REVATURE

Table Of Contents

Programming Concepts

OOP

SDLC

Programming Concepts

What is Programming?

A programming methodology deals with:

- Analyzing issues utilizing algorithms developed through current programming approaches,
- developing programs using relevant languages, and
- Bringing these programs into action on a suitable platform.

Programmer

Who is a Programmer?

A programmer is someone who writes computer software or applications by providing specific instructions to the computer. They have a wide knowledge of computing and coding, working with various programming languages and platforms,

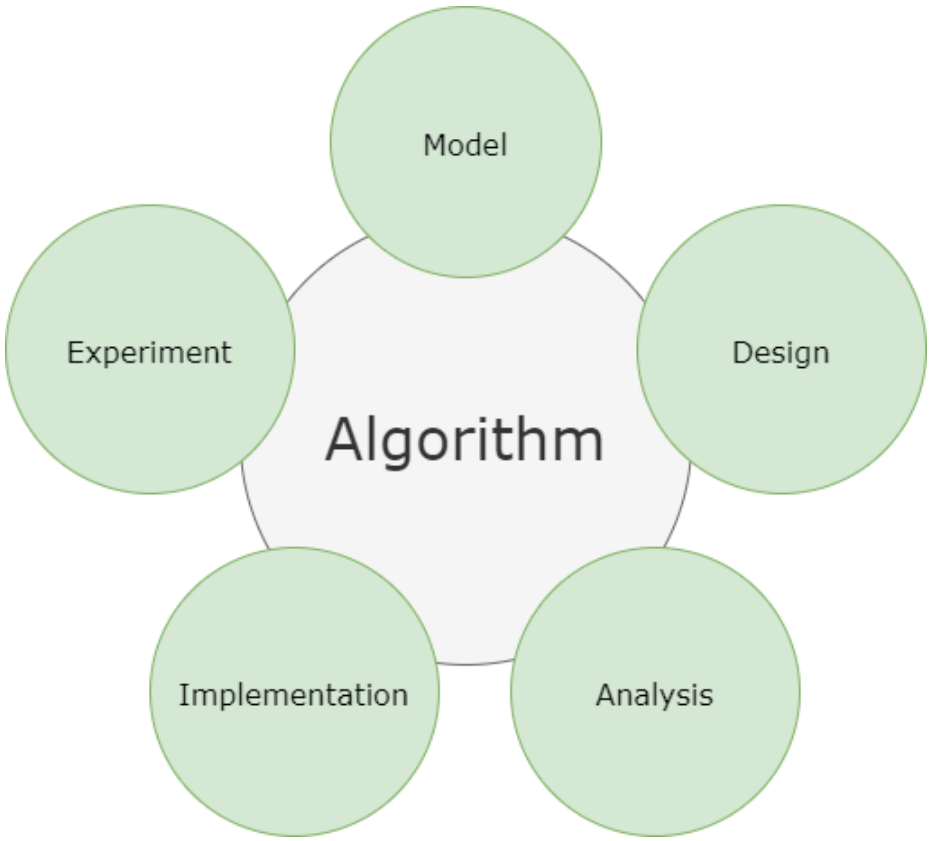
Programmers can specialize in different areas of computing, such as database management, security, software/firmware development, mobile app development, and web development. They play a crucial role in advancing computer technology and the overall field of computing.

Algorithm

What is an Algorithm?

An algorithm is a set of step-by-step instructions that must be followed in a specific order to achieve a desired outcome. Algorithms are created independently of programming languages, so they can be implemented in multiple languages.

Important characteristics of an algorithm include being **clear**, **precise**, **efficient**, and not reliant on a particular programming language. The significance of an algorithm is determined by its scalability and performance.



Pseudocode

Pseudocode (pronounced SOO-doh-kohd) is a detailed yet readable description of what a computer program or algorithm must do, expressed in a formally styled natural language rather than in a programming language. It allows designers or lead programmers to express the design in detail and provides programmers with a detailed template for the next step of writing code in a specific programming language.

Elements of Pseudocode

There are three basic elements of pseudocode:

Sequence: There is an assumption here that the code will run in order, stepwise, from top to bottom.

Selection or Flow Control: At some points during execution, the flow may need to be diverted to some other point based on some conditions.

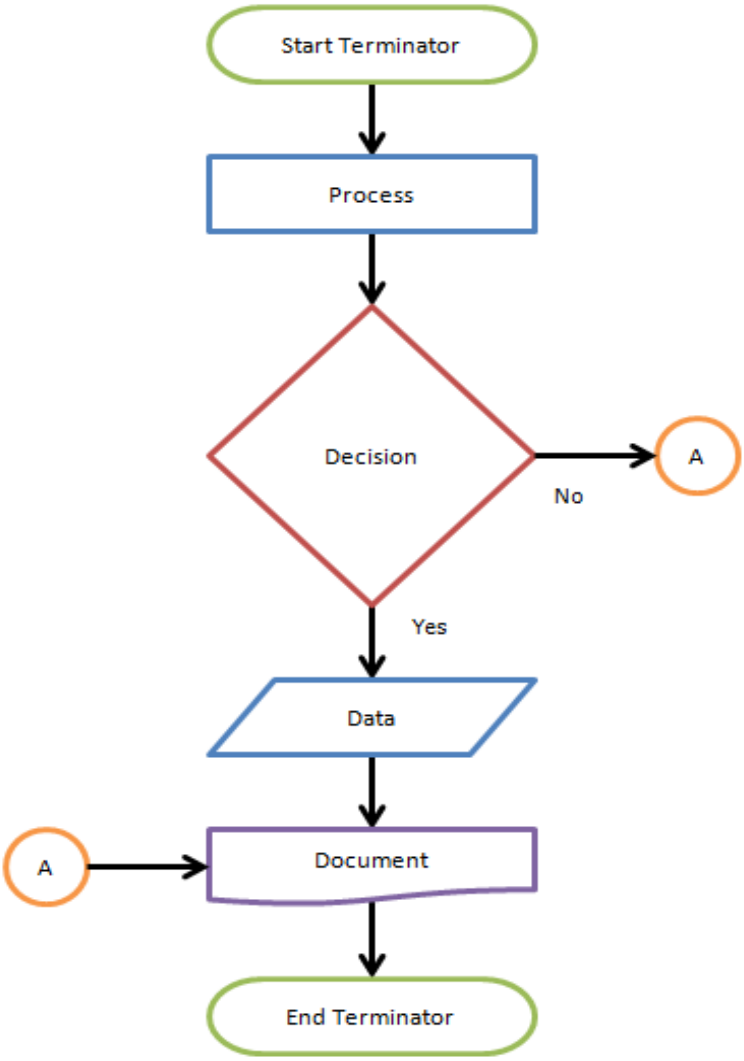
Iteration or Looping: In computer programming, iteration and looping refer to the repetition of a set of instructions or a block of code. It allows you to execute a specific section of code multiple times, either for a fixed number of iterations or until a certain condition is met.

Suggestions to Write Pseudocode:

1. Begin with writing down what is the purpose of the process.
2. Start with “BEGIN,” end with “END,” and always capitalize the initial word.
3. Have only one statement per line.
4. Organize and indent sections of pseudocode properly (for clarity of decision control and execution mechanism and readability). Indent to show hierarchy, improve readability, and show nested constructs.
5. Always end multi-line sections using any of the END keywords like “ENDIF,” “ENDWHILE,” etc.,

Flowchart

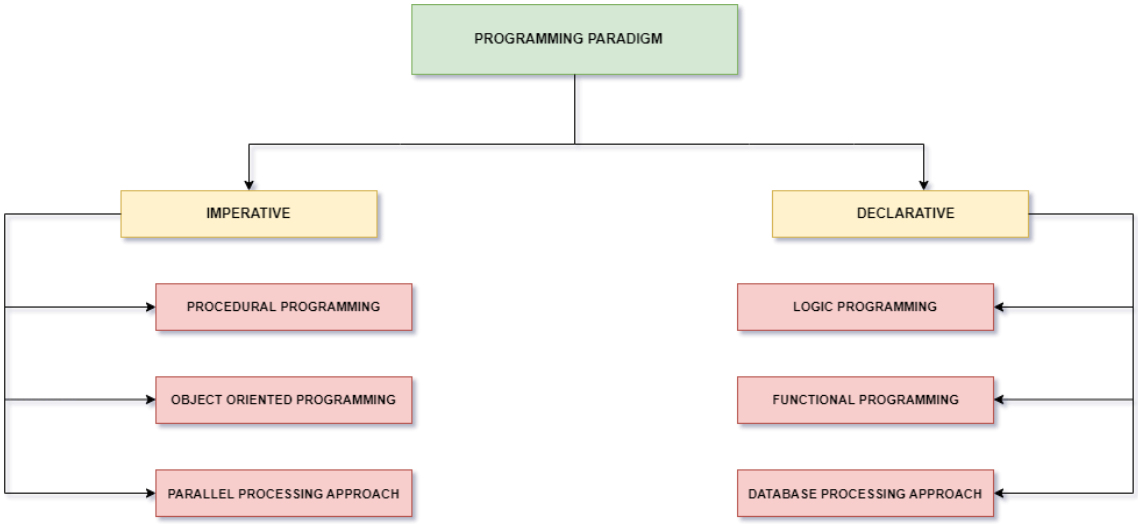
The term flowchart or flow chart are used for diagrams that depict processes, workflow, programs, and computer algorithms using a set of symbols that have pre-defined functions. This diagrammatic representation is also known by other terms such as a flow sheet, flow diagram, workflow diagram, and business flow diagram.



Programming Paradigms

Programming paradigms refer to the fundamental styles and approaches used in programming languages. Each paradigm has its own set of rules, concepts, and practices that govern how programmers write code.

The most common programming paradigms include imperative, functional, object-oriented, logic, and procedural programming.



Imperative Programming Paradigm

Imperative programming is a paradigm where the programmer specifies how the program should execute by giving step-by-step instructions. It focuses on changing the state of the program through statements such as loops, conditionals, and assignments.

Procedural Programming Paradigm

Procedural programming is a paradigm where the focus is on dividing the program into smaller procedures or functions. Programs are organized around these procedures, which can be called in a specific order to execute the program.

Object-Oriented Programming Paradigm

Object-oriented programming is a paradigm where the focus is on modeling real-world objects and their interactions. Programs are organized into classes and objects, which have properties and methods that define their behavior.

Parallel processing approach

Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system possesses many numbers of processors with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer.

Logic Programming

One of the main advantages of logic programming is its ability to manage complex problems involving enormous amounts of data. Programs written in this paradigm consist of a set of rules and facts that are used to derive current information. However, it can also be difficult to understand and debug, especially for programmers who are not familiar with the logic-based approach.

Functional Programming Paradigm

Functional programming is a paradigm where the focus is on the evaluation of functions rather than changing the state of the program. Functions are treated as first-class citizens and can be passed on as arguments or returned as values.

Database/Data driven Programming

This programming methodology is based on data and its movement. Program statements are defined by data rather than hard coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query, and reporting functions. There are several programming languages that are developed mostly for database applications. For example, SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So, it has its own wide application.

Variables

In Java, a variable is a named storage location used to store data during the execution of a program. It represents a memory location that can hold a value of a specific type. Here are some key points about variables in Java:

Variable Declaration: A variable is declared by specifying its type, followed by the variable name. For example:

```
int age; // Declaration of an integer variable
```

Variable Initialization: After declaring a variable, it can be optionally initialized with an initial value using the assignment operator `=`. For example:

```
int age = 25; // Initialization of the age variable with the value 25
```

Variable Types: Java supports several types of variables, including primitive types (e.g., `int`, `double`, `boolean`) and reference types (e.g., objects, arrays, classes). Each variable type determines the range of values it can hold and the operations that can be performed on it.

Variable Names: Variable names are identifiers used to refer to a particular variable. They must follow certain naming conventions in Java. Variable names should start with a letter (or underscore) and can contain letters, digits, and underscores. They are case-sensitive.

Variable Assignment: Variables can be assigned new values using the assignment operator `=`. For example:

```
age = 30; // Assigning a new value to the age variable
```

Variable Scope: The scope of a variable refers to the region of the program where the variable is accessible. Variables can have local scope (limited to a specific block or method) or global scope (accessible throughout the entire class).

Variable Usage: Variables can be used in expressions and statements to perform operations or calculations. They provide a way to store and manipulate data dynamically during program execution.

Here is an example that demonstrates the declaration, initialization, and usage of a variable in Java:

Sample Code

```
int count; // Declaration of an integer variable
count = 10; // Initialization of the count variable with the value 10
int total = count * 5; // Using the count variable in an expression
System.out.println("Total: " + total); // Output: Total: 50
```

In the example above, `count` is a variable of type `int` that is declared, initialized, and used to calculate the `total` value, which is then printed to the console.

Datatypes

In Java, data types specify the kind of values that can be stored in variables and the operations that can be performed on those values. Data types define the size and format of the values that variables can hold. Java has two main categories of data types:

Primitive Data Types: These are the basic data types provided by Java. They are predefined and built into the language. Primitive data types include:

- Numeric Types:

  - `byte`: Represents an 8-bit signed integer.
  - `short`: Represents a 16-bit signed integer.
  - `int`: Represents a 32-bit signed integer.
  - `long`: Represents a 64-bit signed integer.
  - `float`: Represents a single-precision 32-bit floating-point number.
  - `double`: Represents a double-precision 64-bit floating-point number.

- `boolean`: Represents a boolean value (`true` or `false`).

- `char`: Represents a single character (16-bit Unicode value).

Reference Data Types: These data types refer to objects and are created using classes or interfaces. Reference data types include:

- Classes: Custom data types defined by the programmer.

- Arrays: Ordered collections of elements of the same type.

- Interfaces: Abstract types that define a set of methods that implementing classes must implement.

Typecasting is the process of converting one type of data to another type in programming. It allows you to change the representation or format of a value so that it can be used in a different context or operation.

This conversion can be done automatically by the programming language (implicit typecasting) or manually specified by the programmer (explicit typecasting). Typecasting is useful when you need to ensure compatibility between different data types or perform calculations or operations that require values of a specific type.

Java also allows for type conversion or casting between compatible data types. For example, you can convert an `int` to a `double` or a `char` to an `int` using explicit casting.

Here is an example that demonstrates the usage of different data types in Java:

Sample Code

Java

```
int age = 25;
double weight = 65.5;
boolean isStudent = true;
char grade = 'A';
String name = "John Doe";

int[] numbers = {1, 2, 3, 4, 5};
System.out.println("Age: " + age);
System.out.println("Weight: " + weight);
System.out.println("Is Student: " + isStudent);
System.out.println("Grade: " + grade);
System.out.println("Name: " + name);
System.out.println("Numbers: " + Arrays.toString(numbers));
```

In the example above, variables of different data types (`int`, `double`, `boolean`, `char`, `String`, and `int[]`) are declared and initialized with specific values. The values are then printed to the console using `System.out.println() ` statements.

Sample Code

Python

```
age = 25
weight = 65.5
is_student = True
grade = 'A'
name = "John Doe"

numbers = [1, 2, 3, 4, 5]
print("Age:", age)
print("Weight:", weight)
print("Is Student:", is_student)
print("Grade:", grade)
print("Name:", name)
print("Numbers:", numbers)
```

In the Python code, we declare variables age, weight, is_student, grade, and name with their respective values. The variable numbers are declared as a list with elements 1, 2, 3, 4, and 5. We then use the print() function to display the values of these variables.

Sample Code

C#

```
using System;

class Program
{
    static void Main()
    {
        int age = 25;
        double weight = 65.5;
        bool isStudent = true;
        char grade = 'A';
        string name = "John Doe";

        int[] numbers = { 1, 2, 3, 4, 5 };
        Console.WriteLine("Age: " + age);
        Console.WriteLine("Weight: " + weight);
        Console.WriteLine("Is Student: " + isStudent);
        Console.WriteLine("Grade: " + grade);
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Numbers: " + string.Join(", ", numbers));
    }
}
```

In the Python code, we declare variables age, weight, is_student, grade, and name with their respective values. The variable numbers are declared as a list with elements 1, 2, 3, 4, and 5. We then use the print() function to display the values of these variables.

Statements

Statements are individual instructions that make up a program. They represent the basic building blocks for controlling the flow of execution, performing operations, and manipulating data. Here are some commonly used types of statements in Java, python, and C#:

Declaration Statements: These statements declare variables and specify their data types. For example:

```
Java
int age=0; // Variable declaration statement

Python
age = 0  # Variable assignment statement

C#
int age; // Variable declaration statement
age = 0; // Variable assignment statement
```

Expression Statements: These statements evaluate expressions and can perform operations or calculations. For example:

```
Java
int total = count * 5; // Expression statement

Python
total = count * 5  # Expression statement

C#
int total = count * 5; // Expression statement
```

In Java, Python and C#, the expression count * 5 is assigned to the variable total. The result of multiplying count by 5 is stored in the total variable.

Control Flow Statements: These statements control the execution flow of the program based on certain conditions. Common control flow statements include:

- `if` statement: Executes a block of code if a condition is true.
- `else` statement: Executes a block of code if the preceding `if` condition is false.
- `switch` statement: Evaluates different cases and executes the corresponding block of code.
- `for` statement: Repeatedly executes a block of code for a specified number of iterations.
- `while` statement: Repeatedly executes a block of code while a condition is true.
- `do-while` statement: Repeatedly executes a block of code at least once, then continues while a condition is true.
- `break` statement: Terminates the execution of a loop or switch statement.
- `continue` statement: Skips the current iteration of a loop and proceeds to the next iteration.

Expression

In common, an expression is a combination of a series of operators, variables, and method calls that are formed using the syntax given. One example is.

```
int marks;
marks = 85;
```

Here marks=85 is an expression that returns an int value.

```
Double a = 2.8, b = 3.6, result;
result = a + b - 3.6;
```

Here in this example,a+b-3.6 is an expression.

Simple Expressions

A simple expression is a literal method call or variable name without any usage of an operator. For example:

Java

```
int x = 5;                  // Assignment expression
int y = x + 3;              // Arithmetic expression
boolean isTrue = x > 0;     // Comparison expression
double result = Math.sqrt(25);  // Method call expression
```

Python

```
x = 5                    # Assignment expression
y = x + 3                # Arithmetic expression
is_true = x > 0          # Comparison expression
result = math.sqrt(25)   # Function call expression
```

C#

```
int x = 5;                // Assignment expression
int y = x + 3;            // Arithmetic expression
bool isTrue = x > 0;      // Comparison expression
double result = Math.Sqrt(25);  // Method call expression
```

Compound Expressions

It generally involves the usage of operators. It comprises one or more simple expressions, which are then integrated into a larger expression by using the operator.

Java

```
int z = (x + y) * 2;                // Arithmetic expression with parentheses
boolean isValid = x > 0 && y < 10;  // Logical expression with logical operator
String greeting = "Hello, " + name; // String concatenation expression
double result = Math.sqrt(x + y);   // Method call expression with arithmetic
operation
```

Python

```
z = (x + y) * 2                    # Arithmetic expression with parentheses
is_valid = x > 0 and y < 10        # Logical expression with logical operator
greeting = "Hello, " + name        # String concatenation expression
result = math.sqrt(x + y)     # Function call expression with arithmetic
operation
```

C#

```
int z = (x + y) * 2;                // Arithmetic expression with parentheses
bool isValid = x > 0 && y < 10;     // Logical expression with logical operator
string greeting = "Hello, " + name; // String concatenation expression
```

```
double result = Math.Sqrt(x + y);   // Method call expression with arithmetic
operation
```

Compound expressions are used to perform complex computations, combine values, make decisions based on conditions, and perform various operations on data. They involve the use of operators, functions, and language constructs to manipulate and transform values in meaningful ways.

Operators

In programming, operators are symbols or keywords that perform operations on operands (values, variables, or expressions). They allow you to manipulate data, perform calculations, make comparisons, and control the flow of execution in a program. Here are some common types of operators are,

Arithmetic Operators: These operators perform mathematical calculations on numeric operands:

- `+` (Addition): Adds two operands.
- `-` (Subtraction): Subtracts the second operand from the first.
- `*` (Multiplication): Multiplies two operands.
- `/` (Division): Divides the first operand by the second.
- `%` (Modulo): Computes the remainder of division.

Assignment Operators: These operators assign values to variables:

- `=` (Assignment): Assigns the value on the right to the variable on the left.
- `+=`, `-=`, `*=`, `/=`, `%=`: Perform the corresponding arithmetic operation and assignment.

Comparison Operators: These operators compare values and produce a boolean result (`true` or `false`):

- `==` (Equal to): Checks if two operands are equal.
- `!=` (Not equal to): Checks if two operands are not equal.
- `>`, `<` (greater than, less than): Checks if the first operand is greater than/less than the second.
- `>=`, `<=` (greater than or equal to, less than or equal to): Checks if the first operand is greater than or equal to/less than or equal to the second.

