2) Java Thread Example by implementing Runnable interface class Multi3

implements Runnable{

public void run(){ System.out.println("thread is running...");

}

public static void main(String args[]){ Multi3 m1=**new**

Multi3();

Thread t1 =**new** Thread(m1); // Using the constructor Thread(Runnable r) t1.start();

}

}

Output:

thread is running...

UNIT-V

Functional programming languages

❖ Functional programming languages:

- Functional programming languages are specially designed to handle symbolic computation and list processing applications.
- Functional programming is based on mathematical functions.
- Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

Functional programming languages are categorized into two groups,

1. **Pure Functional Languages** :These types of functional languages support only the functional paradigms. For example − Haskell.
2. **Impure Functional Languages**:These types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

Advantages:

Functional programming offers the following advantages

- **Bugs-Free Code** − Functional programming does not support **state**, so

there are no side-effect results and we can write error-free codes.

- **Efficient Parallel Programming** − Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- **Efficiency** − Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- **Supports Nested Functions** − Functional programming supports Nested Functions.
- **Lazy Evaluation** − Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

Introduction to LISP:

- Lisp is a programming language that has an overall style that is organized around expressions and functions.
- Every Lisp procedure is a function, and when called, it returns a data object as its value.
- It is also commonly referred to as “functions” even though they may have side effects.
- Lisp is the second-oldest high-level programming language in the world which is invented by John McCarthy in the year 1958 at the Massachusetts Institute of Technology.

Features of LISP Programming Language:

1. It is a machine-independent language
2. It uses iterative design methodology and is easily extensible
3. It allows us to create and update the programs and applications dynamically.
4. It provides high-level debugging.
5. It supports object-oriented programming.
6. It supports all kinds of data types like objects, structures, lists, vectors, adjustable arrays, set, trees, hash-tables, and symbols.
7. It is an expression-based language
8. It can support different decision-making statements like if, when, case, and Cond
9. It will also support different iterating statements like do, loop,loopfor, do times and do list.
10. It will support input and output functions
11. By using lisp we can also create our own functions

These are the features of LISP Programming.

Hello World program in LISP:

we can start writing a string by using the write-line method

Syntax:

```
(write-line string)
```

Example:

- Lisp

```
;this is a comment
(write-line "Hello Geeks")
```

Output:

```
Hello Geeks
```

Naming Conventions:

- The naming Conventions mean the way we are declaring variables in a program.
- A variable can contain any number of alphanumeric characters other than whitespace, open and closing parentheses

Example:

Acceptable – hello,saisravan, etc

Not Acceptable – hell()0,sat{ sravab{,,,,,,,,etc

- A variable can not contain double and single quotes, backslash, comma, colon, semicolon, and vertical bar.

Example:

Acceptable – hello,saisravan, etc

Not Acceptable – hell””)0,sat//*& sra//>vab{,,,,,,,,etc

- A variable can not start with a digit but. it can contain any number of digits

Example:

Acceptable – hello88Geeks, r56lisp, ,,,,,etc Not

Acceptable – 40geeks,4klll,….etc **Example Code:**

For acceptable names

- Lisp

```
;acceptable naming conventions
(write-line "hello")
(terpri)
;acceptable naming conventions
(write-line "hello99")
(terpri)
;acceptable naming conventions
(write-line "hello geeks")
(terpri)
;acceptable naming conventions
(write-line "hello_Geek")
(terpri)
;acceptable naming conventions
(write-line "hello123")
```

Output:

```
hello hello99
hello geeks
hello_Geek
hello123
```

Imperative Languages:

- Imperative languages are those which facilitate the computation by mean of state changes.
- By a state, it means the condition of a computer's random access memory (RAM) or storage.
- It is helpful to think of computer memory as a sequence of snapshots, each one capturing the values in all memory cells at a particular time.
- Each snapshot records a state.

