Technical –

|  |  |
| --- | --- |
| C# |  |
| Dapper |  |
| ReactJs |  |
| SQL server |  |
| Redux |  |
| Design |  |
| SOLID principal |  |
| Linq |  |
| Azure |  |
| .Net fundamentals |  |
| Sorting and algorithms |  |
| OOps Concept |  |
| Web api / rest api |  |

1. OOps
2. authentication
3. API hitting
4. authorization
5. dependency injection
6. delegate
7. Indexer
8. event looping
9. event bubling
10. higher order function
11. hoising
12. 37,38,45
13. <https://medium.com/@vivekmadurai/different-ways-to-authenticate-a-web-application-e8f3875c254a>

1. 1. ref & out

3. Indexers

4. virtual abstract method   --method overriding

5. custom exception

6.delegates

7. Queue

8. List

9. Tree

10. Async await

11. Threading

React api callng....fetch etc

1. Adal
2. Higher order component
3. Webpack

BFS & DFS

how to implement stack using queue & vice versa

<https://hackr.io/blog/c-sharp-interview-questions>

1. 16,,29,30,33,34,35,36,38,40,42 54,55,65,69,71,74,81,82,90,114,127,128,231-332

controlled & uncontrolled component

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/jagged-arrays

https://www.geeksforgeeks.org/c-sharp-jagged-arrays/

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/null-coalescing-operator

https://www.tutorialspoint.com/csharp/csharp\_interview\_questions.htm

SQL –

**1.   Indexes:**

* Indexes are **special lookup tables** that is use to speed up data retrieval. Simply put, an index is a pointer to data in a table.

Eg: An index in a database is very similar to an index in the back of a book

* An index helps to speed up **SELECT** queries and **WHERE** clauses, but it slows down data input, with the **UPDATE** and the **INSERT** statements.
* Indexes can be created or dropped with no effect on the data.
* Indexes can also be unique, like the **UNIQUE** constraint, in that the index prevents duplicate entries in the column or combination of columns on which there is an index.
* Whether to create a single-column index or a composite index, take into consideration the column(s) that you may use very frequently in a query's WHERE clause as filter conditions. Should there be only one column used, a single-column index should be the choice. Should there be two or more columns that are frequently used in the WHERE clause as filters, the composite index would be the best choice.
* Clustered indexes are automatically created for primary key constraints and unique constraints.
* Clustered index:

Clustered indexes sort and store the data rows in the table or view based on their key values. There can be only one clustered index per table, because the data rows themselves can be stored in only one order.

* Nonclustered index:

Nonclustered indexes have a structure separate from the data rows. A nonclustered index contains the nonclustered index key values and row locator.

* Drop index:
  + Drop Index index\_name
* **When should indexes be created:**
  + A column contains a wide range of values.
  + A column does not contain a large number of null values.
  + One or more columns are frequently used together in a where clause or a join condition.
* **When should indexes be avoided:**
  + The table is small
  + The columns are not often used as a condition in the query
  + The column is updated frequently

1. **What are the differences between local and global temporary tables?**

* Local temporary tables are visible to the user who created the connection, and are deleted when the connection is closed.

CREATE TABLE #<tablename>

* Global temporary tables are visible to all users, and are deleted when the connection that created it is closed.

CREATE TABLE ##<tablename>

1. **What’s the difference between a primary key and a unique key?**

* The primary key cannot have null value while unique key can have one NULL value

1. **What is Mirroring?**

 It is designed to maintain a hot standby server which is consistent with the primary server in terms of a transaction. Transaction Log records are sent directly from the principal server to a secondary server which keeps a secondary server up to date with the principal server.

1. **What is Log Shipping?**

Log shipping is nothing but the automation of backup and restores the database from one server to another standalone standby server. This is one of the disaster recovery solutions. If one server fails for some reason we will have the same data available on the standby server.

1. **What is an execution plan?**

An execution plan is a graphical or textual way of showing how the SQL server breaks down a query to get the required result. It helps a user to determine why queries are taking more time to execute and based on the investigation user can update their queries for the maximum result.

Query Analyzer has an option, called “Show Execution Plan” (located on the Query drop-down menu). If this option is turned on, it will display a query execution plan in a separate window when the query is run again.