Logical Operators: These operators perform logical operations on boolean operands and produce a boolean result:

- `&&` (Logical AND): Returns `true` if both operands are `true`.
- `||` (Logical OR): Returns `true` if at least one of the operands is `true`.
- `!` (Logical NOT): Negates the boolean value of the operand.

Increment and Decrement Operators: These operators increase or decrease the value of a variable:

- `++` (Increment): Increases the value by 1.
- `--` (Decrement): Decreases the value by 1.

Conditional Operator:

- `?:` (Ternary Operator): Evaluates a condition and returns one of two values based on the result.

Here are examples of how the ternary operator can be used in Java, Python, and C#:

Java

```
int x = 10;
int y = 5;

int max = (x > y) ? x : y;
System.out.println("The maximum value is: " + max);
```

Python

```
x = 10
y = 5

max_value = x if x > y else y
print("The maximum value is:", max_value)
```

C#

```
int x = 10;
int y = 5;

int max = (x > y) ? x : y;
Console.WriteLine("The maximum value is: " + max);
```

In the above examples, the ternary operator (condition) ? (Value if true) : (value if false) is used to assign the maximum value of x and y to the variable max. The condition (x > y) is evaluated, and if it is true, the value of x is assigned to max; otherwise, the value of y is assigned to max. Finally, the maximum value is printed on the console.

Bitwise Operators: These operators perform operations on individual bits of integer operands:

- `&` (Bitwise AND), `|` (Bitwise OR), `^` (Bitwise XOR): Perform bitwise operations.
- `<<` (Left shift), `>>` (Right shift): Shift the bits of the operand left or right.

Here are examples of how bitwise operators can be used in Java, Python, and C#:

Java

```
int x = 5;  // Binary: 0101
int y = 3;  // Binary: 0011

// Bitwise AND
int resultAnd = x & y;  // Binary: 0001 (Decimal: 1)
System.out.println("Bitwise AND: " + resultAnd);

// Bitwise OR
int resultOr = x | y;   // Binary: 0111 (Decimal: 7)
System.out.println("Bitwise OR: " + resultOr);
```

```
// Bitwise XOR
int resultXor = x ^ y;  // Binary: 0110 (Decimal: 6)
System.out.println("Bitwise XOR: " + resultXor);

// Bitwise complement
int resultComplementX = ~x;  // Binary: 1010 (Decimal: -6)
System.out.println("Bitwise Complement of x: " + resultComplementX);
```

Python

```
x = 5    # Binary: 0101
y = 3    # Binary: 0011

# Bitwise AND
result_and = x & y   # Binary: 0001 (Decimal: 1)
print("Bitwise AND:", result_and)

# Bitwise OR
result_or = x | y    # Binary: 0111 (Decimal: 7)
print("Bitwise OR:", result_or)

# Bitwise XOR
result_xor = x ^ y   # Binary: 0110 (Decimal: 6)
print("Bitwise XOR:", result_xor)

# Bitwise complement
result_complement_x = ~x   # Binary: 1010 (Decimal: -6)
print("Bitwise Complement of x:", result_complement_x)
```

C#

```
int x = 5;   // Binary: 0101
int y = 3;   // Binary: 0011

// Bitwise AND
int resultAnd = x & y;   // Binary: 0001 (Decimal: 1)
```

```
Console.WriteLine("Bitwise AND: " + resultAnd);

// Bitwise OR
int resultOr = x | y;    // Binary: 0111 (Decimal: 7)
Console.WriteLine("Bitwise OR: " + resultOr);

// Bitwise XOR
int resultXor = x ^ y;   // Binary: 0110 (Decimal: 6)
Console.WriteLine("Bitwise XOR: " + resultXor);

// Bitwise complement
int resultComplementX = ~x;   // Binary: 1010 (Decimal: -6)
Console.WriteLine("Bitwise Complement of x: " + resultComplementX);
```

In the above examples, the following bitwise operators are demonstrated:

Bitwise AND (&): Performs a binary AND operation on each corresponding pair of bits. The result is 1 if both bits are 1; otherwise, it is 0.

Bitwise OR (|): Performs a binary OR operation on each corresponding pair of bits. The result is 1 if at least one of the bits is 1; otherwise, it is 0.

Bitwise XOR (^): Performs a binary XOR (exclusive OR) operation on each corresponding pair of bits. The result is 1 if the bits are different; otherwise, it is 0.

Bitwise complement (~): Inverts each bit of the operand, changing 1s to 0s and 0s to 1s.

The examples illustrate these bitwise operations using two integer variables, x, and y, and then print the results.

These are some of the commonly used operators. They allow you to perform various operations and control the behavior of your program. The choice of operator depends on the type of operation you want to perform, and the data types involved.

Strings

In Java, strings are a sequence of characters, and they are represented as objects of the `String` class. Strings in Java are immutable, which means their values cannot be changed after they are created. Any operation that appears to modify a string creates a new string object with the desired value.

Here are some key points to understand about strings in Java in a simple manner:

1. **String Creation:** Strings can be created in Java using string literals, which are enclosed in double quotes, or by creating objects of the `String` class using the `new` keyword.

Java

```
String str1 = "Hello, world!";  // Using string literal
String str2 = new String("Hello, world!"); // Using String class constructor
```

Python

```
str1 = 'Hello, world!'  # Using single quotes
str2 = "Hello, world!"  # Using double quotes
str3 = '''Hello,
world!'''                 # Using triple quotes for multi-line string
```

C#

```
string str1 = "Hello, world!";  // Using double quotes
string str2 = @"Hello, world!"; // Using verbatim string literal
```

In all three languages, strings are immutable, meaning their values cannot be changed after they are created. You can perform various operations on strings, such as concatenation, substring extraction, searching, and more, to manipulate and work with string data**.**

2. **String Concatenation:** Strings can be concatenated using the `+` operator or the `concat()` method. For example:

Java

```
String greeting = "Hello" + " " + "World"; // Using + operator
String message = "Welcome".concat(" to").concat(" Java");
// Using concat() method
```

Python

```
greeting = "Hello" + " " + "World"  # Using + operator
message = "Welcome" + " to" + " Python"  # Using + operator
```

C#

```
string greeting = "Hello" + " " + "World";  // Using + operator
string message = "Welcome" + " to" + " C#";  // Using + operator
```

In Python and C#, the + operator is used for string concatenation. The + operator concatenates multiple strings together, creating a new string that is the result of joining the individual strings.

In Python, the + operator can be used directly to concatenate strings. In C#, the + operator is also used for string concatenation.

So, the Python code uses the + operator to concatenate the strings "Hello", " ", and "World", resulting in the string "Hello World". Similarly, the C# code uses the + operator to concatenate the strings "Hello", " ", and "World", resulting in the string "Hello World".

3. **String Length:** The length of a string, i.e., the number of characters in the string, can be obtained using the `length()` method. For example:

Java

```
String message = "Hello, world!";
int length = message.length();
System.out.println(length);  // Output: 13
```

Python

```
message = "Hello, world!"
length = len(message)
print(length)  # Output: 13
```

In Python, you can use the built-in len() function to get the length of a string. The len() function returns the number of characters in the string, including spaces and punctuation marks.

C#

```
string message = "Hello, world!";
int length = message.Length;
Console.WriteLine(length);  // Output: 13
```

In C#, you can use the Length property of a string to retrieve its length. The Length property returns the number of characters in the string, like Python's len() function.

4. **String Comparison:** Strings can be compared for equality using the `equals()` method, which compares the contents of the strings, or using the `==` operator, which compares the memory addresses of the string objects. For example:

Java

```
String str1 = "Hello";
String str2 = "Hello";
String str3 = new String("Hello");

System.out.println(str1.equals(str2)); // true
System.out.println(str1 == str2); // true
System.out.println(str1.equals(str3)); // true
System.out.println(str1 == str3); // false
```

Python

```
str1 = "Hello"
str2 = "Hello"
str3 = str("Hello")
print(str1 == str2)  # True
print(str1 is str2)  # True.
print(str1 == str3)  # True
print(str1 is str3)  # False
```

In Python, strings can be compared using the == operator to check if they have the same content. The `is` operator checks if two strings refer to the same object in memory.

C#

```
using System;

class Program
{
```

```
    static void Main()
    {
        string str1 = "Hello";
        string str2 = "Hello";
        string str3 = new string("Hello".ToCharArray());

        Console.WriteLine(str1.Equals(str2));  // True
        Console.WriteLine(str1 == str2);       // True
        Console.WriteLine(str1.Equals(str3));  // True
        Console.WriteLine(str1 == str3);       // False
    }
}
```

In C#, strings can be compared using the Equals() method to check for content equality. The == operator can also be used to compare strings, and it checks if the references are equal.

5. **String Manipulation:** Strings in Java provide many built-in methods for manipulating strings, such as charAt(), substring(), toUpperCase(), `toLowerCase()`, `trim()`, and many more. For example:

Java

```
String str = "Hello World";
char ch = str.charAt(0); // 'H'
String sub = str.substring(6); // "World"
String upper = str.toUpperCase(); // "HELLO WORLD"
String lower = str.toLowerCase(); // "hello world"
String trimmed = str.trim();
// "Hello World" (without leading/trailing spaces)
```

Python

```
str = "Hello World"
ch = str[0]  # 'H'
sub = str[6:]  # "World"
```

```
upper = str.upper()  # "HELLO WORLD"
lower = str.lower()  # "hello world"
trimmed = str.strip()  # "Hello World" (without leading/trailing spaces)
```

In Python, string indexing is done using square brackets "[]" and slicing is done using the colon (:) notation. The upper() and lower() methods are used to convert the string to uppercase and lowercase, respectively. The strip() method is used to remove leading and trailing spaces.

C#

```
string str = "Hello World";
char ch = str[0];  // 'H'
string sub = str.Substring(6);  // "World"
string upper = str.ToUpper();  // "HELLO WORLD"
string lower = str.ToLower();  // "hello world"
string trimmed = str.Trim();  // "Hello World" (without leading/trailing
spaces)
```

In C#, string indexing is also done using square brackets "[]" and the Substring() method is used for extracting substrings. The ToUpper() and ToLower() methods are used to convert the string to uppercase and lowercase, respectively. The Trim() method is used to remove leading and trailing spaces.

6. **String Formatting:** Strings in Java can be formatted using the printf() method, which allows for creating formatted output with placeholders for values. For example:

Java

```
String name = "Alice";
int age = 30;
double height = 5.6;
System.out.printf("Name: %s, Age: %d, Height: %.2f%n", name, age, height);
```

```
// Output: Name: Alice, Age: 30, Height: 5.60
```

Python

```
name = "Alice"
age = 30
height = 5.6
print("Name: %s, Age: %d, Height: %.2f" % (name, age, height))

# Output: Name: Alice, Age: 30, Height: 5.60
```

In Python, the print() function is used to display the formatted string. The string interpolation is achieved using the % operator with the format specifiers %s for strings, %d for integers, and %.2f for floating-point numbers with two decimal places.

C#

```
string name = "Alice";
int age = 30;
double height = 5.6;
Console.WriteLine("Name: {0}, Age: {1}, Height: {2:F2}", name, age, height);

// Output: Name: Alice, Age: 30, Height: 5.60
```

In C#, the Console.WriteLine() method is used to display the formatted string. The string interpolation is achieved using curly braces {} to enclose the placeholders, and the format specifiers :F2 is used to format the floating-point number with two decimal places.

Arrays

Arrays in Java, Python, and C# are data structures used to store multiple values of the same type. They provide a way to efficiently manage collections of elements, such as numbers, strings, or objects.

In Java:

In Java, an array is a fixed-size container that can hold elements of the same type. The size of an array is determined when it is created and cannot be changed. Array elements are accessed using an index, starting from 0. Java arrays can store primitive data types (like int, char, etc.) as well as objects. The syntax to declare an array in Java is:

```
dataType[] arrayName = new dataType[arraySize];

// Declaration and initialization of an integer array with size 5
int[] numbers = new int[5];

// Declaration and initialization of a string array with size 3
String[] names = new String[3];

int[] numbers = {1, 2, 3, 4, 5};

// Accessing the element at index 2
int element = numbers[2];
System.out.println(element);  // Output: 3
```

In Python:

In Python, an array is represented by the list data structure, which is a dynamic and flexible container. Python lists can store elements of different types and can grow or shrink in size dynamically. Elements in a list are also accessed using an index, starting from 0. The syntax to declare a list in Python is:

```
listName = [element1, element2, ...]
```

```
# Declaration and initialization of an integer array
numbers = [1, 2, 3, 4, 5]

# Declaration and initialization of a string array
names = ["Alice", "Bob", "Charlie"]

numbers = [1, 2, 3, 4, 5]

# Accessing the element at index 2
element = numbers[2]
print(element)  # Output: 3
```

In C#:

In C#, arrays are similar to Java, where they are fixed-size collections of elements of the same type. The size of an array is determined when it is created and cannot be changed. C# arrays can store primitive data types (like int, char, etc.) as well as objects. Elements in an array are accessed using an index, starting from 0. The syntax to declare an array in C# is:

```
dataType[] arrayName = new dataType[arraySize];

// Declaration and initialization of an integer array with size 5
int[] numbers = new int[5];

// Declaration and initialization of a string array with size 3
string[] names = new string[3];

int[] numbers = {1, 2, 3, 4, 5};

// Accessing the element at index 2
int element = numbers[2];
Console.WriteLine(element);  // Output: 3
```

In Java, Python, and C#, arrays can be classified as single-dimensional or multidimensional based on the number of indices required to access their elements:

Single-Dimensional Array:

A single-dimensional array, also known as a one-dimensional array, is an array that uses a single index to access its elements. It represents a linear collection of elements. It is the most basic form of an array.

```
Java
int[] numbers = {1, 2, 3, 4, 5};

Python
numbers = [1, 2, 3, 4, 5]

C#
int[] numbers = {1, 2, 3, 4, 5};
```

Multidimensional Array:

A multidimensional array is an array that uses multiple indices to access its elements. It represents a collection of elements arranged in multiple dimensions, such as rows and columns in a table-like structure.

```
Java
int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

Python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

C#
int[,] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

In the examples above, a two-dimensional array is used as a demonstration of a multidimensional array. However, multidimensional arrays can have more than two dimensions, such as three-dimensional, four-dimensional, and so on, depending on the requirements of your program.

To access elements in a multidimensional array, you need to provide the indices for each dimension. For example, in a two-dimensional array, you would use two indices to access an element: one for the row and one for the column.

Control Statements

Control statements in Java are statements that control the order in which the program executes. They allow the programmer to specify conditions under which certain parts of the code should be executed or skipped, to loop over a block of code, or to jump to a different part of the program. The control statements in Java are:

if-else statements: These statements allow the program to execute certain code if a particular condition is true and execute different code if the condition is false.

Java

```
int number = 10;
if (number > 0) {
    System.out.println("The number is positive.");
} else {
    System.out.println("The number is negative or zero.");
}
```

Python

```
number = 10
if number > 0:
    print("The number is positive.")
else:
    print("The number is negative or zero.")
```

C#

```
int number = 10;
if (number > 0)
{
    Console.WriteLine("The number is positive.");
```

```
}
else
{
    Console.WriteLine("The number is negative or zero.");
}
```

The code snippet checks whether the variable number is greater than 0 and prints a corresponding message based on the result.

if-else-if: These statement in Java is a type of conditional statement that allows you to specify multiple conditions to be checked sequentially and execute different code blocks based on the first condition that evaluates to true. It provides a way to evaluate multiple conditions in a chain, where each condition is evaluated only if the previous conditions are false.

Java

```
int number = 0;

if (number > 0) {
    System.out.println("The number is positive.");
} else if (number < 0) {
    System.out.println("The number is negative.");
} else {
    System.out.println("The number is zero.");
}
```

Python

```
number = 0
if number > 0:
    print("The number is positive.")
elif number < 0:
```

```
    print("The number is negative.")
else:
    print("The number is zero.")
```

C#

```
int number = 0;
if (number > 0)
{
    Console.WriteLine("The number is positive.");
}
else if (number < 0)
{
    Console.WriteLine("The number is negative.");
}
else
{
    Console.WriteLine("The number is zero.");
}
```

The code snippet checks the value of the variable number and prints the corresponding message based on whether it is positive, negative, or zero.

A **nested-if** statement in Java is a type of conditional statement that allows you to use one or more if statements within the body of another if statement. It provides a way to have multiple levels of conditional branching, where the inner if statements are executed based on the outcome of the outer if statements.

Java

```
int age = 25;
boolean hasLicense = true;
if (age >= 18) {
```

```
    if (hasLicense) {
        System.out.println("You are eligible to drive.");
    } else {
        System.out.println("You are eligible to apply for a license.");
    }
} else {
    System.out.println("You are not eligible to drive.");
}
```

Python

```
age = 25
hasLicense = True

if age >= 18:
    if hasLicense:
        print("You are eligible to drive.")
    else:
        print("You are eligible to apply for a license.")
else:
    print("You are not eligible to drive.")
```

C#

```
int age = 25;
bool hasLicense = true;

if (age >= 18)
{
    if (hasLicense)
    {
        Console.WriteLine("You are eligible to drive.");
```

```
    }
    else
    {
        Console.WriteLine("You are eligible to apply for a license.");
    }
}
else
{
    Console.WriteLine("You are not eligible to drive.");
}
```

The code snippet checks the age and license status of an individual and prints an appropriate message based on their eligibility to drive. If the age is 18 or above, it checks whether they have a license or not and prints a corresponding message. If the age is below 18, it prints a message stating they are not eligible to drive.

switch statements: These statements allow the program to execute different codes based on the value of an expression.

Java

```
int dayOfWeek = 1; // 1 for Sunday, 2 for Monday, and so on
switch (dayOfWeek) {
    case 1:
        System.out.println("Today is Sunday");
        break;
    case 2:
        System.out.println("Today is Monday");
        break;
    case 3:
        System.out.println("Today is Tuesday");
        break;
    case 4:
        System.out.println("Today is Wednesday");
        break;
    case 5:
        System.out.println("Today is Thursday");
```

```
        break;
    case 6:
        System.out.println("Today is Friday");
        break;
    case 7:
        System.out.println("Today is Saturday");
        break;
    default:
        System.out.println("Invalid day of week");
        break;
}
```

Python

```
dayOfWeek = 1  # 1 for Sunday, 2 for Monday, and so on

if dayOfWeek == 1:
    print("Today is Sunday")
elif dayOfWeek == 2:
    print("Today is Monday")
elif dayOfWeek == 3:
    print("Today is Tuesday")
elif dayOfWeek == 4:
    print("Today is Wednesday")
elif dayOfWeek == 5:
    print("Today is Thursday")
elif dayOfWeek == 6:
    print("Today is Friday")
elif dayOfWeek == 7:
    print("Today is Saturday")
else:
    print("Invalid day of week")
```

C#

```
int dayOfWeek = 1; // 1 for Sunday, 2 for Monday, and so on

switch (dayOfWeek)
{
    case 1:
        Console.WriteLine("Today is Sunday");
        break;
    case 2:
        Console.WriteLine("Today is Monday");
        break;
    case 3:
        Console.WriteLine("Today is Tuesday");
        break;
    case 4:
        Console.WriteLine("Today is Wednesday");
        break;
    case 5:
        Console.WriteLine("Today is Thursday");
        break;
    case 6:
        Console.WriteLine("Today is Friday");
        break;
    case 7:
        Console.WriteLine("Today is Saturday");
        break;
    default:
        Console.WriteLine("Invalid day of week");
        break;
}
```

The code snippet determines the day of the week based on the value of the dayOfWeek variable and prints the corresponding day name. If the value is not within the range of 1 to 7, it prints an "Invalid day of week" message.

Loops

While loops: These loops execute a block of code repeatedly if a certain condition is true.

Java

```
int count = 1;
while (count <= 10) {
    System.out.println("Count: " + count);
    count++;
}
```

Python

```
count = 1

while count <= 10:
    print("Count:", count)
    count += 1
```

C#

```
int count = 1;

while (count <= 10)
{
    Console.WriteLine("Count: " + count);
    count++;
}
```

The code snippet initializes a variable count to 1 and then enters a while loop. The loop iterates if the count is less than or equal to 10. Within each iteration, it prints the current value of count and then increments it by 1. The loop continues until the count reaches 11.

do-while loops: These loops are like while loops, but they execute the code block at least once before checking the condition.

Java

```
int count = 1;
do {
    System.out.println("Count: " + count);
    count++;
} while (count <= 10);
```

Python

Python does not have a built-in "do-while" loop like some other programming languages because it emphasizes simplicity and readability. The "do-while" loop can sometimes make the code harder to understand and may lead to potential bugs if not used carefully. Python encourages the use of "while" loops with a conditional check at the beginning, which can achieve the same functionality as a "do-while" loop while keeping the code clean and easy to follow.

Here's a simple explanation of why Python does not have a "do-while" loop, along with an example:

In other languages, a "do-while" loop looks like this:

```
#include <iostream>

int main() {
    int count = 0;

    do {
        std::cout << "Count: " << count << std::endl;
        count++;
```

```
    } while (count < 5);

    return 0;
}
```

In this C++ code, the block of code inside the loop is executed at least once before checking the condition count < 5. If the condition is true, it will continue to execute the loop.

Now, in Python, you can achieve the same functionality using a "while" loop:

```
count = 0

while count < 5:
    print("Count:", count)
    count += 1
```

This Python code achieves the same functionality as the C++ "do-while" loop. The block of code inside the "while" loop is executed as long as the condition count < 5 is true. The code will run at least once, just like a "do-while" loop.

Python's emphasis on readability and avoiding unnecessary complexity makes it a popular choice among developers.

C#

```
int count = 1;
do
{
    Console.WriteLine("Count: " + count);
    count++;
} while (count <= 10);
```

The code snippet initializes a variable count to 1 and then enters a do-while loop. The loop executes the code block at least once and continues if the count is less than or equal to 10. Within each iteration, it prints the current value of count and then increments it by 1. the loop continues until the count reaches 11.

for loop: These loops execute a block of code a specific number of times, based on a counter variable.

Java

```
for (int i = 1; i <= 10; i++) {
    System.out.println("Count: " + i);
}
```

Python

```
for i in range(1, 11):
    print("Count:", i)
```

C#

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("Count: " + i);
}
```

The code snippet uses a for loop to iterate from 1 to 10. The loop initializes a variable i to 1, continues as long as i is less than or equal to 10, and increments i by 1 in each iteration.