- When a program is entered, associated data exists in a certain condition, say an unsorted list off-line.
- It is the programmer's job to specify a sequence of changes to the store that will produce the desired final state, perhaps a sorted list.
- The store involves much more than data and a stored program.
- It includes a symbol table, run-time stack (S), an operating system, and its CPU itself can be viewed as part of the initial state.

Functional Languages:

- A functional language is a programming language built over and around logical functions or procedures within its programming structure.
- It is dependent on and is equivalent to mathematical functions in its program flow.
- Functional languages change their basic structure from the numerical structure of Lambda calculus and combinatory logic.
- Erlang, LISP, Haskell, and Scala are the most famous functional languages.

Imperative Languages	**Functional Languages**
Imperative languages are based on Von-Neumann Architecture.	The functional languages are not based on Von-Neumann Architecture.
The programmer is concerned with the management of variables and the assignment of values to them.	The programmer requires not to be concerned with variables because memory cells need not be abstracted into the language.
The imperative languages facilitate the computation using the state changes.	The functional languages facilitate the functions that the program represents, instead of only stating changes as the

Imperative Languages	Functional Languages
	program executes, statement by statement.
It can increase the efficiency of execution.	It can decrease the efficiency of execution.
It is used for the laborious construction of programs.	Less labor required than programming in an imperative language.
It is a very clean syntactic framework.	It is a much more complex syntactic structure than imperative language.
Concurrent execution is difficult to design and use.	Concurrent execution is easy to design and use.
The semantics are difficult to understand.	The semantics are simple as compared with imperative languages.
The programmer should create a static division of the program into its concurrent elements, which are then written as functions. This can be a complex process.	Programs in functional languages can be broken into concurrent elements dynamically by the execution system, creating the process highly flexible to the hardware on which it is running.
An example of imperative languages includes C, C++, ADA, Pascal, etc.	An example of functional languages includes LISP, ML, scheme, etc.

Functional programming:

- A logic programming language is a type of programming language that uses logic to express rules and facts about a problem domain.

- Logic programming languages use logical operators and inference rules to derive new conditions based on known true conditions.

Some examples of logic programming languages include:

- **Prolog**: Uses logical clauses to express rules and regulations

Functional Programming	**Logical Programming**
It is totally based on functions.	It is totally based on formal logic.
In this programming paradigm, programs are constructed by applying and composing functions.	In this programming paradigm, program statements usually express or represent facts and rules related to problems within a system of formal logic.
These are specially designed to manage and handle symbolic computation and list processing applications.	These are specially designed for fault diagnosis, natural language processing, planning, and machine learning.
Its main aim is to reduce side effects that are accomplished by isolating them from the rest of the software code.	Its main aim is to allow machines to reason because it is very useful for representing knowledge.
Some languages used in functional programming include Clojure, Wolfram Language, Erland, OCaml, etc.	Some languages used for logic programming include Absys, Cycl, Alice, ALF (Algebraic logic functional programming language), etc.

Functional Programming	Logical Programming
It reduces code redundancy, improves modularity, solves complex problems, increases maintainability, etc.	It is data-driven, array-oriented, used to express knowledge, etc.
It usually supports the functional programming paradigm.	It usually supports the logic programming paradigm.
Testing is much easier as compared to logical programming.	Testing is comparatively more difficult as compared to functional programming.
It simply uses functions.	It simply uses predicates. Here, the predicate is not a function i.e., it does not have a return

P**rolog:**

- Prolog or Programming in Logic's is a logical and declarative programming language.
- It is one major example of the fourth generation language that supports the declarative programming paradigm.
- This is particularly suitable for programs that involve symbolic or non-numeric computation.
- This is the main reason to use Prolog as the programming language in Artificial Intelligence, where symbol manipulation and inference manipulation are the fundamental tasks.

- In Prolog, we need not mention the way how one problem can be solved, we just need to mention what the problem is, so that Prolog automatically solves it.
- However, in Prolog we are supposed to give clues as the solution method.

Prolog language basically has three different elements −

Facts − The fact is predicate that is true, for example, if we say, “Tom is the son of Jack”, then this is a fact.