1. Stored procedure:

* Stored proc is group of statements that are stored together in a [database](https://www.simplilearn.com/what-is-database-management-article). We don't have to write same query again and again.
* The server only passes the procedure name instead of the whole query, reducing network traffic.
* Upon the first use, a plan for the stored procedure is created and stored in the buffer pool for quick execution for the next time. It cache the result, so when we call SP next time it directly pulls result and display. Hence improves performance

1. **What is recursive stored procedure?**

SQL Server supports recursive stored procedure which calls by itself. Recursive stored procedure can be defined as a method of problem solving wherein the solution is arrived repetitively. It can nest up to 32 levels.

CREATE PROCEDURE [dbo].[Fact]   
(

EXEC Fact @In, @Out OUTPUT - Same stored procedure has been called again(Recursively)

SELECT @RetVal = @Number \* @Out   
END   
ELSE   
BEGIN   
SELECT @RetVal = 1   
END   
RETURN   
GO

1. **What is sub query and its properties?**

A sub-query is a query which can be nested inside a main query like Select, Update, Insert or Delete statements. This can be used when expression is allowed. Properties of sub query can be defined as

A sub query should not have order by clause

A sub query should be placed in the right hand side of the comparison operator of the main query

A sub query should be enclosed in parenthesis because it needs to be executed first before the main query

More than one sub query can be included

1. Difference between Stored proc & function

Stored Procedures:

1. Stored Procedures are pre-compiled objects which are compiled for the first time and its compiled format is saved, which executes (compiled code) whenever it is called. Whereas, a function is compiled and executed every time whenever it is called
2. The function must return a value but in**Stored Procedure** it is optional. Even a procedure can return zero or n values.
3. Functions can have only input parameters for it whereas Procedures can have input or output parameters.
4. Functions can be called from Procedure whereas Procedures cannot be called from a Function.
5. Stored procedure can be debug while function cannot be
6. An exception can be handled by try-catch block in a Procedure whereas try-catch block cannot be used in a Function.
7. The procedure allows SELECT as well as DML(INSERT/UPDATE/DELETE) statement in it whereas Function allows only SELECT statement in it.
8. Stored Procedures cannot be used in the [SQL](https://en.wikipedia.org/wiki/SQL) statements anywhere in the WHERE/HAVING/SELECT section whereas Function can be.
9. Functions that return tables can be treated as another rowset. This can be used in JOINs with other tables.
10. Inline Function can be though of as views that take parameters and can be used in JOINs and other Rowset operations.
11. We can use Transactions in Procedure whereas we can't use Transactions in Function.
12. **What are the types of sub query?**

There are three types of sub query –

1. Single row sub query which returns only one row
2. Multiple row sub query which returns multiple rows
3. Multiple column sub query which returns multiple columns to the main query. With that sub query result, Main query will be executed.

* A correlated subquery cannot be considered as an independent query, but it can refer to the column in a table listed in the FROM of the main query.
* A non-correlated subquery can be considered as an independent query and the output of the subquery is substituted in the main query.

1. **What is the SQL Server Agent?**

SQL Server agent allows us to schedule the jobs and scripts.

1. ISNULL():

The ISNULL() function is used to replace NULL with the specified replacement value. This function contains only two arguments.

**Syntax**

ISNULL(UnitsOnOrder, 0))

1. Coalesce():

The Coalesce() function returns the first non-null value among its arguments. This function doesn't limit the number of arguments, but they must all be of the same data type.

 SELECT COALESCE(NULL, NULL, NULL, 'W3Schools.com', NULL, 'Example.com');

O/p- W3schools

**Syntax**

COALESCE ( expression [ ,...n ] )

COALESCE() function is equivalent to the following CASE expression.CASE

WHEN (exp1 IS NOT NULL) THEN exp1

WHEN (exp2 IS NOT NULL) THEN exp2

...

ELSE expN

1. Differences Between IsNull() and Coalesce() Functions

 The **ISNULL()** function contains only two parameters. The **COALESCE()**function contains multiple parameters. If we use more than two parameters with the ISNULL function then we must use nested ISNULL functions.

**Example 1**

ISNULL() function

**SELECT** ISNULL(NULL, NULL, 'Hello')

**OUTPUT**

Graphical user interface, text, application, email

Description automatically generated

COALESCE() function

**SELECT** COALESCE(NULL, NULL, 'hello')

**OUTPUT**

Graphical user interface, text, application, email

Description automatically generated

**4.**The **ISNULL()** function looks at the first value and the second parameter value is automatically limited to that length but **COALESCE()** does not have this restriction.