Within each iteration, it prints the current value of i. The loop iterates a total of 10 times, printing the values from 1 to 10.

Break statements: These statements terminate the current loop or switch statement and resume execution at the next statement.

Java

```
for (int i = 1; i <= 10; i++) {
    System.out.println("Count: " + i);
    if (i == 5) {
        break;
    }
}
```

Python

```
for i in range(1, 11):
    print("Count:", i)
    if i == 5:
        break
```

C#

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("Count: " + i);
    if (i == 5)
    {
        break;
    }
}
```

The code snippet uses a for loop to iterate from 1 to 10. The loop initializes a variable i to 1, continues as long as i is less than or equal to 10, and increments i by 1 in each iteration. Within each iteration, it prints the current value of i. Additionally, it checks if i is equal to 5 and if so, it breaks out of the loop, prematurely terminating the iteration.

Continue statements: These statements skip the rest of the current iteration of a loop and continue with the next iteration.

Java

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        continue;
    }
    System.out.println("Count: " + i);
}
```

Python

```
for i (in range= 1,11)
    if i == 5:
        continue
    print("Count: ", i)
}
```

C#

```
for (int i = 1; i <= 10; i++)
{
    if (i == 5)
    {
        continue;
    }
```

```
    Console.WriteLine("Count: " + i);
}
```

The code snippet uses a for loop to iterate from 1 to 10. The loop initializes a variable i to 1, continues as long as i is less than or equal to 10, and increments i by 1 in each iteration. Within each iteration, it checks if i is equal to 5. If it is, the continue statement is executed, skipping the rest of the code block for that iteration, and moving on to the next iteration. If i is not equal to 5, it proceeds to print the current value of i.

Functions and its Basics

In common, functions are called methods. Here are the basic concepts related to methods in Java, Python and C#:

Method Declaration: A method is declared within a class using the following syntax:

Java

```
returnType methodName(parameterType parameter1, parameterType parameter2, ...) {
    // Method body
    // Code to be executed
    // Return statement (if applicable)
}
```

Python

```
def functionName(parameter1, parameter2, ...):
    # Function body
    # Code to be executed
    # Return statement (if applicable)
```

C#

```
returnType MethodName(parameterType parameter1, parameterType parameter2, ...)
{
    // Method body
    // Code to be executed
    // Return statement (if applicable)
}
```

Return Type: The return type specifies the type of value that the method will return after execution. If the method doesn't return any value, the return type is `void`.

Method Name: It is the identifier used to call the method. It should be unique within the class.

Parameters: Parameters are optional and allow you to pass values to the method for processing. They are specified inside parentheses and separated by commas. Each parameter has a type and a name.

Method Body: It contains a set of statements that define the behavior of the method. The statements are enclosed within curly braces.

Return Statement: If the method has a return type other than `void`, it must use the `return` statement to return a value of that type. The return statement also terminates the method execution.

Method Call: To execute a method, you need to call it by its name and provide any required arguments in parentheses.

Sample Code

Java

```
public class ExampleJava {
    public static void main(String[] args) {
        int result = multiply(3, 4);
        System.out.println("Result: " + result);
    }

    public static int multiply(int a, int b) {
        return a * b;
    }
}
```

Python

```
def multiply(a, b):
    return a * b

result = multiply(3, 4)
print("Result:", result)
```

C#

```
using System;

public class ExampleCSharp
{
    public static void Main(string[] args)
    {
        int result = Multiply(3, 4);
        Console.WriteLine("Result: " + result);
    }

    public static int Multiply(int a, int b)
    {
        return a * b;
    }
}
```

In all three examples, we define a function called multiply that takes two arguments a & b. The function multiplies these two values together and returns the result. We then call the multiply function with arguments 3 & 4 and store the returned value in the result variable. Finally, we print the result to the console.

Static Keyword

In Java, Python, and C#, the `static` keyword is used to define class-level members and variables that belong to the class itself, rather than to an instance of the class. The behavior and usage of the `static` keyword are similar in all three languages, but there might be some subtle differences in the way it is applied.

Here's a brief explanation of `static` in each language:

Java:

In Java, the `static` keyword can be applied to variables, methods, and nested classes. When applied to a variable, it means that the variable is shared among all instances of the class. When applied to a method, it means that the method belongs to the class itself and can be called without creating an instance of the class. Static nested classes are associated with the outer class and do not require an instance of the outer class to be instantiated.

Python:

In Python, the `static` keyword is not explicitly used to define static members. Instead, you can achieve similar behavior by defining variables and methods within the class scope. These members are then accessible using the class name itself, without the need for an instance. Although Python doesn't have a dedicated `static` keyword, the concept of static members is still present.

C#:

In C#, the `static` keyword is used to define members that are associated with the class itself rather than with instances of the class. It can be applied to variables, methods, properties, events, constructors, and classes. Static members are accessed using the class name directly, without the need for an instance. They are shared among all instances of the class and are typically used for utility methods, global counters, or constants.

Overall, the `static` keyword is used to define class-level members that are shared among instances or belong to the class itself rather than individual objects.

Here are examples of using the static keyword in Java, Python, and C#:

Java

```
public class Example {
    public static int count; // static variable

    public static void incrementCount() { // static method
        count++;
    }
}

public class Main {
    public static void main(String[] args) {
        Example.incrementCount(); // Accessing static method without
creating an instance
        System.out.println(Example.count);    //    Accessing    static
variable without creating an instance
    }
}
```

Python

```
class Example:
    count = 0  # static variable

    @staticmethod
    def increment_count():  # static method
        Example.count += 1
```

```
Example.increment_count()   # Accessing static method without creating an instance
print(Example.count)    # Accessing static variable without creating an instance.
```

C#

```
public class Example {
    public static int Count; // static variable

    public static void IncrementCount() { // static method
        Count++;
    }
}

public class Main {
    static void Main() {
        Example.IncrementCount(); // Accessing static method without creating an instance
        Console.WriteLine(Example.Count);   //   Accessing   static   variable without creating an instance
    }
}
```

In all three examples, the static keyword is used to define a static variable (count in Java and Python, Count in C#) and a static method (incrementCount() in Java and Python, IncrementCount() in C#). These static members can be accessed without creating an instance of the class.

Final Keyword

The final keyword in Java, Python, and C# is used to define entities (variables, methods, classes) that cannot be modified or overridden once they are initialized or declared. The behavior and usage of the final keyword are similar in all three languages.

Here is an example of using the final keyword in each language:

Java

```
public class Example {
    public final int constantValue = 10; // final variable

    public final void finalMethod() { // final method
        // Method implementation
    }
}

public class SubExample extends Example {
    // Cannot override finalMethod()
    // Cannot modify constantValue
}
```

Python

```
class Example:
    constant_value = 10  # final variable

    def final_method(self):  # final method
        # Method implementation
        pass

class SubExample(Example):
    # Cannot override final_method()
```

```
    # Cannot modify constant_value
    pass
```

C#

```
public class Example {
    public const int ConstantValue = 10; // final variable (constant)

    public sealed void FinalMethod() { // final method (sealed)
        // Method implementation
    }
}

public class SubExample : Example {
    // Cannot override FinalMethod()
    // Cannot modify ConstantValue
}
```

In the examples above:

Java: The final keyword is used to define a final variable (constantValue) and a final method (finalMethod()) in the Example class. In the SubExample class, it is not possible to override the final method or modify the final variable.

Python: In Python, there is no explicit final keyword. Instead, it is a convention to use uppercase names for constants (e.g., constant_value). By convention, it indicates that the variable should not be modified. Similarly, the final method final_method() in the Example class cannot be overridden in the SubExample class.

C#: The final keyword is not used directly in C#. Instead, the const keyword is used to define compile-time constants (e.g., ConstantValue). Additionally, the sealed keyword is used to prevent a method from being overridden (e.g., FinalMethod()). The behavior is

Similar to final in other languages, where the constant variable and sealed method cannot be modified or overridden in the derived class.

Note that the specific use of final might vary depending on the language and its conventions, but the overall purpose remains the same—to mark entities as unmodifiable or uninheritable.

Collections

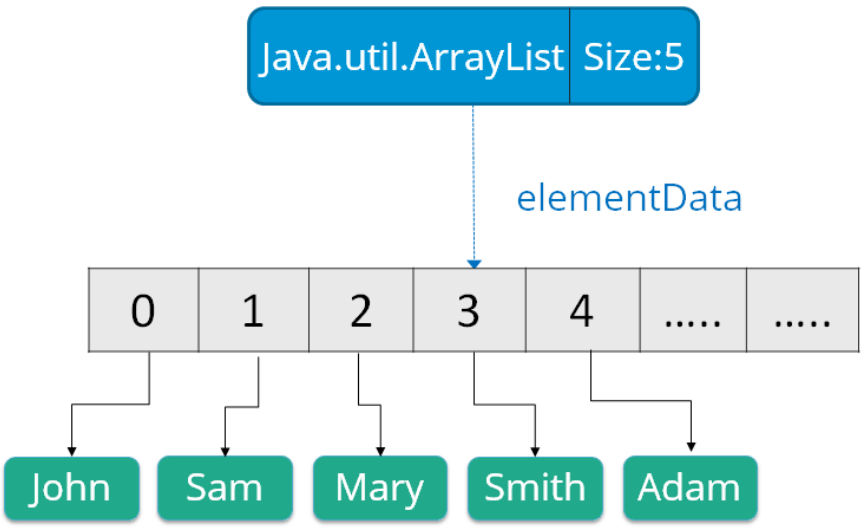
List

The List interface represents an ordered collection of elements, where each element has an index that can be used to access it which may contain duplicates. It is an interface that extends the Collection interface. Lists are further classified into the following.

- **ArrayList**
- **LinkedList**
- **Vectors**

Array List

ArrayList is the implementation of List Interface where the elements can be dynamically added or removed from the list. Also, the size of the list is increased dynamically if the elements are added more than the initial size.



Syntax JAVA

```
ArrayList<DataType> listName = new ArrayList<DataType>();
```

In this syntax, **DataType** specifies the type of element that the **ArrayList** will hold, and **listName** is the name of the **ArrayList** that you are creating.

C#

```
List<DataType> listName = new List<DataType>();
```

However, it is important to note that the ArrayList class is not commonly used in modern C# code. Instead, it is recommended to use the generic List<T> class from the System.Collections.Generic namespace, which provides better type safety and performance. Here is how you can use List<T> in C#.

Python

```
list_name = []
```

Python lists are dynamically typed and can contain elements of several types. Python lists do not require specifying the data type of the elements in advance. You can simply create an empty list using square brackets "[]".

Sample Code

Here is an example of how to create an **ArrayList** of strings, add some elements to it, and iterate through the elements:

Java

```
ArrayList<String> list = new ArrayList<String>();
list.add("apple");
list.add("banana");
list.add("orange");
for (String fruit : list)
{
System.out.println(fruit);
}
```

Overall, ArrayList is an especially useful data structure in Java that provides dynamic size, random access, and flexible insertion and removal of elements.

C#

```
List<string> list = new List<string>();
list.Add("apple");
list.Add("banana");
list.Add("orange");
foreach (string fruit in list)
{
    Console.WriteLine(fruit);
}
```

In C#, we use **List<T>** from the **System.Collections.Generic** namespace, where **"T"** represents the type of elements in the list. The **Add** method is used to add elements to the list, and the foreach loop is used to iterate over the list and print each fruit to the console using **Console.WriteLine().**

Python

```
list = []
list.append("apple")
list.append("banana")
list.append("orange")
for fruit in list:
    print(fruit)
```

In Python, you can create an empty list using square brackets []. The append() method is used to add elements to the list. In the loop, we iterate over the list and print each fruit using the print() function.

Linked List

LinkedList is implemented using a linked list data structure, where each element is stored in a node that contains a reference to the next and previous nodes in the list.

Syntax

JAVA

```
LinkedList<DataType> listName = new LinkedList<DataType>();
```

In this syntax, **DataType** specifies the type of element that the LinkedList will hold, and **listName** is the name of the LinkedList that you are creating.

C#

```
LinkedList<DataType> listName = new LinkedList<DataType>();
```

In C#, you can use the LinkedList<T> class from the **System.Collections.Generic** namespace to create a linked list.

Python

```
list_name = []  # Creating an empty list
```

In Python, you can use a list to achieve similar functionality as a Java LinkedList.
However, Python lists are more flexible and don't require specifying the data type of elements upfront.

Sample Code

Java

Here's an example Java program that demonstrates how to create a LinkedList, add elements to it, remove elements from it, and iterate over its elements using a for-each loop:

```
import java.util.LinkedList;
public class LinkedListExample
{
```

```
    public static void main(String[] args)
      {
                        // create a new LinkedList of strings
                LinkedList<String> list = new LinkedList<String>();
                        // add some elements to the LinkedList
                  list.add("apple");
                  list.add("banana");
                  list.add("orange");
                  list.add("pear");
                // remove an element from the LinkedList
                 list.remove("banana");
                // iterate over the elements of the LinkedList
                  for (String fruit : list)
            {
                   System.out.println(fruit);
                   }
        }
}
```

In this program, we create a new LinkedList of strings, add some elements to it, remove an element from it using the remove() method, and then iterate over the remaining elements using a for-each loop.

C#

```
using System;
using System.Collections.Generic;

public class LinkedListExample
{
    public static void Main(string[] args)
    {
        // create a new LinkedList of strings
        LinkedList<string> list = new LinkedList<string>();

        // add some elements to the LinkedList
        list.AddLast("apple");
        list.AddLast("banana");
```

```
        list.AddLast("orange");
        list.AddLast("pear");
        // remove an element from the LinkedList
        list.Remove("banana");

        // iterate over the elements of the LinkedList
        foreach (string fruit in list)
        {
            Console.WriteLine(fruit);
        }
    }
}
```

In C#, we use the LinkedList<T> class from the System.Collections.Generic namespace. The methods AddLast and Remove are used to add elements to the linked list and remove an element, respectively. The foreach loop is used to iterate over the elements of the linked list and print them using Console.WriteLine().

Python

```
class LinkedListExample:
    @staticmethod
    def main():
        # Create a new LinkedList of strings
        list = []

        # Add some elements to the LinkedList
        list.append("apple")
        list.append("banana")
        list.append("orange")
        list.append("pear")

        # Remove an element from the LinkedList
        list.remove("banana")

        # Iterate over the elements of the LinkedList
        for fruit in list:
            print(fruit)
```

```
# Execute the main method
LinkedListExample.main()
```

In Python, we can use a simple list to represent a collection of items. The append method is used to add elements to the list, and the remove method is used to remove a specific element. We can iterate over the elements of the list using a for loop and print each element using the print function.

Linked List class uses two types of Linked list to store the elements.

- Singly Linked List
- Doubly Linked List

Vectors

Vectors are like arrays, where the elements of the vector object can be accessed via an index into the vector. Vector implements a dynamic array. Also, the vector is not limited to a specific size, it can shrink or grow automatically whenever required, it is like ArrayList.

Stack

The Stack data structure is modelled and implemented by the Stack class. The class is organized around the last-in-first-out philosophy. The class also includes three additional functions: empty, search, and peek, in addition to the fundamental push and pop operations. The subclass of Vector can also be used to refer to this class.

Queue

Queue in Java follows a FIFO approach i.e., it orders the elements in (**First in First Out**) manner. In a queue, the first element is removed first, and the last element is removed in the end.

Each basic method exists in two forms: one throws an exception if the operation fails, the other returns a special value.

	Throws Exception	Returns Special Value
Insert	Add(e)	Offer(e)
Remove	Remove()	Poll()
Examine	Element()	Peek()

Classes that implement Queue Interface in Java:

To use the functionalities of Queue, we need to use classes that implement it:

- Priority Queue
- Dequeue

Sets

A Set refers to a collection that cannot contain duplicate elements. It is used to model mathematical set abstraction. Set has its implementation in various classes such as HashSet, TreeSet and LinkedHashSet.

HashSet

HashSet class creates a collection that uses a hash table for storage. Hashset only contains unique elements, and it inherits the AbstractSet class and implements Set interface. Also, it uses a mechanism hashing to store the elements.

Linked Hash Set

LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It contains only unique elements like HashSet. Linked HashSet also provides all optional set operations and maintains insertion order.

Tree Set

TreeSet class implements the Set interface that uses a tree for storage. The objects of this class are stored in ascending order. Also, it inherits AbstractSet class and implements

NavigableSet interface. It contains only unique elements like HashSet. In TreeSet class, access and retrieval time are faster.

Fundamentals and Concepts

Introduction to Object Oriented Programming

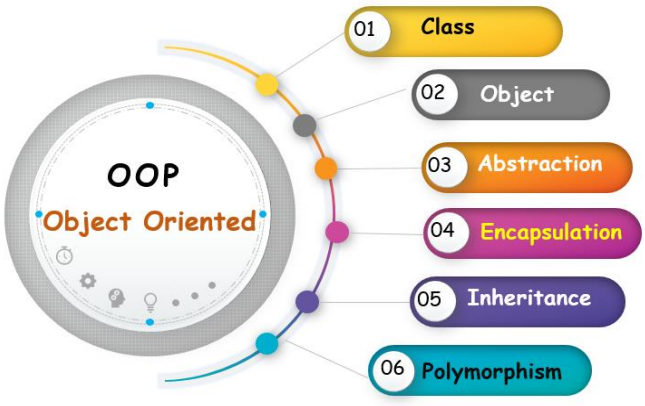
Object Oriented Programming (OOP) is a programming paradigm based on objects, which can contain data and code. OOP is widely used in modern software development because it provides an organized and modular approach to programming.

The main benefits of OOP include reusability, maintainability, and scalability. With OOP, developers can create **classes** which are blueprints for **objects**, and then use those classes to create instances of objects. This allows for code reuse and makes it easier to maintain and scale applications.

Key Concepts of Object-Oriented Programming

There are several key concepts that are central to OOP.

- Classes
- Object
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



Class

Classes are templates for creating objects. They define the properties and behaviors of an object, including its methods and attributes. We can create multiple objects in a class. It is a logical entity that does not occupy any space/memory.

Examples using Java, Python and C#

We can declare a class with the use of a **class** keyword.

Java

Syntax

```
class  <class_name> {
field;
method;
}
```

Sample Code

```
public class Student {
    // Private fields, accessible only within the class
    private String name;
    private int age;

    // Public constructor to create Student objects
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Public getter method for the 'name' field
    public String getName() {
        return name;
    }
```

```
    // Public setter method for the 'name' field
    public void setName(String name) {
        this.name = name;
    }

    // Public getter method for the 'age' field
    public int getAge() {
        return age;
    }

    // Public setter method for the 'age' field
    public void setAge(int age) {
        this.age = age;
    }

    // Public method to display student information
    public void displayInfo() {
        System.out.println("Student Name: " + name);
        System.out.println("Student Age: " + age);
    }
}
```

In this example, the name and age fields are marked as private, which means they can only be accessed within the Student class itself.Public getter and setter methods (getName(), setName(), getAge(), setAge()) are provided to allow controlled access to these private fields from outside the class. The displayInfo() method is public and can be used to display the student's information.

Python

Syntax

```
class MyClass:
    # class definition goes here
```

Inside the class definition, you can define attributes (i.e., variables) and methods (i.e., functions). You can use the `__init__()` method to define a constructor, which is called when an instance of the class is created.