Rules − Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as − grandfather(X, Y) :- father(X, Z), parent(Z, Y)

This implies that for X to be the grandfather of Y, Z should be a parent of Y and X should be father of Z.

Questions − And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

Syntax and Basic Fields :

In prolog, We declare some facts. These facts constitute the Knowledge Base of the system. We can query against the Knowledge Base. We get output as affirmative if our query is already in the knowledge Base or it is implied by Knowledge Base, otherwise we get output as negative. So, Knowledge Base can be considered similar to database, against which we can query. Prolog facts are expressed in definite pattern. Facts contain entities and their relation. Entities are written within the parenthesis separated by comma (,). Their relation is expressed at the start and outside the parenthesis. Every fact/rule ends with a dot (.). So, a typical prolog fact goes as follows :

Format : relation(entity1, entity2, ... k'th entity).

Example : friends(raju,
mahesh). singer(sonu).
odd_number(5).

Explanation :

These facts can be interpreted as : raju and mahesh are friends.

sonu is a singer.

5 is an odd number.

Runningqueries :

A typical prolog query can be asked as :

Query 1 : ?- singer(sonu).

Output : Yes.

Explanation : As our knowledge base contains the above fact, so output was 'Yes', otherwise it would have been 'No'.

Query 2 : ?- odd_number(7).

Output : No.

Explanation : As our knowledge base does not contain the above fact, so output was 'No'.

Applications of Prolog:

The applications of prolog are as follows:

- Specification Language
- Robot Planning
- Natural language understanding
- Machine Learning
- Problem Solving
- Intelligent Database retrieval
- Expert System
- Automated Reasoning

Scripting Language:

- A script or scripting language is a computer language that does not need the compilation step and is rather interpreted one by one at runtime.
- It is where the script is written and where instructions for a run-time environment are written.
- In contrast to programming languages that are compiled first before running, scripting languages do not compile the file and execute the file without being compiled.
- Scripts are often utilized to create dynamic web applications nowadays because they are linked to web development.
- Server-Side Scripting Languages and Client-Side Scripting Languages are the two types of scripting languages.
- Python, PHP, and Perl are examples of server-side scripting languages, while JavaScript is the greatest example of a client-side scripting language.
- These languages are often developed with the goal of communicating with other programming languages.

There are multiple scripting languages available some are as follows:

- **Bash:** It is a scripting language that's the default command interpreter on most GNU/Linux systems and can be found on a variety of operating systems. As compared to other programming languages, the use of bash is much easier to create scripts .It stores documentation for others to use, defines the tools to use and command line code, and provides useful reusable scripts. Its name is short for 'Bourne-Again Shell'.
- **Ruby:** It is a scripting and pure object-oriented programming language that enables developers to create innovative software. It was established in 1993 by Yukihiro Matsumoto of Japan and is excellent for web development. Ruby offers the same features that are included in the languages such as Python, Perl, and Smalltalk.

- **Node js:** Writing network applications in JavaScript is open-source and cross-platform. It is not a programming language that reads and writes files on a computer/server and handles networking, but it does employ JavaScript as the core programming interface. For real-time web applications, corporate users of Node.js include Yahoo, Netflix, PayPal, IBM, Microsoft, and LinkedIn.

- **Python:** It is an object-oriented programming language that is the most widely used language among developers, in modern times. It is simple and interpreted. It's a dynamically semantic language with enormous scripted lines of code. It has high-level data structures built in, making it easy to use and suitable for Rapid Application Development. It supports code reuse and software modularity by allowing modules and packages.
- **Perl:** It is a scripting as well as dynamic programming language with innovative features. These features make it popular and different from other languages. It is available on all Linux and Windows servers, which was developed by Larry Wall in 1987. Although it has no official Full form, the most used expended form is "Practical Extraction and Reporting Language. High-traffic websites widely use Perl, including IMDB, priceline.com, and it also helps in text manipulation tasks.