**Example 2**

**declare** @test **varchar**(3)

**select** isnull(@test, 'ABCD') **AS** ISNULLResult

**select** coalesce(@test, 'ABCD') **AS** coalesceResult

**OUTPUT**

Graphical user interface, text, application

Description automatically generated

In the above image, the test variable has length 3. So the ISNULL function returns **tes**and the COALESCE() function does not; depending on the length, it returns**test.**

**5.**The **ISNULL()** function contains various types of parameters. The **COALESCE()** function doesn't limit the number of arguments, but they must all be of the same data type.

**Example 3**

ISNULL() function

**DECLARE** @a **VARCHAR**(5)='Hello',

@b **INT** =5

**SELECT** ISNULL(@a, @b) **AS** ISNULLResult

**OUTPUT**

Graphical user interface, text, application

Description automatically generated

The COALESCE() function:

**DECLARE** @a **VARCHAR**(5)='Hello',

@b **INT** =5

**SELECT** COALESCE(@a, @b) **AS** COALESCEResult

**OUTPUT**

1. **What is the purpose of FLOOR function?**

FLOOR function is used to round up a non-integer value to the previous least integer. Example is given

FLOOR(6.7)

Returns 6.

1. **Can we check locks in database?**

Yes, using in-built stored procedure called sp\_lock.

1. **What is Bulkcopy in SQL?**

Bulkcopy is a tool used to copy large amount of data from Tables. This tool is used to load large amount of data in SQL Server.

1. **What will be query used to get the list of triggers in a database?**

Select \* from sys.objects where type='tr'

1. **What is the difference between UNION and UNION ALL?**

UNION: To select related information from two tables UNION command is used. It is similar to JOIN command.

UNION All: The UNION ALL command is equal to the UNION command, except that UNION ALL selects all values. It will not remove duplicate rows, instead it will retrieve all rows from all tables.

1. **What is the use of SET NOCOUNT ON/OFF statement?**
   * Returns number of rows affected
   * By default, NOCOUNT is set to OFF
2. **What are Magic Tables in SQL Server?**

During DML operations like Insert, Delete, and Update, SQL Server creates magic tables to hold the values during the DML operations. These magic tables are used inside the triggers for data transaction.

1. **What is the difference between SUBSTR and CHARINDEX in the SQL Server?**

The SUBSTR function is used to return specific portion of string in a given string. But, CHARINDEX function gives character position in a given specified string.

SUBSTRING('Smiley',1,3)

O/P: Smi

CHARINDEX('i', 'Smiley',1)

O/P: 3

1. **What will be the maximum number of index per table?**

1 Clustered Index and 999 Non-clustered indexes per table can be used in SQL Server.

1. **difference between varchar and nvarchar types:**

Varchar and nvarchar are same but the only difference is that nvarhcar can be used to store Unicode characters for multiple languages and it also takes more space when compared with varchar.

1. **What is Filtered Index?**

Filtered Index is used to filter some portion of rows in a table to improve query performance, index maintenance and reduces index storage costs. When the index is created with WHERE clause, then it is called Filtered Index

1. Collation:

Collation is a set of rules that tell database engine how to compare and sort the character data in SQL Server

Collations in SQL Server provide sorting rules, case, and accent sensitivity properties for your data. Collations that are used with character data types, such as **char** and **varchar**, dictate the code page and corresponding characters that can be represented for that data type.

It improves performance

1. Method to copy data from one table to another table:

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

Copy all columns from one table to another table:

INSERT INTO *table2*   
*SELECT* \* FROM *table1*   
*WHERE* *condition*;

Copy only some columns from one table into another table:

INSERT INTO *table2*(*column1*, *column2*, *column3*, ...)   
SELECT *column1*, *column2*, *column3*, ...   
FROM *table1*   
*WHERE* *condition*;

1. Remove duplicate records from table:

Delete from abc where Id not in (

Select Max(Id)  from abc

 group by name)

1. SQL Injection

SQL injection is a code injection technique that might destroy your database. SQL injection is the placement of malicious code in SQL statements, via web page input

Methods:

1. 1=1

Provide input in userid textbox of application.

UserId: 999 or 1=1

Then, the SQL statement will look like this:

SELECT \* FROM Users WHERE UserId = 105 OR 1=1;

The SQL above is valid and will return ALL rows from the "Users" table, since OR 1=1 is always TRUE.

A hacker might get access to all the user names and passwords in a database, by simply inserting 105 OR 1=1 into the input field.

1. ""=""   
   Username:" or ""="

Password:" or ""="

SELECT \* FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""

The SQL above is valid and will return all rows from the "Users" table, since OR ""="" is always TRUE.

1. SELECT \* FROM Users where userid=105; DROP TABLE Suppliers

Deletes the table

Use SQL Parameters for Protection

To protect a web site from SQL injection, you can use SQL parameters.

SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

Example:

txtUserId = getRequestString("UserId");   
txtSQL = "SELECT \* FROM Users WHERE UserId = @0";   
db.Execute(txtSQL,txtUserId);

1. **What is Normalization?**

The process of table design to minimize the data redundancy is called normalization. We need to divide a database into two or more tables and define relationships between them. Normalization usually involves dividing a database into two or more tables and defining relationships between the tables.

* **1NF (Eliminate Repeating Groups)**: Make a separate table for each set of related attributes, and give each table a primary key. Each field contains at most one value from its attribute domain.
* **2NF (Eliminate Redundant Data)**: If an attribute depends on only part of a multi-valued key, remove it to a separate table.
* **3NF (Eliminate Columns Not Dependent On Key)**: If attributes do not contribute to the description of the key, remove them to a separate table. All attributes must be directly dependent on the primary key.
* **BCNF (Boyce-Codd Normal Form):** If there are non-trivial dependencies between candidate key attributes, separate them into distinct tables.
* **4NF (Isolate Independent Multiple Relationships):** No table may contain two or more 1:n or n:m relationships that are not directly related.
* **5NF (Isolate Semantically Related Multiple Relationships):** There may be practical constraints on information that justifies separating logically related many-to-many relationships.
* **ONF (Optimal Normal Form):** A model limited to only simple (elemental) facts, as expressed in Object Role Model notation.
* **DKNF (Domain-Key Normal Form):** A model free from all modification is said to be in DKNF.

1. **Define UNION, UNION ALL, MINUS, INTERSECT?**

**UNION –** returns all distinct rows selected by either query.

**UNION ALL –** returns all rows selected by either query, including all duplicates.

**MINUS –** The MINUS operator in SQL is used to remove duplicates from the result-set obtained by the second SELECT query from the result-set obtained by the first SELECT query and then return the filtered results from the first.

First query result – second query result

**INTERSECT –** returns all distinct rows selected by both queries.

**Delete**- DELETE statement is used to delete rows from a table.

**DELETE** **FROM** Candidates   
**WHERE** CandidateId > 1000;

**TRUNCATE**- delete all the rows from the table and free the space containing the table.

**TRUNCATE** **TABLE** Candidates;

**DROP**- All the rows in the table are deleted and the table structure is removed from the database.

1. What is the difference between DELETE and TRUNCATE statements?

* The TRUNCATE command is used to delete all the rows from the table and free the space containing the table.
* The DELETE command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified. But it does not free the space containing the table.

1. View:
   1. Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

Syntax for view creation:

CREATE VIEW view\_name AS   
SELECT column1, column2.....   
FROM table\_name   
WHERE condition;

Syntax for executing view:

SELECT \* FROM view\_name;

Syntax for deletion of view:

DROP VIEW view\_name;

1. **Uses of a View :**   
   A good database should contain views due to the given reasons:
   1. Restricting data access –   
      Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
   2. Hiding data complexity –   
      A view can hide the complexity that exists in a multiple table join.
   3. Simplify commands for the user –   
      Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.
   4. Store complex queries –   
      Views can be used to store complex queries.
   5. Rename Columns –   
      Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to to hide the names of the columns of the base tables.
   6. Multiple view facility –   
      Different views can be created on the same table for different users.
2. Function:
   1. Function is to select values from database
   2. It is of two types:
      1. Scalar function – returns only single value
      2. Table function – returns data as table
   3. Execute function:
      1. SELECT [dbo].AddNumbers(14,97,24) AS Addition
      2. Where 14, 97,24 are parameters and addition is the name of the column under which we will display the result. It is optional to specify the column name.
3. Trigger:
   1. A trigger is special type of stored procedure that is automatically executed in response to certain events on a specified table. John is the marketing officer in a company. When a new customer data is entered into the company’s database he has to send the welcome message to each new customer. If it is one or two customers John can do it manually, but what if the count is more than a thousand? Well in such scenario triggers come in handy.

Thus, now John can easily create a trigger which will automatically send a welcome email to the new customers once their data is entered into the database.

* 1. Always remember that there cannot be two triggers with similar action time and event for one table. For example, we cannot have two BEFORE UPDATE triggers for a table. But we can have a *BEFORE UPDATE* and a *BEFORE INSERT* trigger, or a *BEFORE UPDATE* and an *AFTER UPDATE* trigger.