Sample Code

```
class Student:

    def __init__(self, name, age):

        # Private fields, accessible only within the class

        self._name = name

        self._age = age

    # Public getter method for the 'name' field

    def get_name(self):

        return self._name

    # Public setter method for the 'name' field

    def set_name(self, name):

        self._name = name

    # Public getter method for the 'age' field

    def get_age(self):

        return self._age

    # Public setter method for the 'age' field

    def set_age(self, age):

        self._age = age

    # Public method to display student information

    def display_info(self):

        print(f"Student Name: {self._name}")

        print(f"Student Age: {self._age}")
```

The `Student` class defines a simple student data structure. It has private fields `_name` and `_age`, which are accessed using getter and setter methods (`get_name`, `set_name`, `get_age`, `set_age`). The `display_info` method displays the student's name and age. In Python, fields are conventionally marked as private with an underscore (`_`) prefix, but access control is not enforced by the language.

C#

Syntax

```
<access specifier> class class_name
{
// member variables
// member methods
}
```

Sample Code

```
public class Student

{

    // Private fields, accessible only within the class

    private string _name;

    private int _age;

    // Public constructor to create Student objects

    public Student(string name, int age)

    {

        _name = name;

        _age = age;

    }

    // Public property for the 'name' field

    public string Name

    {

        get { return _name; }

        set { _name = value; }
```

```
    }

    // Public property for the 'age' field

    public int Age

    {

        get { return _age; }

        set { _age = value; }

    }

    // Public method to display student information

    public void DisplayInfo()

    {

        Console.WriteLine($"Student Name: {_name}");

        Console.WriteLine($"Student Age: {_age}");

    }

}
```

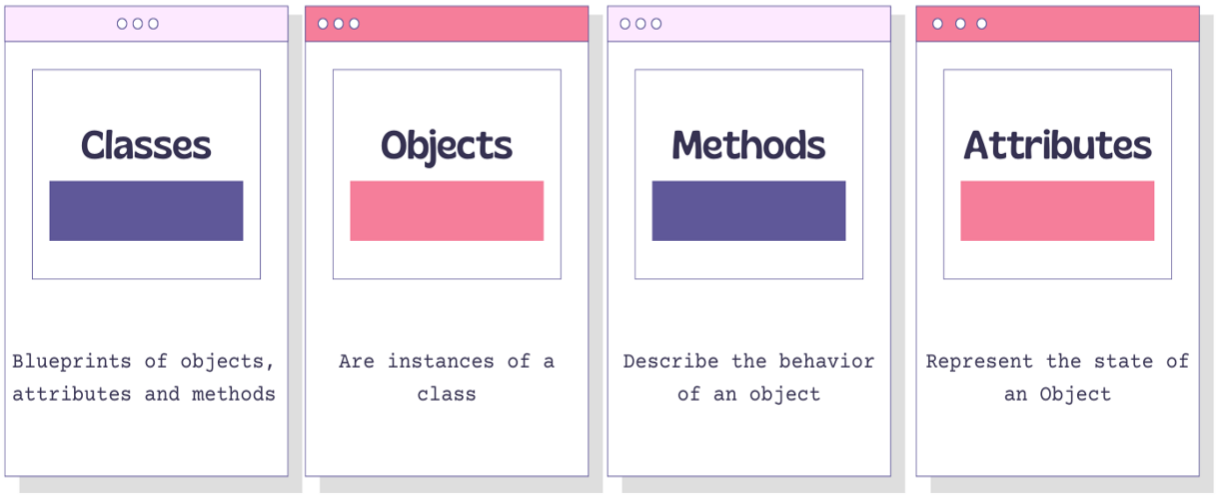
The `Student` class defines a student data structure. It has private fields `_name` and `_age`, which are encapsulated using public properties (`Name` and `Age`) for access control. The constructor initializes the student's name and age. The `DisplayInfo` method displays the student's name and age. In C#, properties provide controlled access to private fields, similar to Java's getter and setter methods.

Objects

Objects, on the other hand, are instances of classes. They are created from a class template and can have their own unique values for attributes. Object is a real-world entity such as pen, laptop, mobile etc. Object allocates memory when it is created.

Object is created through **new** keyword mainly, for example:

```
Student s1=new Student ()
```



Examples using Java, Python and C#

Java

Sample Code

```
class Student
{
      int id = 1001;
      String name = "Amy";
       public static void main(String args[])
       {
        Student s1=new Student();
       System.out.println(s1.id);
            System.out.println(s1.name);
      }
}
```

This code is a sample example for a class Student with attributes and behavior which includes object creation.

Python

Syntax

```
# Step 1: Define a class
class ClassName:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2

    def method1(self):
        # Method code goes here
        pass
# Step 2: Create an instance of the class
object_name = ClassName(value1, value2)

# Step 3: Access attributes and methods
```

```
print(object_name.attribute1)
print(object_name.attribute2)
object_name.method1()
```

Sample Code

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
     def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years
old.")

 # Create an instance/object of the Person class
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# Access object attributes
print(person1.name)  # Output: Alice
print(person2.age)   # Output: 30

# Call object methods
person1.greet()  # Output: Hello, my name is Alice, and I am 25 years old.
person2.greet()  # Output: Hello, my name is Bob, and I am 30 years old.
```

To create an instance of the `Person` class, we use the class name followed by parentheses and provide values for the `name` and `age` parameters. In this case, `person1` represents a person named "Alice" who is 25 years old, and `person2` represents a person named "Bob" who is 30 years old.

We can access the attributes of an object using dot notation. For example, `person1.name` gives us the value "Alice", and `person2.age` gives us the value 30.

The `greet()` method can be called on each object using dot notation followed by parentheses. This will print a greeting message that includes the object's `name` and `age` attributes. In this case, `person1.greet()` prints "Hello, my name is Alice, and I am 25 years old.", and `person2.greet()` prints "Hello, my name is Bob, and I am 30 years old.".

C#

Syntax

```
Class-name object-name = new Class-name ();
```

Sample Code

```
using System;
class Student
{
    int id = 1001;
    string name = “Amy”;

    static void Main(string[] args)
    {
        Student s1 = new Student();
        Console.WriteLine(s1.id);
        Console.WriteLine(s1.name);
    }
}
```

In the above code Class student has two variables name as ‘id’(int) and ‘name’(string). Inside the main() method an instance of the class student is created as s1.

Constructor

Constructors are special methods that are called when an object is created from a class. They are used to initialize the object's attributes and perform any other necessary setup. A constructor for a class has the same name as the class name. Constructors do not return any type while method(s) have the return type or void if does not return any value.

Java

In Java, a constructor is a special method with the same name as the class. It does not have a return type, not even void. Here is an example of a constructor in Java:

Syntax

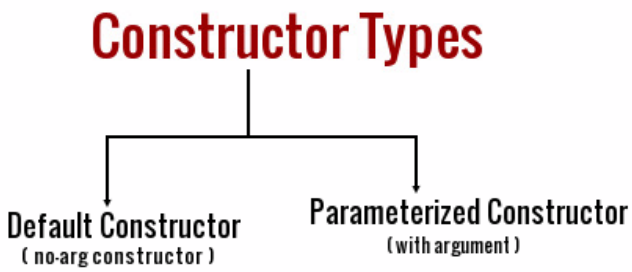
```
public class ExampleJava {
    private int value;

    // Constructor
    public ExampleJava(int initialValue) {
        value = initialValue;
    }

    public void printValue() {
        System.out.println("Value: " + value);
    }
    public static void main(String[] args) {
        ExampleJava obj = new ExampleJava(10);
        obj.printValue();
    }
}
```

There are two types of constructors.

1. Default Constructor
2. Parameterized Constructor



Default Constructor

A constructor that has no parameters is known as default the constructor. If a class does not define any constructor, then a default constructor is automatically generated by the compiler. The default constructor changed into the parameterized constructor. But Parameterized constructor cannot change the default constructor.

Parameterized Constructor

A parameterized constructor is a constructor that takes one or more parameters. It is used to create objects of a class with specific initial values for their properties. There can be more than one parameterized constructor in a class.

Sample Code

Here is an example for both default and parameterized constructors.

```
public class ExampleJava {
    private int value;

    // Default Constructor
    public ExampleJava() {
        value = 0;
```

```
        }

        // Parameterized Constructor
        public ExampleJava(int initialValue) {
            value = initialValue;
        }

        public void printValue() {
            System.out.println("Value: " + value);
        }

        public static void main(String[] args) {
            ExampleJava obj1 = new ExampleJava(); // Default Constructor
            obj1.printValue();

            ExampleJava obj2 = new ExampleJava(10); // Parameterized
Constructor
            obj2.printValue();
        }
    }
```

Python

In Python, a constructor is defined using the __init__ method. It is called automatically when an object is created. Here is a syntax of a constructor in Python:

Syntax

```
class ExamplePython:
    def __init__(self, initial_value):
        self.value = initial_value

    def print_value(self):
        print("Value:", self.value)

obj = ExamplePython(10)
```

```
obj.print_value()
```

Default Constructor

In Python, the __init__ method acts as the constructor, and it is automatically called when an object is created. It can take parameters to initialize the object's attributes.

Parameterized Constructor

The __init__ method can accept parameters to initialize the object's attributes.

Sample Code

Here is an example for both default and parameterized constructors.

```
class ExamplePython:
    def __init__(self):
        self.value = 0  # Default Constructor

    def __init__(self, initialValue):
        self.value = initialValue  # Parameterized Constructor

    def print_value(self):
        print("Value:", self.value)

obj1 = ExamplePython()  # Default Constructor
obj1.print_value()

obj2 = ExamplePython(10)  # Parameterized Constructor
obj2.print_value()
```

C#

In C#, a constructor is also defined using the same name as the class. Here is an example of a constructor in C#:

Syntax

```
public class ExampleCSharp
{
    private int value;

    // Constructor
    public ExampleCSharp(int initialValue)
    {
        value = initialValue;
    }

    public void PrintValue()
    {
        Console.WriteLine("Value: " + value);
    }

    public static void Main(string[] args)
    {
        ExampleCSharp obj = new ExampleCSharp(10);
        obj.PrintValue();
    }
}
```

Default Constructor

Like Java, a constructor with no parameters. If no constructor is explicitly defined, a default constructor is provided by the compiler.

Parameterized Constructor

A constructor that takes one or more parameters to initialize the object's member variables.

Sample Code

Here is an example for both default and parameterized constructors.

```
using System;

public class ExampleCSharp
{
    private int value;

    // Default Constructor
    public ExampleCSharp()
    {
        value = 0;
    }

    // Parameterized Constructor
    public ExampleCSharp(int initialValue)
    {
        value = initialValue;
    }

    public void PrintValue()
    {
        Console.WriteLine("Value: " + value);
    }

    public static void Main(string[] args)
    {
        ExampleCSharp obj1 = new ExampleCSharp(); // Default Constructor
        obj1.PrintValue();

        ExampleCSharp obj2 = new ExampleCSharp(10); // Parameterized
Constructor
        obj2.PrintValue();
    }
}
```

Copy Constructor

A constructor that creates a new object by copying the values from another object of the same class.

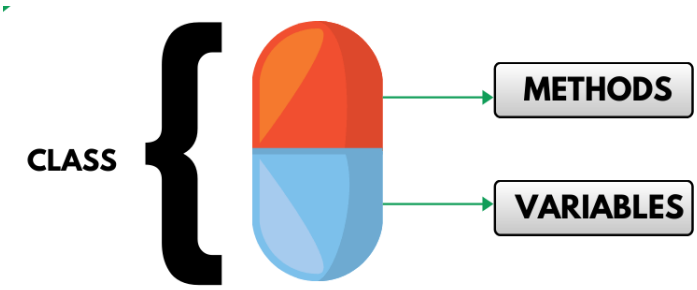
Static Constructor

A special constructor that is used to initialize static members of a class. It is called only once when the class is first accessed.

Encapsulation

Encapsulation is a fundamental pillar of object-oriented programming, which refers to the ability to hide implementation details and expose only necessary information through a well-defined interface. In simpler terms, it means that an object's internal state is protected from outside interference.

One of the key benefits of encapsulation is that it promotes modularity and code reusability. By hiding the implementation details, we can change the internal workings of an object without affecting the rest of the program. This makes it easier to maintain and update the codebase over time.



Before starting encapsulation, you need to understand access modifiers,

To encapsulate members of our class, we need to use certain access modifiers. These are keywords that specify the level of access, or protection, class members have.

Modifier	Description
public	No access restriction. Can be accessed from an object or any child class
private	Can only be accessed from within the class that defines it
protected	Can only be accessed from within the class that defines it, as well as any child class
default	Can only be accessed from within the current package

Points to Remember

- Encapsulation is when we define the access level of our classes.
- The public access modifier does not impose any restrictions.
- The private access modifier only allows members to be accessed within the current class, not outside of it (through an object).
- The protected access modifier allows the current class, and any child classes that inherit from it, to access its members.
- Accessor and mutator methods are used to access and modify private and protected members.

Implementation of Encapsulation using Java, Python and C#

Java

Syntax

```
class MyClass {
    private int myPrivateVariable; // private variable
    public int myPublicVariable; // public variable
    private void myPrivateMethod() { // private method
             // do something
    }
 public void myPublicMethod() { // public method
            // do something
    }
}
```

Sample Code

```
class Employee {
    private String name;
    private int age;
    private double salary;
    // Getter methods
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public double getSalary() {
        return salary;
    }
    // Setter methods
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
}
public class Main {
    public static void main(String[] args) {
        // Create an instance of Employee
        Employee employee = new Employee();
        // Set the employee details using setter methods
        employee.setName("John Doe");
        employee.setAge(30);
        employee.setSalary(50000.0);
        // Get and display employee details using getter methods
        System.out.println("Name: " + employee.getName());
        System.out.println("Age: " + employee.getAge());
```

```
        System.out.println("Salary: " + employee.getSalary());
    }
}
```

In this example, we have an employee class that encapsulates the data of an employee. The class has private member variables (name, age, and salary) that can only be accessed and modified through getter and setter methods.

The getter methods (getName(), getAge(), and getSalary()) allow us to retrieve the values of the private variables. The setter methods (setName(), setAge(), and setSalary()) allow us to set new values for the private variables, ensuring that we can control access to and modification of the data.

In the Main class, we create an instance of Employee, set the employee details using setter methods, and then retrieve and display the details using getter methods. By encapsulating the data and providing controlled access through getter and setter methods, we maintain the integrity of the data and establish encapsulation.

Python

Syntax

```
class MyClass:
    def __init__(self):
        self.public_attr = 10
        self._protected_attr = 20
        self.__private_attr = 30

    def public_method(self):
        # implementation of public method

    def _protected_method(self):
        # implementation of protected method

    def __private_method(self):
```

```
        # implementation of private method
```

In this example, the `MyClass` class defines three attributes with various levels of access, and three methods with the default public access. The public attribute and methods are defined without any access modifiers, while the protected and private attributes and methods are defined with the underscore and double underscore prefix, respectively.

Sample Code

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number    # public attribute
        self._balance = balance                 # protected attribute
        self.__pin = 1234                       # private attribute

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        if amount <= self._balance:
            self._balance -= amount
        else:
            print("Insufficient balance.")

    def get_balance(self):
        return self._balance

    def __validate_pin(self, pin):
        return self.__pin == pin

    def change_pin(self, old_pin, new_pin):
        if self.__validate_pin(old_pin):
            self.__pin = new_pin
            print("PIN changed successfully.")
        else:
            print("Invalid PIN.")
```

In this example, the `BankAccount` class has three attributes - `account_number`, `_balance`, and `__pin`, with public, protected, and private access modifiers, respectively. The class has four methods - `deposit()`, `withdraw()`, `get_balance()`, and `change_pin()`, with public access.

The `deposit()` and `withdraw()` methods modify the `_balance` attribute, which is a protected attribute. The `get_balance()` method returns the `_balance` attribute. The `change_pin()` method changes the value of the `__pin` attribute, which is a private attribute. The `__validate_pin()` method is a private method that is used internally to validate the PIN.

By using access modifiers, the implementation details of the class are hidden from the outside world, and the public interface is provided through the public methods. \

C#

Syntax

```
AccessModifier datatype variable_name;
AccessModifier datatype method_name();

public class MyClass
{
    public int public_attr = 10;
    protected int protected_attr = 20;
    private int private_attr = 30;

    public void public_method()
    {
        // implementation of public method
    }

    protected void protected_method()
    {
        // implementation of protected method
    }

    private void private_method()
    {
        // implementation of private method
    }
}
```

In this example, the `MyClass` class defines three attributes with various levels of access, and three methods with the default public access. The public attribute and methods are defined without any access modifiers, while the protected and private attributes and methods are defined with the underscore and double underscore prefix, respectively.

Sample Code

```
public class BankAccount
{
    public int account_number;    // public attribute
    protected decimal balance;    // protected attribute
    private int pin;              // private attribute

    public BankAccount(int account_number, decimal balance)
    {
        this.account_number = account_number;
        this.balance = balance;
        this.pin = 1234;
    }

    public void Deposit(decimal amount)
    {
        balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= balance)
        {
            balance -= amount;
        }
        else
        {
            Console.WriteLine("Insufficient balance.");
```

```
        }
    }

    public decimal GetBalance()
    {
        return balance;
    }

    private bool ValidatePin(int pin)
    {
        return this.pin == pin;
    }

    public void ChangePin(int old_pin, int new_pin)
    {
        if (ValidatePin(old_pin))
        {
            pin = new_pin;
            Console.WriteLine("PIN changed successfully.");
        }
        else
        {
            Console.WriteLine("Invalid PIN.");
        }
    }
}
```

In this example, we have an Employee class that encapsulates the data of an employee. The class has private member variables (name, age, and salary) that can only be accessed and modified through getter and setter methods.

The getter methods (getName(), getAge(), and getSalary()) allow us to retrieve the values of the private variables. The setter methods (setName(), setAge(), and setSalary()) allow us to set new values for the private variables, ensuring that we can control access to and modification of the data.

In the Main class, we create an instance of Employee, set the employee details using setter methods, and then retrieve and display the details using getter methods. By encapsulating the data and providing controlled access through getter and setter methods, we maintain the integrity of the data and establish encapsulation.

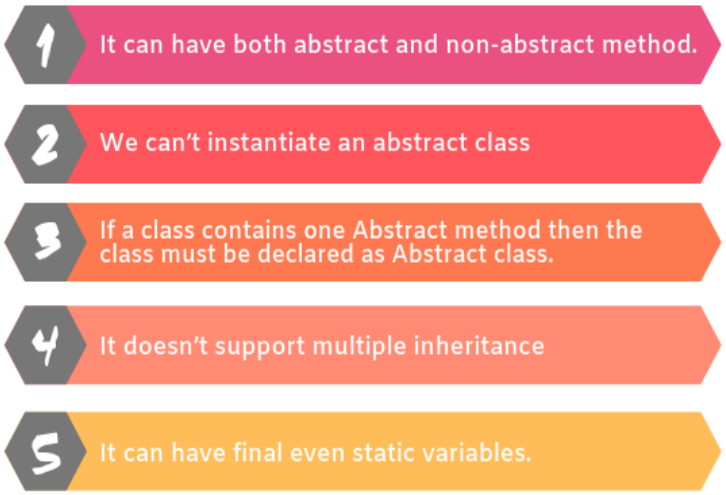
Abstraction

Abstraction refers to the process of extracting essential features of an object and ignoring unnecessary details. It allows us to focus on the most critical aspects of an object and ignore the rest, making the code more manageable and easier to understand. One of the primary benefits of abstraction is that it promotes simplicity and reduces complexity. It helps to improve the security of an application or program as only vital details are provided to the user.

There are 2 ways to achieve abstraction, they are as follows:

1. Abstract Class
2. Interface

Points to Remember



Implementation of Abstraction using Java, Python and C#

Java

Sample Code

```
abstract class Shape
{
// abstract method
      public abstract void draw();
      }
      class Circle extends Shape {
// implementation of draw() method
      public void draw() {
      System.out.println("Drawing Circle");
      }
      }
      class Rectangle extends Shape {
// implementation of draw() method
      public void draw() {
      System.out.println("Drawing Rectangle");
      }
}
```

In this example, the Shape class is an abstract class that defines an abstract method called draw (). The **Circle** and **Rectangle** classes extend the Shape class and implement the draw () method with their own implementation.

In the Main class, we create objects of the **Circle** and **Rectangle** classes and assign them to a Shape reference variable. We can then call the draw () method on these objects.

without knowing the actual implementation details of the method, since it is declared as abstract in the Shape class.

This is an example of abstraction, as we are hiding the implementation details of the draw () method from the users of the **Shape, Circle,** and **Rectangle** classes.

Python

Syntax

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass
```

In this example, the `Shape` class is an abstract class that contains two abstract methods `area()` and `perimeter()`. Any class that inherits from the `Shape` class must implement these methods, or it will also be considered an abstract class.

Sample Code

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
```

```
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

    def perimeter(self):
        return 2 * 3.14 * self.radius

# Instantiate the Rectangle class
rectangle = Rectangle(4, 5)
print("Rectangle area:", rectangle.area())
print("Rectangle perimeter:", rectangle.perimeter())

# Instantiate the Circle class
circle = Circle(3)
print("Circle area:", circle.area())
print("Circle perimeter:", circle.perimeter())
```

In this example, we have created an abstract class `Shape` that contains two abstract methods `area()` and `perimeter()`. These methods are not implemented in the `Shape` class but are defined as abstract methods using the `@abstractmethod` decorator. The `Shape` class is meant to be inherited by other classes that will implement these methods.

We have then created two child classes `Rectangle` and `Circle` that inherited from the `Shape` class and implement the `area()` and `perimeter()` methods according to their own functionality.

Finally, we have instantiated objects of the `Rectangle` and `Circle` classes and called their `area()` and `perimeter()` methods to calculate the area and perimeter of a rectangle and circle, respectively.

C#

Sample Code

```
abstract class Shape
{
// abstract method
      public abstract void draw ();
      }
      class Circle: Shape {
// implementation of draw () method
      public void draw () {
      Console.WriteLine ("Drawing Circle");
      }
      }
      class Rectangle: Shape {
// implementation of draw () method
      public void draw () {
      Console.WriteLine ("Drawing Rectangle");
      }
}
```

In the above code shape is defined as an abstract class having an abstract method draw. Class circle is extending abstract class shape and providing its own implementation for draw method. Rectangle class also extending shape class and providing its own implementation for draw method.

Access Modifiers

Access modifiers specify the access control wherever I am placing it or using it in classes, variables, and methods. There are four different access modifiers:

1.Public: When a class, method, or variable is declared as public, it can be accessed by any other code in the same package or in a different package. This is the most permissive access modifier.

2. **Protected:** When a class, method, or variable is declared as protected, it can be accessed by any code in the same package or in a subclass, regardless of whether the subclass is in the same package or not.

3. **Private:** When a class, method, or variable is declared as private, it can only be accessed within the same class. This is the most restrictive access modifier. Here the data is most secure.

4**. Default:** When a class, method, or variable is declared without any access modifier, it can only be accessed by code in the same package. This is the default access modifier.

Java

```
public class MyClass {
    public int publicVariable;
    private int privateVariable;
    protected int protectedVariable;
    int defaultVariable;

    public void publicMethod() {
        // Accessible from anywhere
    }

    private void privateMethod() {
        // Accessible only within the same class
    }
```

```
    protected void protectedMethod() {
        // Accessible within the same class and subclasses
    }

    void defaultMethod() {
        // Accessible within the same package
    }
}
```

Python

```
class MyClass:
    def __init__(self):
        self.public_variable = 10
        self._private_variable = 20
        self.__mangled_variable = 30

    def public_method(self):
        # Accessible from anywhere
        pass

    def _private_method(self):
        # Accessible within the same class (convention, not enforced by the
language)
        pass
```

C#

```
public class MyClass
{
    public int publicVariable;
    private int privateVariable;
    protected int protectedVariable;
```

```
    internal int internalVariable;
    protected internal int protectedInternalVariable;

    public void PublicMethod()
    {
        // Accessible from anywhere
    }

    private void PrivateMethod()
    {
        // Accessible only within the same class
    }

    protected void ProtectedMethod()
    {
        // Accessible within the same class and subclasses
    }

    internal void InternalMethod()
    {
        // Accessible within the same assembly
    }

    protected internal void ProtectedInternalMethod()
    {
        // Accessible within the same assembly and subclasses
    }
}
```

In the examples above, you can see the usage of different access modifiers in Java, Python, and C#. Each access modifier defines the visibility and accessibility of the class members and controls how they can be accessed from within the program.

Inheritance

Inheritance is another essential pillar of OOP, which allows us to create new classes based on existing ones. The new class inherits all the properties and methods of the parent class, and we can add or modify them as needed.

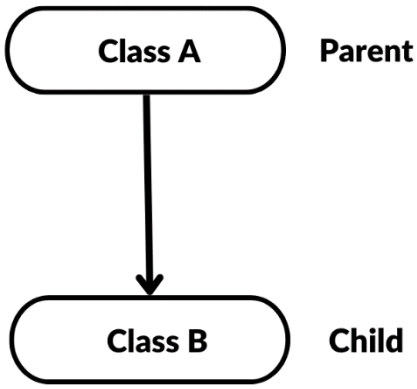
One of the primary benefits of inheritance is that it promotes code reuse and reduces duplication. Instead of writing the same code multiple times, we can create a base class and extend it, as necessary. This makes the code more manageable and easier to maintain.

Different types of Inheritance

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

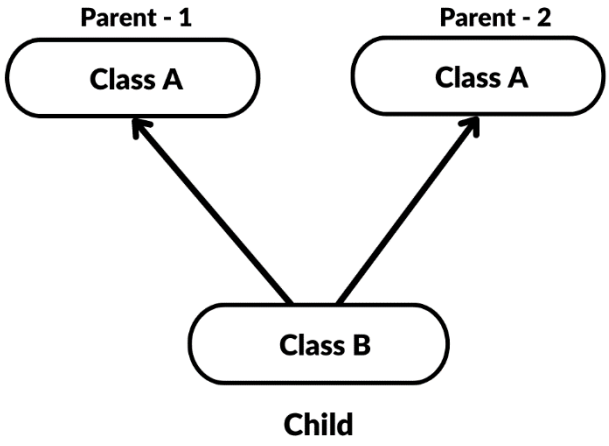
Single Inheritance

In single inheritance, child class will extend only one parent class. Therefore, Class B is extending Class A.



Multiple Inheritance

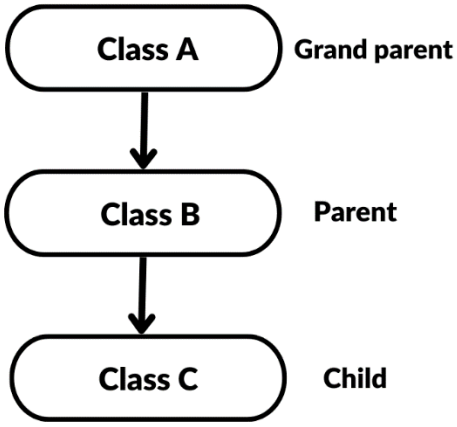
In multiple inheritance, child class will extend more than one class. Also, Multiple Inheritance is not supported in Java.



Multilevel Inheritance

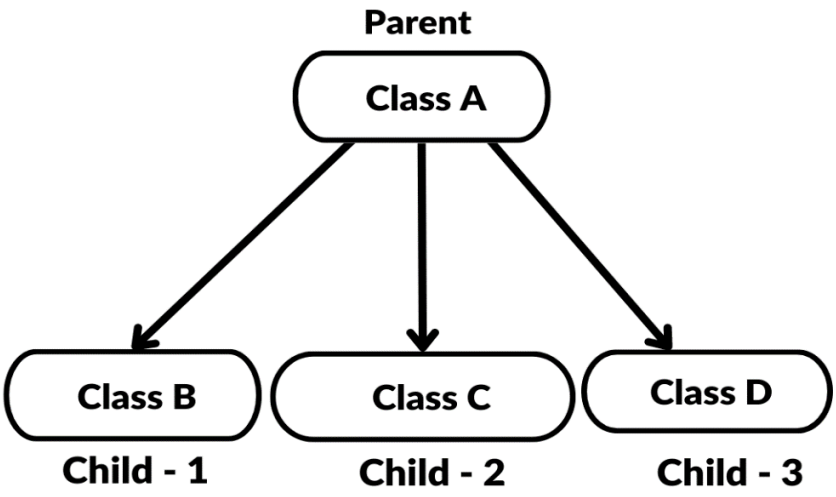
In multilevel inheritance, one class can **extend** from a derived class and next time derived class will become the parent class for a new class. Therefore, code defined in Class A can be easily accessible to Class C in a multilevel manner.

Here you can see Class C extends to Class B where Class B is a parent to Class C. In addition, when Class B extends Class A, then Class B will become the child to Class A.



Hierarchical Inheritance

In hierarchical inheritance, one class is inherited by many other child classes. Hierarchical inheritance is a type of inheritance in object-oriented programming in which a single base class is inherited by multiple derived classes.



Implementation of Inheritance using Java, Python and C#

Java

Single Inheritance

Syntax

```
class base class
{
     .... methods
}
class derivedClass name extends baseClass
{
    methods ... along with this additional feature
}
```

Sample Code

```
// Parent class
class Parent {
    public void parentMethod() {
        System.out.println("This is a parent method.");
    }
}
// Child class inheriting from Parent
class Child extends Parent {
    public void childMethod() {
        System.out.println("This is a child method.");
    }
}
```

In this example, the Child class extends the Parent class using the extends keyword, which establishes a parent-child relationship. The Parent class has a single method called parentMethod(), and the Child class has a single method called childMethod()

Multiple Inheritance

Sample Code

```
    interface Parent1 {
    void method1();
}
interface Parent2 {
    void method2();
}
class Child implements Parent1, Parent2 {
    public void method1() {
        System.out.println("This is method 1 from Parent 1");
    }

    public void method2() {
        System.out.println("This is method 2 from Parent 2");
    }
    public void method3() {
        System.out.println("This is method 3 from Child");
    }
}
public class MultipleInheritanceExample {
    public static void main(String[] args) {
        Child child = new Child();

        // Access methods from Parent1
        child.method1();

        // Access methods from Parent2
        child.method2();

        // Access method from Child
        child.method3();
    }
}
```

Multilevel Inheritance

Sample Code

```
      class Grandparent {
      void method1() {
      System.out.println("This is method 1 from Grandparent");
      }
      }
      class Parent extends Grandparent {
      void method2() {
      System.out.println("This is method 2 from Parent");
      }
      }
      class Child extends Parent {
      void method3() {
      System.out.println("This is method 3 from Child");
      }
      }
      public class Main {
      public static void main(String[] args) {
              Child childObj = new Child();
              childObj.method1();  // Output: This is method 1 from
Grandparent
              childObj.method2();  // Output: This is method 2 from Parent
              childObj.method3();  // Output: This is method 3 from Child
          }
      }
```

In this example, Child class is derived from Parent class, which is derived from Grandparent class. The Grandparent class defines method1, which is inherited by Parent class. The Parent class defines method2, which is inherited by Child class. The Child class also defines its own method method3.

When an instance of Child class is called method1, the method inherited from Grandparent class is executed. Similarly, when an instance of Child class calls method2, the method inherited from Parent class is executed. And when an instance of Child class is called method3, the method defined in Child class itself is executed.

Hierarchical Inheritance

Sample Code

```
class Grandparent {
    void method1() {
        System.out.println("This is method 1 from Grandparent");
    }
}

class Parent1 extends Grandparent {
    void method2() {
        System.out.println("This is method 2 from Parent1");
    }
}

class Parent2 extends Grandparent {
    void method3() {
        System.out.println("This is method 3 from Parent2");
    }
}

public class Main {
    public static void main(String[] args) {
        Parent1 parent1Obj = new Parent1();
        Parent2 parent2Obj = new Parent2();

        parent1Obj.method1();    // Output: This is method 1 from
Grandparent
        parent1Obj.method2();  // Output: This is method 2 from Parent1
```

```
        parent2Obj.method1();    // Output: This is method 1 from
Grandparent
        parent2Obj.method3();  // Output: This is method 3 from Parent2
    }
}
```

In this example, the `Parent1` and `Parent2` classes both inherit from the `Grandparent` class, demonstrating hierarchical inheritance.

Hybrid Inheritance

Sample Code

```
  // Define an interface
interface Interface1 {
    void method4();
}

class Grandparent {
    void method1() {
        System.out.println("This is method 1 from Grandparent");
    }
}

class Parent extends Grandparent {
    void method2() {
        System.out.println("This is method 2 from Parent");
    }
}

class Child extends Parent implements Interface1 {
    void method3() {
```

```
        System.out.println("This is method 3 from Child");
    }

    @Override
    public void method4() {
        System.out.println("This is method 4 from Interface1");
    }
}

public class Main {
    public static void main(String[] args) {
        Child childObj = new Child();
        childObj.method1();    //  Output:  This  is  method  1  from
Grandparent
        childObj.method2();  // Output: This is method 2 from Parent
        childObj.method3();  // Output: This is method 3 from Child
        childObj.method4();    //  Output:  This  is  method  4  from
Interface1
    }
}
```

In this example, the `Child` class inherits from `Parent` (multilevel inheritance) and also implements `Interface1` (multiple inheritance through interfaces), demonstrating hybrid inheritance.

Python

Single Inheritance

Syntax

```
class ParentClass:
    pass
```

```
class ChildClass(ParentClass):
    pass
```

Sample Code

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("This animal speaks.")

# Child class inheriting from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        # Calling the constructor of the parent class
        super().__init__(name)
        self.breed = breed

    def speak(self):
        print("Woof!")

# Creating objects of the classes
animal = Animal("Generic Animal")
dog = Dog("Max", "German Shepherd")

# Calling methods of the classes
animal.speak()  # Output: This animal speaks.
dog.speak()     # Output: Woof!
print(dog.name) # Output: Max
print(dog.breed) # Output: German Shepherd
```

In this example, `Animal` is the parent class and `Dog` is the child class. `Dog` inherits the `__init__()` method and the `speak()` method from `Animal`. The `__init__()` method of `Dog` calls the `__init__()` method of `Animal` using the `super()` function.

`Dog` also defines its own `speak()` method, which overrides the `speak()` method of `Animal`. When you create an object of `Dog` and call the `speak()` method, it prints "Woof!" instead of the default message "This animal speaks.". This is an example of single inheritance, as `Dog` extends a single parent class `Animal`.

Multiple Inheritance

Syntax

```
class ParentClass1:
    pass

class ParentClass2:
    pass

class ChildClass(ParentClass1, ParentClass2):
    pass
```

Sample Code

```
# Parent class 1
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years
old.")

# Parent class 2
class Employee:
    def __init__(self, emp_id, salary):
        self.emp_id = emp_id
        self.salary = salary

    def work(self):
```

```
        print("This employee is working.")

# Child class inheriting from Person and Employee
class Manager(Person, Employee):
    def __init__(self, name, age, emp_id, salary, department):
        # Calling the constructors of the parent classes
        Person.__init__(self, name, age)
        Employee.__init__(self, emp_id, salary)
        self.department = department

    def manage(self):
        print("This manager is managing.")

# Creating an object of the Manager class
manager = Manager("John", 35, "M001", 50000, "Sales")

# Calling methods of the classes
manager.greet()  # Output: Hello, my name is John, and I am 35 years old.
manager.work()   # Output: This employee is working.
manager.manage() # Output: This manager is managing.
print(manager.department) # Output: Sales
```

In this example, `Person` and `Employee` are the parent classes and `Manager` is the child class. `Manager` inherits the `__init__()` method and the `greet()` method from `Person` and the `__init__()` method and the `work()` method from `Employee`.

`Manager` defines its own `manage()` method. When you create an object of `Manager` and call the `greet()`, `work()` and `manage()` methods, it prints the appropriate output. This is an example of multiple inheritance, as `Manager` extends multiple parent classes `Person` and `Employee`.

Multilevel Inheritance

Syntax

```
class GrandparentClass:
    pass

class ParentClass(GrandparentClass):
    pass

class ChildClass(ParentClass):
    pass
```

Sample Code

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("This animal speaks.")

# Child class inheriting from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        # Calling the constructor of the parent class
        super().__init__(name)
        self.breed = breed

    def speak(self):
        print("Woof!")
# Grandchild class inheriting from Dog
class GermanShepherd(Dog):
    def __init__(self, name):
        # Calling the constructor of the parent class
        super().__init__(name, "German Shepherd")
```

```
    def guard(self):
        print("This German Shepherd is guarding.")

# Creating an object of the GermanShepherd class
german_shepherd = GermanShepherd("Max")

# Calling methods of the classes
german_shepherd.speak()    # Output: Woof!
german_shepherd.guard()    # Output: This German Shepherd is guarding.
print(german_shepherd.name)    # Output: Max
print(german_shepherd.breed)   # Output: German Shepherd
```

In this example, `Animal` is the parent class, `Dog` is the child class and `GermanShepherd` is the grandchild class.

`Dog` inherits the `__init__()` method and the `speak()` method from `Animal`. `GermanShepherd` inherits the `__init__()` method and the `speak()` method from `Dog`. `GermanShepherd` also defines its own `guard()` method.

When you create an object of `GermanShepherd` and call the `speak()`, `guard()` and access the `name` and `breed` attributes, it prints the appropriate output.
This is an example of multilevel inheritance, as `GermanShepherd` extends `Dog`, which itself extends `Animal`.

Hierarchical Inheritance

Syntax

```
class ParentClass:
    pass

class ChildClass1(ParentClass):
    pass
```

```
class ChildClass2(ParentClass):
    pass
```

Sample Code

```
# Parent class
class Shape:
    def __init__(self, color):
        self.color = color

    def draw(self):
        print(f"This {self.color} shape is drawing.")

# Child class 1 inheriting from Shape
class Circle(Shape):
    def __init__(self, color, radius):
        # Calling the constructor of the parent class
        super().__init__(color)
        self.radius = radius

    def draw(self):
         print(f"This {self.color} circle with radius {self.radius} is
drawing.")

# Child class 2 inheriting from Shape
class Square(Shape):
    def __init__(self, color, side):
        # Calling the constructor of the parent class
        super().__init__(color)
        self.side = side

    def draw(self):
        print(f"This {self.color} square with side {self.side} is drawing.")

# Creating objects of the Circle and Square classes
circle = Circle("red", 5)
```

```
square = Square("blue", 10)

# Calling methods of the classes
circle.draw()   # Output: This red circle with radius 5 is drawing.
square.draw()   # Output: This blue square with side 10 is drawing.
```

In this example, `Shape` is the parent class and `Circle` and `Square` are the child classes. Both `Circle` and `Square` inherit the `__init__()` method and the `draw()` method from `Shape`.
When you create objects of `Circle` and `Square` and call the `draw()` method, it prints the appropriate output.

This is an example of hierarchical inheritance, as both `Circle` and `Square` inherited from the same parent class `Shape`.

Hybrid Inheritance

Syntax

```
class GrandparentClass:
    pass

class ParentClass1(GrandparentClass):
    pass

class ParentClass2(GrandparentClass):
    pass

class ChildClass(ParentClass1, ParentClass2):
    pass
```

Sample Code

```
# Parent class
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

# Child class 1 inheriting from Vehicle
class Car(Vehicle):
    def __init__(self, make, model, num_doors):
        # Calling the constructor of the parent class
        super().__init__(make, model)
        self.num_doors = num_doors

# Child class 2 inheriting from Vehicle
class Motorcycle(Vehicle):
    def __init__(self, make, model, num_wheels):
        # Calling the constructor of the parent class
        super().__init__(make, model)
        self.num_wheels = num_wheels

# Child class 3 inheriting from both Car and Motorcycle
class Hybrid(Car, Motorcycle):
    def __init__(self, make, model, num_doors, num_wheels):
        # Calling the constructors of the parent classes
        Car.__init__(self, make, model, num_doors)
        Motorcycle.__init__(self, make, model, num_wheels)

# Creating an object of the Hybrid class
hybrid = Hybrid("Toyota", "Prius", 4, 2)

# Accessing the attributes of the object
print(hybrid.make)          # Output: Toyota
print(hybrid.model)         # Output: Prius
print(hybrid.num_doors)     # Output: 4
print(hybrid.num_wheels)    # Output: 2
```

In this example, `Vehicle` is the parent class, `Car` and `Motorcycle` are the child classes that inherit from `Vehicle`, and `Hybrid` is the child class that inherits from both `Car` and `Motorcycle`.

`Hybrid` class constructor calls the constructors of both `Car` and `Motorcycle` classes. This demonstrates the concept of multiple inheritance in Python.

When you create an object of `Hybrid` and access its attributes, it prints the appropriate output.

This is an example of hybrid inheritance, as `Hybrid` class is inheriting from both `Car` and `Motorcycle` classes.

C#

Single Inheritance

Syntax

```
public class ParentClass
{
    // Implementation of the parent class
}

public class ChildClass : ParentClass
{
    // Implementation of the child class inheriting from ParentClass
}
```

Sample Code

```
using System;

public class Animal
{
    public string name;

    public Animal(string name)
    {
        this.name = name;
    }
      public virtual void Speak()
    {
        Console.WriteLine("This animal speaks.");
    }
}
  public class Dog : Animal
{
    public string breed;
```

```
    public Dog(string name, string breed) : base(name)
    {
        this.breed = breed;
    }

    public override void Speak()
    {
        Console.WriteLine("Woof!");
    }
}
  class Program
{
    static void Main(string[] args)
    {
        Animal animal = new Animal("Generic Animal");
        Dog dog = new Dog("Max", "German Shepherd");

        animal.Speak();     // Output: This animal speaks.
        dog.Speak();        // Output: Woof!
        Console.WriteLine(dog.name);   // Output: Max
        Console.WriteLine(dog.breed);  // Output: German Shepherd
    }
}
```

In this example, `Animal` is the parent class and `Dog` is the child class. `Dog` inherits the constructor and the `Speak ()` method from `Animal`. The constructor of `Dog` calls the constructor of `Animal` using the base keyword.

`Dog` also defines its own `speak()` method, which overrides the `speak()` method of `Animal`. When you create an object of `Dog` and call the `speak()` method, it prints "Woof!" instead of the default message "This animal speaks.".
This is an example of single inheritance, as `Dog` extends a single parent class `Animal`.

Multiple Inheritance

Syntax

```
public class ParentClass1
{
    // Implementation of the first parent class
}

public class ParentClass2
{
    // Implementation of the second parent class
}

public class ChildClass : ParentClass1, ParentClass2
{
    // Implementation of the child class inheriting from ParentClass1 and
ParentClass2
}
```

In Multiple Inheritance, one child or subclass class can have more than one base class or superclass and inherit features from every parent class which it inherits. But Multiple Inheritance is not supported by C#.

Multilevel Inheritance

Syntax

```
public class GrandparentClass
{
    // Implementation of the grandparent class
}

public class ParentClass : GrandparentClass
{
    // Implementation of the parent class inheriting from GrandparentClass
}
```

```
public class ChildClass : ParentClass
{
    // Implementation of the child class inheriting from ParentClass
}
```

Sample Code

```
using System;

public class Animal
{
    public string name;

    public Animal(string name)
    {
        this.name = name;
    }

    public virtual void Speak()
    {
        Console.WriteLine("This animal speaks.");
    }
}

public class Dog : Animal
{
    public string breed;

    public Dog(string name, string breed) : base(name)
    {
        this.breed = breed;
    }

    public override void Speak()
    {
        Console.WriteLine("Woof!");
    }
```

```
}

public class GermanShepherd : Dog
{
    public GermanShepherd(string name) : base(name, "German Shepherd")
    {
    }

    public void Guard()
    {
        Console.WriteLine("This German Shepherd is guarding.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        GermanShepherd germanShepherd = new GermanShepherd("Max");

        germanShepherd.Speak();          // Output: Woof!
        germanShepherd.Guard();          // Output: This German Shepherd is
guarding.
        Console.WriteLine(germanShepherd.name);    // Output: Max
        Console.WriteLine(germanShepherd.breed);   // Output: German
Shepherd
    }
}
```

In this example, `Animal` is the parent class, `Dog` is the child class and `GermanShepherd` is the grandchild class.

`Dog` inherits the constructor and the `speak()` method from `Animal`. `GermanShepherd` inherits the constructor and the `speak()` method from `Dog`. `GermanShepherd` also defines its own `guard()` method.

When you create an object of `GermanShepherd` and call the `speak()`, `guard()` and access the `name` and `breed` attributes, it prints the appropriate output.
This is an example of multilevel inheritance, as `GermanShepherd` extends `Dog`, which itself extends `Animal`.

Hierarchical Inheritance

Syntax

```
public class ParentClass
{
    // Implementation of the parent class
}

public class ChildClass1 : ParentClass
{
    // Implementation of the first child class inheriting from ParentClass
}

public class ChildClass2 : ParentClass
{
    // Implementation of the second child class inheriting from ParentClass
}
```

Sample Code

```
using System;

public class Shape
{
    public string color;

    public Shape(string color)
    {
```

```
        this.color = color;
    }

    public virtual void Draw()
    {
        Console.WriteLine($"This {color} shape is drawing.");
    }
}

public class Circle : Shape
{
    public int radius;

    public Circle(string color, int radius) : base(color)
    {
        this.radius = radius;
    }

    public override void Draw()
    {
        Console.WriteLine($"This {color} circle with radius {radius} is
drawing.");
    }
}
 public class Square : Shape
{
    public int side;

    public Square(string color, int side) : base(color)
    {
        this.side = side;
    }
      public override void Draw()
    {
        Console.WriteLine($"This {color} square with side {side} is
drawing.");
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Circle circle = new Circle("red", 5);
        Square square = new Square("blue", 10);

        circle.Draw();  // Output: This red circle with radius 5 is
drawing.
        square.Draw();  // Output: This blue square with side 10 is
drawing.
    }
}
```

In this example, `Shape` is the parent class and `Circle` and `Square` are the child classes. Both `Circle` and `Square` inherit the Constructor and the `draw()` method from `Shape`. When you create objects of `Circle` and `Square` and call the `draw()` method, it prints the appropriate output.

This is an example of hierarchical inheritance, as both `Circle` and `Square` inherited from the same parent class `Shape`.

Hybrid Inheritance

Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance.

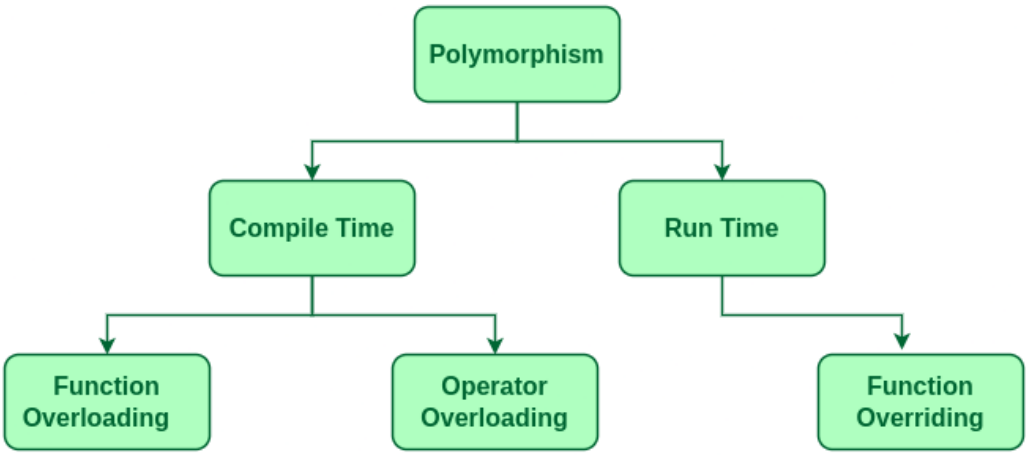
Hybrid inheritance is not directly supported by c#.

Polymorphism

Polymorphism is one of the fundamental concepts in object-oriented programming. It refers to the ability of an object to take on many forms or to have multiple behaviors depending on the context in which it is used.

In other words, polymorphism allows objects of different classes to be treated as if they were of the same class, making it easier to write code that can work with two diverse types of objects.

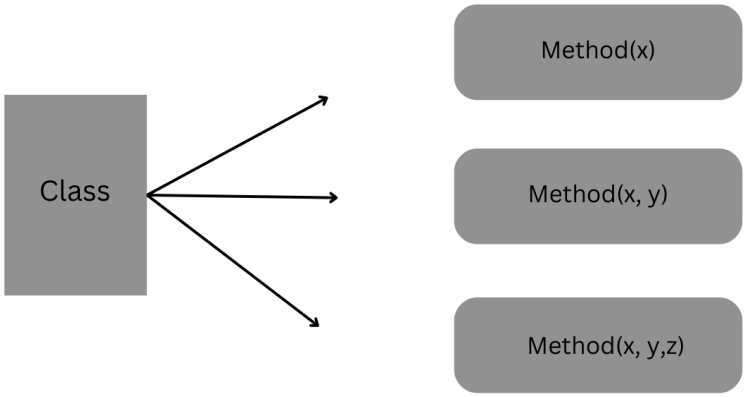
1. Method Overloading or Compile-time Polymorphism
2. Method Overriding or Runtime Polymorphism



Method Overloading or Compile-time Polymorphism

One way to achieve static polymorphism is through method overloading. Method overloading allows multiple methods with the same name to exist in the same class if they have different parameter lists.

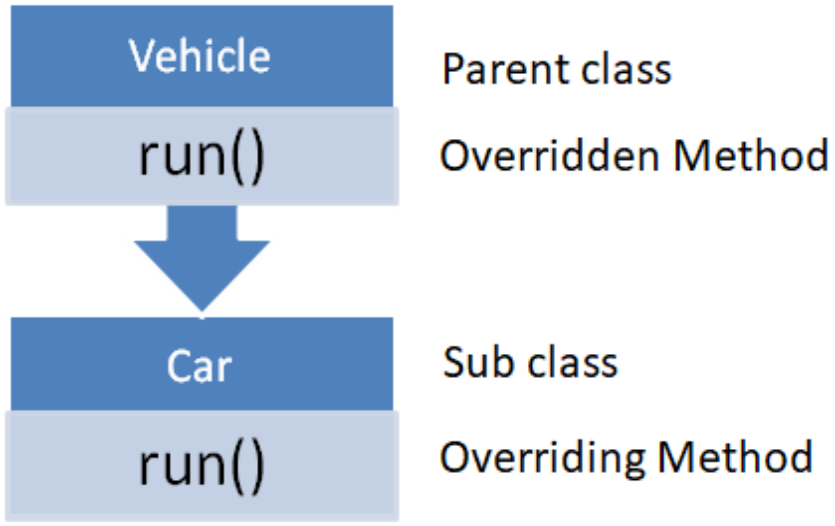
When a method is called, the compiler determines which version of the method to use based on the arguments passed to it. This allows for more flexibility in method design and makes it easier to reuse code.



Method Overriding or Runtime Polymorphism

Method overriding is an example of runtime polymorphism, which allows different objects to respond to the same message in diverse ways at ways at runtime. Method overriding also allows for better code reuse.

When a class is inherited and we declare a method in the subclass which is already present in the base class, it is referred to as method overriding. Overriding is done so that the subclass could define and implement the method specific to the subclass type i.e., the method defined in the subclass will override the functionality of the existing method of the super or base class. Hence, the method of the parent class is called the overridden method and the method in the subclass, or the child class is known as the overriding method.



Implementation of Polymorphism using Java, Python and C#

Java

Method Overloading

Syntax

```
public class Calculator {
   public int add(int a, int b) {
      return a + b;
   }
   public double add(double a, double b) {
      return a + b;
   }
}
```

In this example, we have two add() methods with the same name but different parameter types (int and double). Depending on the arguments passed to the method, the appropriate method will be called at compile time.

Sample Code

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b) {
        return a + b;
    }
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
    int result1 = calculator.add(2, 3);
        System.out.println("Result (int): " + result1);
        double result2 = calculator.add(2.5, 3.7);
        System.out.println("Result (double): " + result2);
    }
```

```
}
```

In this example, the Calculator class has two added methods with different parameter types (int and double). This is an example of method overloading, where multiple methods have the same name but differ in the number or types of parameters.

The add method with integer parameters performs integer addition, and the add method with double parameters performs double addition. This allows you to add two integers or two doubles using the appropriate method based on the parameter types.

In the main method, we create an instance of Calculator and invoke the added methods with different arguments. We print the results to the console, demonstrating the use of the overloaded methods.

Method Overriding

Syntax

```
class ParentClass {
    public void someMethod() {
        // implementation goes here
}
}
class ChildClass extends ParentClass {
    //Override
    public void someMethod() {
        // new implementation goes here
 }
}
```

In this example, ParentClass is the parent class or superclass, and ChildClass is the child class or subclass. The method someMethod() is defined in ParentClass and is overridden in ChildClass.

Sample Code

```
class ParentClass {
    public void someMethod() {
        System.out.println("ParentClass - someMethod");
    }
}

class ChildClass extends ParentClass {
    @Override
    public void someMethod() {
        System.out.println("ChildClass - someMethod");
    }
}
public class Main {
    public static void main(String[] args) {
        ParentClass parent = new ParentClass();
        parent.someMethod(); // Output: "ParentClass - someMethod"

        ChildClass child = new ChildClass();
        child.someMethod(); // Output: "ChildClass - someMethod"

        ParentClass polymorphicChild = new ChildClass();
        polymorphicChild.someMethod(); // Output: "ChildClass - someMethod"
    }
}
```

In this example, ParentClass has a method called someMethod(). The ChildClass extends the ParentClass and overrides the someMethod() with its own implementation.

In the main() method, we create instances of both the ParentClass and ChildClass to demonstrate the behavior of method overriding. When we invoke the someMethod() on the ParentClass instance, it executes the implementation defined in the ParentClass. When we invoke the someMethod() on the ChildClass instance, it executes the overridden implementation defined in the ChildClass.

Furthermore, we demonstrate polymorphism by assigning an instance of ChildClass to a variable of type ParentClass. Even though the reference type is ParentClass, when we invoke the someMethod(), it still executes the overridden implementation in the ChildClass. This is an example of runtime polymorphism in Java.

Python

Method Overloading

Syntax

```
class MyClass:
    def my_method(self, arg1, arg2=None, arg3=None):
        # First version of the method

    def my_method(self, arg1, arg2):
        # Second version of the method

    def my_method(self, *args):
        # Third version of the method
```

In this syntax, `MyClass` class defines three versions of the `my_method` method with the same name, but different parameter lists. The first version takes three arguments, the second version takes two arguments, and the third version takes any number of arguments using the `*args` syntax.

Note that this is just an example of the basic syntax of method overloading in Python, and the implementation of the methods is left out for brevity. The actual implementation of the methods will depend on the specific requirements of your program.

Sample Code

```
class MyClass:
    def my_method(self, arg1):
        print("One argument: {}".format(arg1))
```

```
    def my_method(self, arg1, arg2):
        print("Two arguments: {}, {}".format(arg1, arg2))

# Create an object of MyClass
obj = MyClass()

# Call my_method() with one argument
obj.my_method("hello")

# Call my_method() with two arguments
obj.my_method("hello", 123)
```

In this example, the MyClass class defines two versions of the my_method() method, each taking a different number of arguments. When the my_method() method is called on an instance of the MyClass class, the correct version of the method is called based on the number of arguments passed to it.

Note that in Python, method overloading is not a built-in feature, and the above example is just a demonstration of how method overloading can be achieved in Python.

Method Overriding

Syntax

```
class ParentClass:
    def my_method(self):
        # implementation of the method

class ChildClass(ParentClass):
    def my_method(self):
        # implementation of the method in the child class
```

In this example, the ChildClass subclass inherits the my_method() method from its ParentClass superclass. However, it also provides a different implementation of the

method by defining it with the same name in ChildClass. When the my_method() method is called on an instance of the ChildClass, the implementation of the method in the child class will be used instead of the implementation in the parent class.

Note that to override a method in Python, the method in the subclass must have the same name and signature (i.e., the same parameters) as the method in the parent class.

Sample Code

```
class Animal:
    def speak(self):
        print("Animal is speaking.")
 class Dog(Animal):
    def speak(self):
        print("Dog is barking.")

# Create an object of Dog
my_dog = Dog()

# call speak() method on my_dog
my_dog.speak()
```

In this example, the `Animal` class has a `speak()` method that simply prints "Animal is speaking." The `Dog` class is a subclass of `Animal` and overrides the `speak()` method with its own implementation that prints "Dog is barking." When the `speak()` method is called on an instance of the `Dog` class, the overridden version of the method is called.

Note that method overriding is a widespread practice in object-oriented programming to provide a more specific implementation of a method in a subclass.

C#

Method Overloading

Syntax

```
public class MyClass
{
    public void MyMethod(string arg1, string arg2 = null, string arg3 =
null)
    {
        // First version of the method
    }

    public void MyMethod(string arg1, string arg2)
    {
        // Second version of the method
    }

    public void MyMethod(params string[] args)
    {
        // Third version of the method
    }
}
```

In this syntax, `MyClass` class defines three versions of the `my_method` method with the same name, but different parameter lists. The first version takes three arguments, the second version has two arguments, and the third version takes any number of arguments using the `params` keyword.

Sample Code

```
using System;

public class MyClass
{
    public void MyMethod(string arg1)
    {
        Console.WriteLine("One argument: " + arg1);
    }

    public void MyMethod(string arg1, object arg2)
    {
        Console.WriteLine("Two arguments: {0}, {1}", arg1, arg2);
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyClass obj = new MyClass();

        obj.MyMethod("hello");

        obj.MyMethod("hello", 123);
    }
}
```

Method Overriding

Syntax

```
public class ParentClass
{
```

```
    public void MyMethod()
    {
        // Implementation of the method
    }
}

public class ChildClass : ParentClass
{
    public new void MyMethod()
    {
        // Implementation of the method in the child class
    }
}
```

The given code snippet represents two classes: ParentClass and ChildClass.

ChildClass inherits from ParentClass. The ParentClass is defined with a method called MyMethod(), and the ChildClass inherits from ParentClass and provides its own implementation of the MyMethod() method.

Sample Code

```
using System;
public class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal is speaking.");
    }
}
public class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog is barking.");
    }
}
class Program
{
    static void Main(string[] args)
    {
        Dog myDog = new Dog();
        myDog.Speak();
    }
}
```

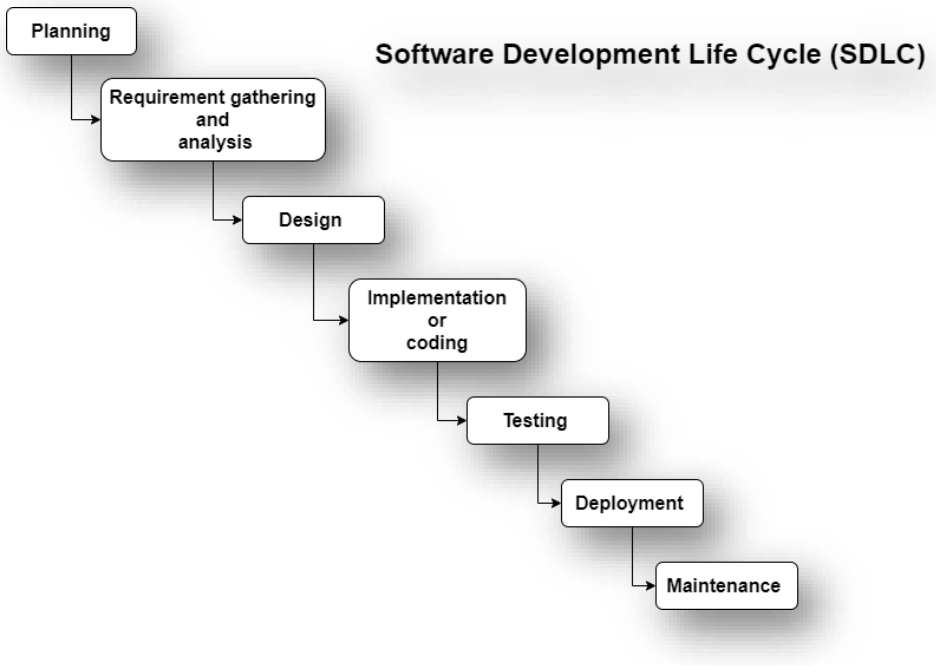
The given code snippet defines a class hierarchy with **Animal** as the base class and **Dog** as a derived class. The **Dog** class overrides the **speak()** method inherited from the **Animal** class .

In the **Main()** method, an object of **Dog** is created (**myDog**) and the **Speak()** method is called on it, which will output "Dog is barking."

SDLC Fundamentals

SDLC stands for Software Development Life Cycle. It is a systematic process that outlines the phases involved in developing software applications. The purpose of SDLC is to provide a structured approach to software development, ensuring that the software is designed, implemented, and evaluated efficiently and effectively.

SDLC encompasses activities such as requirements gathering, system design, coding, testing, deployment, and maintenance, guiding the development team through each stage of the software development process. SDLC helps manage resources, minimize risks, and deliver high-quality software solutions that meet the needs of users and stakeholders.



Waterfall Approach

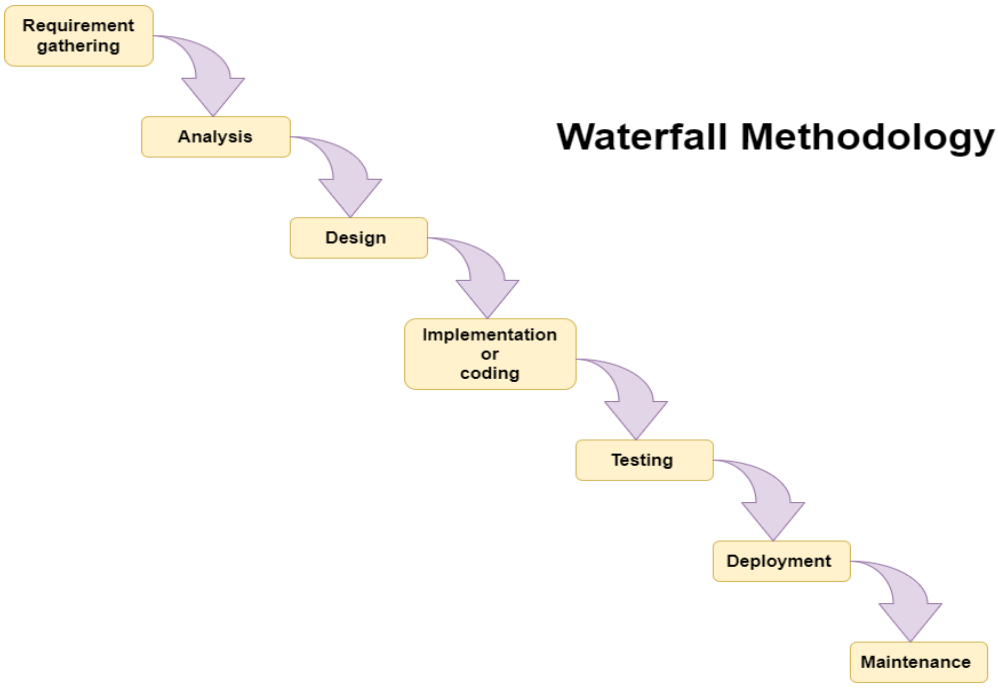
The waterfall model arranges all the phases sequentially so that each new phase depends on the outcome of the previous phase. Conceptually, the design flows from one phase down to the next, like that of a waterfall.

Pros

- Clear expectations on schedule, budget, and resourcing needs
- Extensive documentation helps ensure quality, reliability, and maintainability.
- Progress is easily measured.

Cons

- Inflexible and cumbersome
- Long pole from project starts to something tangible.
- Problems are discovered in user testing.
- Written documentation is never kept up to date, loses usefulness.
- Promotes gap between the business users and the development team.



Agile Approach

Agile methodology is a software development approach that emphasizes flexibility, collaboration, and customer satisfaction. It is based on the Agile Manifesto, a set of guiding principles for software development that prioritize delivering working software frequently, responding to change, and collaborating with customers and stakeholders.

The Agile methodology follows a cyclical process known as the Agile software development life cycle (SDLC). This process is iterative and incremental, meaning that the development team produces a working version of the software at the end of each cycle, known as a sprint.

Pros

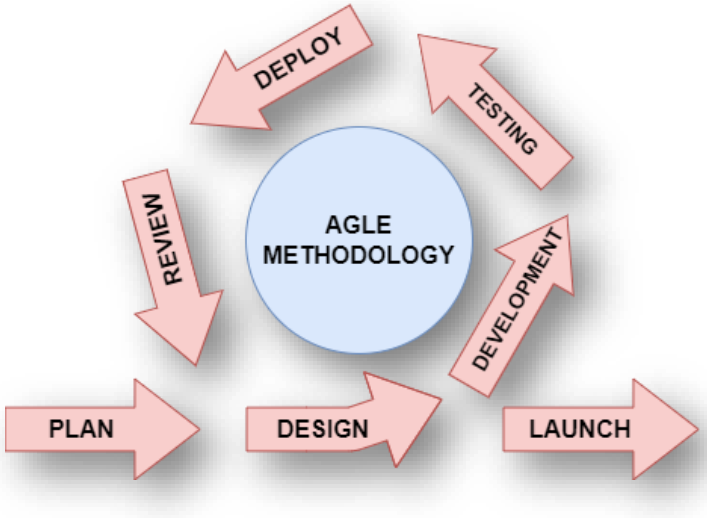
- Flexible and adaptable to changing requirements.
- Gets workable product in-front of the business quicker.
- Promotes collaboration between business and development teams.
- Daily feedback on progress and roadblocks, stops “spinning wheels.”

Cons

- Leads to scope creep due to no defined end date.
- Relies on commitment of all team members.
- A challenge to initially adopt and train in an organization.
- Changing or leaving team members can have a drastically negative effect.

Scrum Process

- Uses iterative approach called sprints.
- Flexible and adaptable, great for the unknown
- Values individuals and interactions over processes and tools
- Values working software over comprehensive documentation.
- Values customer collaboration over contract negotiation
- Values responding to change over following a plan.



Scrum Ceremonies

The project team is made of Product Owner, Scrum Dev Team, and Scrum

Master

Product Owner is responsible for the return on investment (ROI).

• Focuses on the requirements, the “what”

Scrum Dev Team is cross-functional.

• Collaborates to build the product each sprint, the “how”

Scrum Master is the team facilitator but has no power.

• Removes roadblocks, sets periods, and provides visibility.

Overall project team completes sprints. A short iteration from analysis, development, testing and deployment of a product (14-30 days long). It uses a subset of prioritized requirements that form a sprint backlog and requirements are derived from a product backlog.

Artifacts

Product Backlog

- List of features that the business wants, force ranked by the Product Owner (one #1)
- Anyone can add items, could be user stories or use cases
- Does not contain tasks

Sprint Backlog

- Committed features that will be completed within the current sprint
- Has a deadline to complete
- Tasks are tracked as
- Not started
- In Progress
- Completed

Agile Meetings

- Sprint Planning – Commit items to the sprint backlog
    - Daily Scrum – Daily, 15 minutes. Yesterday, today, roadblocks
    - Sprint Review – Demonstrate product, get feedback
    - Sprint Retrospective – Inspect last sprint. Lessons learned.
    - Product Backlog Refinement – Adjust, split, and determine dependencies of backlog items

Agile: When to Use

- The project is unpredictable and will have changing requirements.
- Using or creating leading edge technology

- Organization as an experienced Scrum Master
- Business has experienced resources that can dedicate time to the project.
- Pressure to produce a tangible product quickly.
- Little to no concerns on length of project or budget
- The development team does not have resource constraints.

SQL

Introduction to SQL

The programming language known as SQL, or Structured Query Language, is used to manage and work with relational databases. It offers a standardized method for creating, accessing, updating, and deleting data that is stored in a database, and it is utilized by many applications and sectors that deal with significant amounts of data.

SQL

A relational database's structured query language (SQL) is a computer language used to store and process data. In a relational database, data is stored in tabular form, with rows and columns denoting various data qualities and the connections between the values of those attributes.

The SQL engine decides how to interpret that specific instruction when we execute a SQL command on a relational database management system. The system will automatically pick the appropriate routine to carry out our request.

Advantages of SQL

Data retrieval: SQL provides an easy and efficient way to retrieve data and the conditions under which it should be retrieved.

Data manipulation: SQL provides a powerful set of commands for manipulating data in a database. Users can insert, update, and delete data using SQL commands.

Data aggregation: SQL provides functions for summarizing and aggregating data in a database.

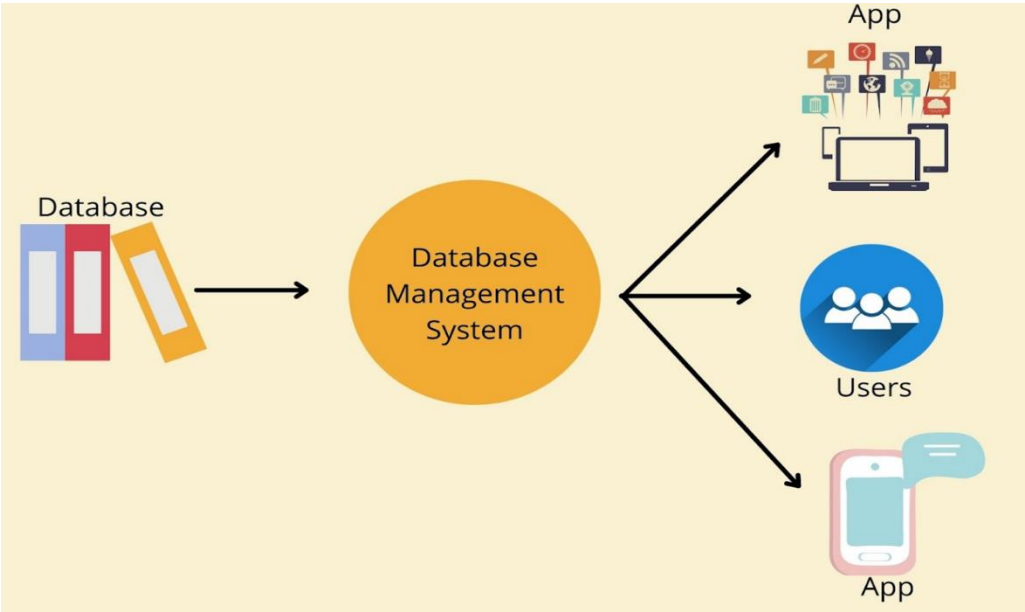
Data modeling: SQL provides a way to define the structure of a database using tables, columns, and relationships between tables.

Data security: SQL provides security features such as access controls and authentication to protect the data in a database from unauthorized access.

Data scalability: SQL is scalable and can manage substantial amounts of data and multiple users.

Database (DBMS)

A database management system organizes, stores, and manages the data of an organization and provides the users with easy access and control over the data. A database management system can manipulate data as and when required. This includes creating data, editing data, and updating it.



RDBMS

Relational Database Management System is referred to as RDBMS. It is a kind of database management system (DBMS) that uses tables with rows and columns to represent data and relationships between them to store data in a structured way.

Primary key

Each row in a table can be uniquely identified by using a primary key, a special kind of constraint. To uniquely identify each entry in a table, a column or set of columns is utilized.

The primary key makes sure that every row in the database has a distinct identity and can be referred to by other tables. By restricting the insertion of duplicate records into the table, it also enforces data integrity.

To create a primary key in SQL, you can use the PRIMARY KEY keyword in the CREATE TABLE statement, followed by the name of the column or columns that you want to use as the primary key.

Sample Query

```
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  email VARCHAR(50)
);
```

id	name	email
101	john	john@gmail.com
102	joe	joe@gmail.com
103	paul	paul@gmail.com

In this example, the id column is used as the primary key for the user's table. It ensures that each row has a unique identifier and can be referenced by other tables in the database.

Foreign key

A constraint called a foreign key is used to connect two tables together. It is a column or set of columns in one table that refer to the primary key in another table, linking the two tables together.

Maintaining referential integrity between the linked tables is the goal of a foreign key. It makes sure that the information in the primary key column(s) of one table matches the foreign key column(s) of another table.

To create a foreign key in SQL, you can use the FOREIGN KEY keyword in the CREATE TABLE statement, followed by the name of the column or columns that you want to use as the foreign key.

Sample Query

```
CREATE TABLE orders (
underbid INT PRIMARY KEY,
customer_id INT,
order_date DATE,
FOREIGN KEY (customer_id) REFERENCES
customers(customer_id)
);
```

In this example, the customer_id column in the orders table is a foreign key that references the customer_id column in the customer's table. This creates a link between the two tables based on the customer_id values, ensuring that each order in the orders table corresponds to a valid customer in the customer's table.

Tables

A table in SQL is a grouping of related data that is set up in rows and columns. A table is created using a CREATE TABLE statement, which specifies the table name, column names, and data types for each column.

Creating a table called employees with four columns: **id, name, salary, and department**:

Sample Query

```
      CREATE TABLE employees
      (
        id INT PRIMARY KEY,
        name VARCHAR (50),
        salary DECIMAL (10,2),
        department VARCHAR (50)
);
```

Output

id	name	salary	department
empty			

In this example, we have specified that the id column is the primary key for the table, which means that its values must be unique for each row in the table. We have also specified the data types for each column: INT for the id column, VARCHAR (50) for the name and department columns (which allows for up to 50 characters of text), and DECIMAL (10,2) for the salary column (which allows for a decimal value with up to 10 digits and 2 decimal places).

Once the table has been created, we can add data to it using an INSERT INTO statement, like this:

INSERT INTO employees (id, name, salary, department) VALUES
(1, 'John Smith', 50000.00, 'Sales'),

(2, 'Jane Doe', 60000.00, 'Marketing'),
(3, 'Bob Johnson', 70000.00, 'Finance');

This statement adds three rows to the employees table, each representing a different employee with a unique id, name, salary, and department.

To query data from the table, we use a SELECT statement. For example, to retrieve all data from the employees table, we would use this statement:

```
SELECT * FROM employees;
```

This statement returns a result set that includes all columns and rows from the employee's table. We can also use WHERE clauses to filter the results based on specific conditions, and ORDER BY clauses to sort the results by a specific column.

Normalization

In SQL, the term "normalization" describes the process of arranging a relational database design to reduce redundancy and enhance data integrity. It entails dividing a big table into smaller, more manageable tables and creating connections between them. For the database structure to be effective and free of data anomalies, the normalization procedure adheres to a set of guidelines known as normal forms.

Let us go through the normalization process step by step with examples:

Consider an initial table called "Employees" with the following structure:

Table: Employees

EmployeeID	EmployeeName	Department	Salary
1	John	Engineering	50000
2	Jane	Marketing	60000

3	Mike	Engineering	55000

First Normal Form (1NF): In 1NF, we ensure that each column contains atomic values, and there are no repeating groups. In the "Employees" table, we have a repeating group in the "Department" column because multiple employees can belong to the same department. To eliminate this repetition, we create a separate table for departments.

Table: Employees

EmployeeID	EmployeeName	Salary
1	John	50000
2	Jane	60000
3	Mike	55000

Table: Departments

DepartmentID	Department
1	Engineering
2	Marketing

Now, we have two separate tables, "Employees" and "Departments," where the data is no longer duplicated, and each table represents a distinct entity.

Second Normal Form (2NF): In 2NF, we ensure that each non-key column is functionally dependent on the entire primary key. In our current structure, both "EmployeeName" and "Salary" are functionally dependent on the "EmployeeID," which is the primary key. Therefore, our structure is already in 2NF.

Third Normal Form (3NF): In 3NF, we ensure that no non-key column is transitively dependent on the primary key. In our current structure, "Department" is transitively

dependent on the "EmployeeID" through the "Employees" table. To remove this transitive dependency, we create a separate table for employee departments.

Table: Employees

EmployeeID	EmployeeName	Salary
1	John	50000
2	Jane	60000
3	Mike	55000

Table: Departments

DepartmentID	Department
1	Engineering
2	Marketing

Table: EmployeeDepartments

EmployeeID	DepartmentID
1	1
2	2
3	1

In the updated structure, we have a new table called "EmployeeDepartments," which contains the "EmployeeID" and "DepartmentID" columns. This table establishes a many-to-many relationship between employees and departments.

By following the normalization process, we have decomposed the initial table into separate tables, eliminating data redundancy and ensuring data integrity. This normalized structure allows for efficient storage, retrieval, and manipulation of data in a relational database.

Datatypes

SQL offers several distinct data types to store several types of data in a database. Depending on the database management system (DBMS) being used, there may be subtle variations in the precise data types that are offered. The following are some of the data types used in SQL frequently.

Numeric Data Types

- INTEGER: Used for storing whole numbers (e.g., 1, 100, -5).
- FLOAT/REAL/DOUBLE: Used for storing decimal numbers with varying precision and scale.
- DECIMAL/NUMERIC: Used for storing decimal numbers with fixed precision and scale.

Character Data Types

- CHAR: Fixed-length character string (e.g., 'Hello', 'OpenAI').
- VARCHAR: Variable-length character string (e.g., 'John', 'Database').

Date and Time Data Types

- DATE: Used for storing dates (e.g., '2023-05-18').
- TIME: Used for storing times (e.g., '14:30:00').
- DATETIME/TIMESTAMP: Used for storing both date and time information (e.g., '2023-05-18 14:30:00').

Boolean Data Type

- BOOLEAN: Used for storing true/false values.

Binary Data Types

- BLOB: Used for storing large binary objects, such as images or files.

- BINARY/VARBINARY: Used for storing fixed-length or variable-length binary data.

Enumerated Data Types

- ENUM: Used for defining a list of predefined values that a column can take.

Miscellaneous Data Types

- TEXT: Used for storing enormous amounts of text data.
- JSON: Used for storing JSON (JavaScript Object Notation) data.

Constraints

Constraints in SQL are used to specify requirements that the data in a database table must meet. They uphold consistency, enforce business rules, and guarantee data integrity. The following are the few often used SQL constraints.

NOT NULL: The NOT NULL constraint ensures that a column cannot have a NULL (empty) value. It enforces the requirement that the column must have a value. For example:

```
CREATE TABLE Employees (
      EmployeeID INT,
      EmployeeName VARCHAR(50) NOT NULL,
      ...
      );
```

UNIQUE: The UNIQUE constraint ensures that each value in a column or a group of columns is unique across the table. It prevents duplicate values in the specified column(s). For example:

```
CREATE TABLE Employees (
      EmployeeID INT UNIQUE,
```

```
      ...
      );
```

CHECK: The CHECK constraint defines a condition that must be true for the values in a column. It allows you to define custom rules or conditions for data in the table. For example:

```
CREATE TABLE Employees (
      EmployeeID INT,
      Age INT,
      ...
      CHECK (Age >= 18)
      );
```

DEFAULT: The DEFAULT constraint sets a default value for a column when no value is specified during an insert operation. It assigns a predefined default value to the column if no explicit value is provided. For example:

```
      CREATE TABLE Employees (
      EmployeeID INT,
      HireDate DATE DEFAULT CURRENT_DATE,
      ...
      );
```

DDL

DDL stands for Data Definition Language, which is a set of SQL commands used to define and manage the structure of a database. The four main categories of DDL commands are:

CREATE: Used to create database objects, such as tables, indexes, views, and procedures.

Sample Query

```
-- Create a new table

      CREATE TABLE customers (
        customer_id INT PRIMARY KEY,
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        email VARCHAR(50)
      );
```

ALTER: Used to modify the structure of an existing database object, such as adding or removing columns from a table.

Sample Query

```
-- Alter an existing table

      ALTER TABLE customers
      ADD COLUMN phone_number VARCHAR(20);
```

DROP: Used to delete a database object, such as dropping a table or view.

Sample Query

```
-- Drop a table

      DROP TABLE customers;
```

TRUNCATE: Used to remove all data from a table, while keeping its structure intact.

Sample Query

```
-- Truncate a table

      TRUNCATE TABLE orders;
```

DML

DML stands for Data Manipulation Language, which is a set of SQL statements used to manipulate the data stored in a database.

SELECT: Used to retrieve data from one or more tables in a database.

Sample Query

```
-- Select data from a table

      SELECT * FROM customers;
```

INSERT: Used to insert new rows of data into a table.

Sample Query

```
-- Insert a new row into a table

INSERT INTO customers (customer_id, first_name, last_name, email)
      VALUES (1, 'John', 'Doe', 'johndoe@example.com');
```

UPDATE: Used to modify existing data in a table.

Sample Query

```
-- Update existing data in a table

      UPDATE customers
      SET email = 'newemail@example.com'
      WHERE customer_id = 1;
```

DELETE: Used to remove rows of data from a table.

Sample Query

```
-- Delete a row from a table

      DELETE FROM customers
      WHERE customer_id = 1;
```

DQL

A relational database's data can be retrieved or queried using DQL commands. With their help, you may define the standards and parameters under which one or more database tables should be searched for data records or subsets of data. A few essential SQL DQL commands are shown below:

SELECT: Data from one or more tables can be retrieved using the SELECT statement. It enables you to choose the columns you want to obtain and can incorporate aggregate functions for data analysis. For example,

```
SELECT column1, column2 FROM table_name;
```

FROM: The table or tables from which data is to be obtained are specified in the FROM clause. It identifies the SELECT statement's source tables. For example,

```
SELECT column1, column2 FROM table1, table2;
```

WHERE: The criteria that must be satisfied for records to be chosen are stated in the WHERE clause. The result set is filtered according to the entered criteria. For example,

```
SELECT column1, column2 FROM table_name WHERE condition;
```

JOIN: Utilising a linked column between two or more tables, the JOIN keyword is used to merge records from those tables. It is used to create connections between tables and simultaneously obtain data from several tables. For example,

```
SELECT column1, column2 FROM table1 JOIN table2 ON table1.column =
table2.column;
```

GROUP BY: The result set is grouped using the GROUP BY clause according to the supplied columns. To do computations on aggregated data, it is frequently used in conjunction with aggregate functions like SUM, COUNT, AVG, etc. For example,

```
SELECT column1, SUM(column2) FROM table_name GROUP BY column1;
```

HAVING: The HAVING clause specifies conditions for filtering the grouped data. It is used to apply conditions to the result set after the GROUP BY clause has been applied. For example,

```
SELECT column1, SUM(column2) FROM table_name GROUP BY column1 HAVING
condition;
```

ORDER BY: The ORDER BY clause is used to sort the result set based on specified columns. It arranges the data in either ascending (ASC) or descending (DESC) order. For example,

```
SELECT column1, column2 FROM table_name ORDER BY column1 ASC;
```

LIMIT: The LIMIT clause is used to specify the maximum number of rows to be returned in the result set. It allows you to control the size of the result set. For example,

```
SELECT column1, column2 FROM table_name LIMIT 10;
```

These DQL commands form the foundation of querying data in SQL. They provide the flexibility to retrieve, filter, join, group, sort, and limit the data according to your specific requirements.

Inbuilt basic function

Several built-in functions in SQL are available for use in performing operations on database-stored data. To produce valuable insights, these functions can alter, aggregate, and change data. The following are the frequently used SQL data functions:

Date and Time Functions

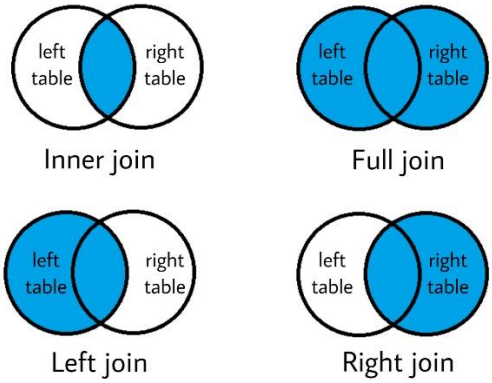
- NOW(): Returns the current date and time.
- DATE(): Extracts the date portion from a date/time value.
- YEAR(), MONTH(), DAY(): Extracts the year, month, or day from a date.
- DATE_FORMAT(): Formats a date or time value as a string.

String Functions

- CONCAT(): Concatenates two or more strings together.
- LENGTH(): Returns the length of a string.
- UPPER(): Converts a string to uppercase.
- LOWER(): Converts a string to lowercase.
- SUBSTRING(): Extracts a portion of a string.
- REPLACE(): Replaces occurrences of a substring in a string.

Joins

When combining data from two or more tables based on a shared column, a join is performed. Data from many tables can be accessed by joins as if they were a single table.



There are several types of joins in SQL, including:

INNER JOIN: Returns only the matching rows between two tables.

Sample Query

```
-- Inner join
      SELECT orders.order_id, customers.first_name, customers.last_name
      FROM orders
      INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

LEFT JOIN (or LEFT OUTER JOIN): Returns all rows from the left table and the matching rows from the right table. If there is no matching row in the right table, the result will contain NULL values.

Sample Query

```
-- Left join
      SELECT customers.first_name, customers.last_name, orders.order_id
      FROM customers
```

```
      LEFT JOIN orders ON customers.customer_id = orders.customer_id;
```

RIGHT JOIN (or RIGHT OUTER JOIN): Returns all rows from the right table and the matching rows from the left table. If there is no matching row in the left table, the result will contain NULL values.

Sample Query

```
-- Right join
      SELECT customers.first_name, customers.last_name, orders.order_id
      FROM customers
      RIGHT JOIN orders ON customers.customer_id = orders.customer_id;
```

FULL JOIN (or FULL OUTER JOIN): Returns all rows from both tables, with NULL values in the columns where there is no match.

Sample Query

```
-- Full join
      SELECT customers.first_name, customers.last_name, orders.order_id
      FROM customers
      FULL OUTER JOIN orders ON customers.customer_id = orders.customer_id;
```

Algorithm

Programming Complexity

Space and time complexity are fundamental concepts in programming that are used to analyze the efficiency of algorithms. They help us understand how the runtime of an algorithm or the amount of memory it requires scales with the size of the input.

Time Complexity

Time complexity is a measure of how the runtime of an algorithm grows as the input size increases. It quantifies the number of operations an algorithm performs relative to the size of the input. Time complexity is usually expressed using Big O notation.

Here are some common time complexity classes from best to worst:

- O(1) - Constant time complexity. The algorithm takes the same amount of time regardless of the input size.

- O(log n) - Logarithmic time complexity. The algorithm's runtime grows logarithmically with the input size.

- O(n) - Linear time complexity. The algorithm's runtime scales linearly with the input size.

- O(n log n) - Linearities time complexity. The algorithm's runtime grows in proportion to the input size multiplied by the logarithm of the input size.

- O(n^2) - Quadratic time complexity. The algorithm's runtime grows quadratically with the input size.

- O(2^n) - Exponential time complexity. The algorithm's runtime grows exponentially with the input size.

Time complexity analysis provides an upper bound on how an algorithm performs as the input size increases. It helps us compare and select the most efficient algorithm for a given problem.

Space Complexity

Space complexity refers to the amount of memory or storage an algorithm requires to solve a problem. It measures how the memory usage of an algorithm scales with the input size.

Like time complexity, space complexity is also expressed using Big O notation. It represents the worst-case scenario for memory usage.

Here are some common space complexity classes from best to worst:

- O(1) - Constant space complexity. The algorithm requires a fixed amount of memory regardless of the input size.

- O(n) - Linear space complexity. The algorithm's memory usage scales linearly with the input size.

- O(n^2) - Quadratic space complexity. The algorithm's memory usage grows quadratically with the input size.

- O(2^n) - Exponential space complexity. The algorithm's memory usage grows exponentially with the input size.

Space complexity analysis helps us understand how much memory an algorithm needs to execute and can guide us in optimizing memory usage or selecting more memory-efficient approaches.

It is important to note that time and space complexity analysis considers the growth rate of an algorithm as the input size increases. They provide a high-level understanding of algorithmic efficiency but may not capture all implementation-specific details or constant factors.

Linear Search

Linear search, also called sequential search, is a simple search algorithm used to find target value in a list or array of values. Check each element of the list in order until a match is found, or the end of the list is reached.

The algorithm starts at the top of the list, examining each element in turn and comparing it to the target value. If a match is found, the index of that item is returned. If the end of the list is reached without finding a match, the algorithm returns an exceptional value (usually -1) indicating that the target value is not in the list.

Here is the step-by-step process of the linear search algorithm:

- Start at the first element of the list.
- Compare the target value with the current element.
- If the target value matches the current element, return the index of the current element.
- If the target value does not match the current element, move to the next element.
- If there are no more elements to check, return -1 to indicate that the target value is not in the list.
- Linear search has a time complexity of O(n), where n is the number of elements in the list. This means that as the size of the list increases, the time required to search for a target value in the list also increases linearly. While linear search is not the most efficient algorithm for large lists, it is still useful for small lists or for cases where the list is unsorted.

Sample Code

```
public class LinearSearchExample {
    public static int linearSearch(int[] arr, int target) {
        for (int i = 0; i < arr.length; i++) {
            if (arr[i] == target) {
                return i;
            }
        }
        return -1; // target value not found
    }
    public static void main(String[] args) {
        int[] arr = {3, 9, 2, 5, 8, 4};
        int target = 5;
        int index = linearSearch(arr, target);
        if (index != -1) {
            System.out.println("Target value found at index " + index);
        } else {
            System.out.println("Target value not found");
        }
    }
}

//Output - Target value found at index 3
```

Explanation:

In this example, we define a method called linearSearch that takes an integer array arr and a target integer value target. The method iterates through each element of the array using a for loop and checks whether the current element is equal to the target value using an if

statement. If a match is found, the index of the current element is returned. If the end of the array is reached without finding a match, the method returns -1 to indicate that the target value is not in the array. In the main method, we create an integer array arr with values {3, 9, 2, 5, 8, 4} and a target value 5. We then call the linearSearch method with arr and target as arguments and store the return value in an integer variable index. If the index is not -1, we print a message indicating that the target value was found at the index returned by the linearSearch method. Otherwise, we print a message indicating that the target value was not found. In this example, the output is "Target value found at index 3" because the target value 5 is found at index 3 of the array.

Binary Search

Binary search is a widely used algorithm for searching elements in a sorted array or list. It efficiently finds the position of a target value within the collection by repeatedly dividing the search space in half. This algorithm is based on the principle of "divide and conquer," making it highly efficient for large datasets.

Here's how binary search works:

1. Initialization: Begin by identifying the search space, which is typically the entire sorted array. Set the low and high pointers to the first and last elements of the array, respectively.

2. Find the middle: Calculate the middle index of the current search space by taking the average of the low and high pointers: `mid = (low + high) / 2`. If the search space has an odd number of elements, round down to the nearest integer.

3. Compare with the target: Retrieve the element located at the middle index and compare it with the target value you want to find.

- If the middle element is equal to the target, the search is successful, and you can return the middle index.

- If the middle element is greater than the target, it means the target can only be present in the left half of the search space. Adjust the high pointer to `mid - 1` and go to step 2.

- If the middle element is less than the target, it means the target can only be present in the right half of the search space. Adjust the low pointer to `mid + 1` and go to step 2.

4. Repeat: Continue steps 2 and 3, halving the search space in each iteration, until the target is found, or the search space is empty (low pointer becomes greater than the high pointer). If the target is not found after the binary search concludes, it means the target value is not present in the array.

Binary search has a time complexity of O(log n), where n is the number of elements in the array. This logarithmic time complexity makes binary search highly efficient, especially for large datasets, as it significantly reduces the number of comparisons required compared to

linear search algorithms. However, binary search requires the data to be sorted beforehand, which can add additional overhead if the sorting is not already performed.

Sample Code

```
public class BinarySearchExample {
    public static int binarySearch(int[] arr, int target) {
        int low = 0;
        int high = arr.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;

            if (arr[mid] == target) {
                return mid;
            } else if (arr[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }

        return -1; // Return -1 if target is not found
    }

    public static void main(String[] args) {
        int[] array = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
        int target = 23;

        int result = binarySearch(array, target);

        if (result == -1) {
            System.out.println("Element not found in the array.");
        } else {
            System.out.println("Element found at index: " + result);
        }
    }
```

```
}

//Output : Element found at index: 5
```

Explanation:

In this example, we have an array containing sorted elements, and we want to find the target value 23 using binary search. The binarySearch method takes the array and the target value as parameters and returns the index where the target value is found. If the target value is not found, it returns to -1. Inside the main method, we create an array array and set the target value to 23. We then call the binarySearch method with the array and target value and store the result in the result variable. The binary search algorithm is executed within the binarySearch method. It initializes the low pointer to the beginning of the array (0) and the high pointer to the end of the array (arr.length - 1). The algorithm enters a while loop that continues until the low pointer is less than or equal to the high pointer. In each iteration, it calculates the middle index mid as the average of the low and high pointers. If the middle element (arr[mid]) is equal to the target value, the method returns the middle index mid. If the middle element is less than the target value, it means the target can only be present in the right half of the search space. In this case, the low pointer is adjusted to mid + 1. If the middle element is greater than the target value, it means the target can only be present in the left half of the search space. In this case, the high pointer is adjusted to mid - 1. The process continues, dividing the search space in half with each iteration, until the target value is found, or the search space is empty. In this example, the target value 23 is found at index 5, and the program prints "Element found at index: 5" as the output. Note: Binary search assumes that the input array is sorted in ascending order. If the array is not sorted, binary search may not produce correct results.

Bubble Sort

Bubble sort is a simple comparison-based sorting algorithm. It works by repeatedly swapping adjacent elements if they are in the wrong order until the complete list is sorted. Bubble sort gets its name because smaller elements "bubble" to the top of the list during each iteration.

Here is how the bubble sort algorithm works:

1. Start with an unsorted list of elements.

2. Compare the first element with the second element. If the first element is greater than the second element, swap them.

3. Move to the next pair of adjacent elements and repeat the comparison and swapping process.

4. Continue this process for each pair of adjacent elements until you reach the end of the list. At this point, the largest element is guaranteed to be at the end of the list.

5. Repeat steps 2-4 for the remaining unsorted portion of the list, excluding the last element since it is already in its correct position.

6. Repeat steps 2-5 until the complete list is sorted.

Bubble sort has a worst-case and average time complexity of O(n^2), where n is the number of elements in the list. This means that as the size of the list increases, the time it takes to sort the list grows quadratically. In practice, bubble sort is not efficient for large datasets, but it can be useful for small lists or educational purposes due to its simplicity.

Sample Code

```
public class BubbleSort {
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
```

```
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 12, 1};
        System.out.println("Original array: " + Arrays.toString(arr));
        bubbleSort(arr);
        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
}

//Output:
Original array: [5, 2, 8, 12, 1]
Sorted array: [1, 2, 5, 8, 12]
```

Explanation

1. The `bubbleSort` method takes an integer array as a parameter and sorts it using the bubble sort algorithm.

2. It starts with initializing the length of the array, `n`, which will be used in the loops.

3. The outer loop iterates `n - 1` times because after each iteration, the largest element gets placed at the end of the array, so the inner loop can be one element shorter.

4. The inner loop iterates from the beginning of the array up to `n - i - 1`. The `n - i - 1` ensures that in each pass, the last `i` elements, which are already sorted, are not considered.

5. Inside the inner loop, each pair of adjacent elements is compared, and if they are in the wrong order, they are swapped.

6. Once the outer and inner loops complete, the array is sorted.

7. In the `main` method, we create an array `[5, 2, 8, 12, 1]` and print it as the original array.

8. We then call the `bubbleSort` method passing the array to sort it.

9. Finally, we print the sorted array.

The output shows the original array and the sorted array using the bubble sort algorithm. In this case, the original array `[5, 2, 8, 12, 1]` is sorted into `[1, 2, 5, 8, 12]`.

Insertion Sort

Insertion sort is a simple comparison-based sorting algorithm that builds the final sorted array one element at a time. It works by dividing the array into a sorted portion and an unsorted portion. Initially, the sorted portion consists of only the first element of the array, and the unsorted portion contains the remaining elements. The algorithm iterates through the unsorted portion, picking one element at a time and inserting it into its correct position in the sorted portion, thereby expanding the sorted portion by one element in each iteration. This process continues until the entire array is sorted.

Here is a step-by-step explanation of the insertion sort algorithm:

1. Start with the second element (index 1) of the array.

2. Compare the second element with the previous element (index 0). If the second element is smaller, swap them.

3. Now, consider the third element (index 2). Compare it with the previous elements (index 1 and index 0) and swap it with the correct position in the sorted portion.

4. Repeat this process for each subsequent element in the unsorted portion, moving it to its correct position in the sorted portion.

5. Continue these steps until the entire array is sorted.

Sample Code

```
public class InsertionSort {
    public static void main(String[] args) {
        int[] array = {5, 2, 4, 6, 1, 3};
        System.out.println("Original Array: ");
        printArray(array);

        insertionSort(array);
```

```
        System.out.println("Sorted Array: ");
        printArray(array);
    }

    public static void insertionSort(int[] array) {
        int n = array.length;
        for (int i = 1; i < n; i++) {
            int key = array[i];
            int j = i - 1;

            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j--;
            }

            array[j + 1] = key;
        }
    }

    public static void printArray(int[] array) {
        for (int i : array) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
}

//Output:
Original Array:
5 2 4 6 1 3
Sorted Array:
1 2 3 4 5 6
```

In this example, the original array [5, 2, 4, 6, 1, 3] is sorted in ascending order using the Insertion Sort algorithm. The sorted array [1, 2, 3, 4, 5, 6] is printed as the output.

Explanation

1. The `main` method initializes an array with values [5, 2, 4, 6, 1, 3]. It then prints the original array using the `printArray` method.

2. The `insertionSort` method takes the array as input and performs the insertion sort algorithm on it.

3. The outer loop iterates from the second element (index 1) to the end of the array.

4. Within the loop, we assign the current element (array[i]) to a variable called `key`. This element will be compared and inserted into its correct position in the sorted portion.

5. The inner while loop checks if the element at index j is greater than the key. If it is, we shift the element to the right (array[j + 1] = array[j]) to make space for the key.

6. We decrement j and repeat the comparison until either j becomes less than 0 or the element at index j is not greater than the key.

7. Finally, we insert the key into its correct position (array[j + 1] = key).

8. After the sorting is complete, the `main` method prints the sorted array using the `printArray` method.

Quick Sort

Quick sort is a divide-and-conquer algorithm that works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. Here is an overview of the quick sort algorithm:

- Choose a pivot element from the array (usually the first or last element).

- Partition the array in such a way that all elements less than the pivot are moved to the left, and all elements greater than the pivot are moved to the right.

- Recursively apply the above two steps to the sub-arrays created on the left and right of the pivot.

- The base case of the recursion is when the sub-array has zero or one element, in which case it is already sorted.

Quick sort has an average-case time complexity of O(n log n), where n is the number of elements to be sorted. However, in the worst case, the time complexity can be O(n^2) if the pivot selection is not optimal (e.g., always choosing the first or last element). Various techniques, such as randomized pivot selection or using median-of-three, can be employed to mitigate the worst-case scenario.

Sample Code

```
public class QuickSort {

    public static void quickSort(int[] array) {
        if (array == null || array.length == 0) {
            return; // Base case: empty array or single element array
        }
        sort(array, 0, array.length - 1);
    }

    private static void sort(int[] array, int low, int high) {
        int i = low;
        int j = high;
        int pivot = array[low + (high - low) / 2]; // Choose the pivot element

        // Partitioning
        while (i <= j) {
            while (array[i] < pivot) {
                i++;
            }
            while (array[j] > pivot) {
                j--;
            }
            if (i <= j) {
                // Swap elements at indices i and j
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
                i++;
                j--;
            }
        }

        // Recursively sort the sub-arrays
        if (low < j) {
            sort(array, low, j);
```

```
        }
        if (i < high) {
            sort(array, i, high);
        }
    }

    public static void main(String[] args) {
        int[] array = {8, 2, 4, 1, 7, 6, 3, 5};
        System.out.println("Original array: " + Arrays.toString(array));
        quickSort(array);
        System.out.println("Sorted array: " + Arrays.toString(array));
    }
}

//Output:
Original array: [8, 2, 4, 1, 7, 6, 3, 5]
Sorted array: [1, 2, 3, 4, 5, 6, 7, 8]
```

Explanation

1. The `quickSort` method serves as the entry point for the quick sort algorithm. It checks if the array is null or empty and then calls the `sort` method to perform the actual sorting.

2. The `sort` method takes the array, the lowest index (`low`), and the highest index (`high`) as parameters. It chooses a pivot element from the array (in this case, the middle element) and partitions the array such that all elements less than the pivot are on the left side, and all elements greater than the pivot are on the right side.

3. The partitioning process is performed using two indices, `i` and `j`, which start at the lowest and highest indices of the current sub-array, respectively. The elements are compared with the pivot, and if they are in the wrong order, they are swapped. The indices are incremented and decremented until they meet.

4. After partitioning, the method recursively calls itself to sort the sub-arrays created on the left and right sides of the pivot. This continues until the base case is reached (sub-array with zero or one element).

5. In the `main` method, we create an example array and print the original array. After calling the `quickSort` method, we print the sorted array.

In the provided example, the original array `[8, 2, 4, 1, 7, 6, 3, 5]` is sorted in ascending order using the Quick Sort algorithm. The output shows the sorted array `[1, 2, 3, 4, 5, 6, 7, 8]`.

Merge Sort

Merge sort is also a divide-and-conquer algorithm that works by dividing the array into two halves, sorting each half recursively, and then merging the sorted halves to produce a sorted output. Here is an overview of the merge sort algorithm:

- Divide the unsorted array into two halves.

- Recursively sort each half by applying merge sort.

- Merge the two sorted halves by comparing the elements and arranging them in the desired order.

- The base case of the recursion is when the sub-array has zero or one element, in which case it is already sorted.

Merge sort guarantees a time complexity of O(n log n) in all cases, which makes it more predictable than quick sort. However, it requires additional space to store the temporary merged sub-arrays, which can be a drawback for large arrays or memory-constrained environments.

Sample Code

```
public class MergeSort {

    public static void mergeSort(int[] arr) {
        if (arr.length < 2) {
            return; // Base case: array is already sorted
        }

        int mid = arr.length / 2;
        int[] left = new int[mid];
        int[] right = new int[arr.length - mid];

        // Split the array into two halves
```

```
        for (int i = 0; i < mid; i++) {
            left[i] = arr[i];
        }
        for (int i = mid; i < arr.length; i++) {
            right[i - mid] = arr[i];
        }

        // Recursively sort the two halves
        mergeSort(left);
        mergeSort(right);

        // Merge the sorted halves
        merge(arr, left, right);
    }

    private static void merge(int[] arr, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;

        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                arr[k++] = left[i++];
            } else {
                arr[k++] = right[j++];
            }
        }

        while (i < left.length) {
            arr[k++] = left[i++];
        }

        while (j < right.length) {
            arr[k++] = right[j++];
        }
    }
    public static void main(String[] args) {
        int[] arr = {9, 4, 2, 7, 1, 5, 8, 3, 6};

        System.out.println("Original array: ");
```

```
        printArray(arr);

        mergeSort(arr);
        System.out.println("Sorted array: ");
        printArray(arr);
    }
    private static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}

Output:
Original array:
9 4 2 7 1
```

Explanation

1. The `mergeSort` method is the entry point of the merge sort algorithm. It takes an array as input and recursively sorts it. If the length of the array is less than 2, it means the array is already sorted, so it returns.
2. Otherwise, it determines the middle index and creates two new arrays, `left` and `right`, to store the elements of the two halves of the original array. It then uses a loop to populate the `left` array with elements from the start of the original array and the `right` array with elements from the middle index to the end of the original array.
3. Next, the method recursively calls itself to sort the `left` and `right` arrays.
4. Finally, the `merge` method is called to merge the sorted `left` and `right` arrays back into the original array. It uses three pointers, `i`, `j`, and `k`, to iterate through the `left`, `right`, and `arr` arrays, respectively. The method compares the elements

at the corresponding pointers and assigns the smaller element to the `arr` array, incrementing the respective pointers. It continues this process until one of the arrays is fully traversed. Then, it copies any remaining elements from the non-empty array to the `arr` array.

5. In the `main` method, we initialize an array, `arr`, with some random elements. We print the original array, call `mergeSort` to sort the array, and then print the sorted array.