Syntax:

Create Trigger Trigger\_Name

(Before | After)  [ Insert | Update | Delete]

on [Table\_Name]

[ for each row | for each column ]

[ trigger\_body ]

Example:

CREATE TRIGGER sample\_trigger

before INSERT

ON student

FOR EACH ROW

SET new.total = new.marks/6

1. Self join
   1. SELECT \* FROM T t1 [INNER | LEFT] JOIN T t2 ON join\_predicate;
   2. A self join uses the [inner join](https://www.sqlservertutorial.net/sql-server-basics/sql-server-inner-join/) or [left join](https://www.sqlservertutorial.net/sql-server-basics/sql-server-left-join/) clause. Because the query that uses the self join references the same table, the [table alias](https://www.sqlservertutorial.net/sql-server-basics/sql-server-alias/) is used to assign different names to the same table within the query.
2. Cross join:

<https://www.sqlshack.com/sql-cross-join-with-examples/>

1. Group By:

* The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause or before having clause.
* SELECT `category\_id`,`year\_released` FROM `movies` GROUP BY `category\_id`,`year\_released`
* The main difference between the WHERE and HAVING clauses comes when used together with the GROUP BY clause. In that case, WHERE is used to filter rows before grouping, and HAVING is used to exclude records after grouping.
* Another worth noting thing about the WHERE and HAVING clause is that the WHERE clause cannot contain aggregate functions like COUNT(), SUM(), MAX(), MIN(), etc but the HAVING clause may contain aggregate functions.
* When to use WHERE and HAVING clauses?

Though both are used to exclude rows from the result set, you should use the WHERE clause to filter rows before grouping and use the HAVING clause to filter rows after grouping. In other words, WHERE can be used to filter on table columns while HAVING can be used to filter on aggregate functions like count, sum, avg, min, and max.   
   
If filtering can be done without aggregate function then you must do it on the [WHERE clause](https://javarevisited.blogspot.com/2013/08/difference-between-where-vs-having-clause-SQL-databse-group-by-comparision.html) because it improves performance because counting and sorting will be done on a much smaller set. If you filter the same rows after grouping, you unnecessarily bear the cost of sorting, which is not used. 

1. Properties of transaction

Diagram

Description automatically generated

DSA

1. **What is a Data Structure?**    
   A way of organizing the data so that it can be used efficiently
2. **What are the various operations that can be performed on different Data Structures?**

Insertion ,Deletion,Traversal, Searching, Sorting,

1. [**How is an Array different from Linked List?**](https://www.geeksforgeeks.org/linked-list-vs-array/)
   1. The size of the arrays is fixed, Linked Lists are Dynamic in size.
   2. Inserting and deleting a new element in an array of elements is expensive, Whereas both insertion and deletion can easily be done in Linked Lists.
   3. Extra memory space for a pointer is required with each element of the Linked list.
   4. Arrays have better cache locality that can make a pretty big difference in performance.
2. **What is Stack and where it can be used?**    
   Stack - LIFO(Last In First Out) or FILO(First In Last Out) for accessing elements.
   1. Basic operations of the stack are: **Push, Pop, Peek**
3. **What is a Queue, how it is different from the** **stack and how is it implemented?**    
   [Queue -](http://en.wikipedia.org/wiki/Queue_%28data_structure%29)**F**irst **I**n **F**irst **O**ut (FIFO) to access elements.
   1. basic operations on queue: **Enqueue, Dequeue**, **Front, Rear**
4. **What is a Linked List and What are its types?**

 Each node of a list is comprising of two items – the data and a reference to the next node. Types of Linked List :

1. **Singly Linked List :**In this type of linked list, every node stores address or reference of next node in list and the last node has next address or reference as NULL. For example 1->2->3->4->NULL
2. **Doubly Linked List :**Here,here are two references associated with each node, One of the reference points to the next node and one to the previous node. Eg. NULL<-1<->2<->3->NULL
3. **Circular Linked List :**Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list. Eg. 1->2->3->1 [The next pointer of last node is pointing to the first]
4. Why we need to do algorithm analysis?

A problem can be solved in more than one ways. So, to find and implement the best suitable algorithm.

1. What are the criteria of algorithm analysis?

time and space

1. What is asymptotic analysis of an algorithm?

Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.

1. What are asymptotic notations?

Best case is represented by Ω(n) notation.

Worst case is represented by Ο(n) notation.

Average case is represented by Θ(n) notation.

1. Search methods:

It is to search element in data list like array, linked list

1. Linear Search

The linear search algorithm iteratively searches all elements of the array. It has the best execution time of one and the worst execution time of n, where n is the total number of items in the search array.

It is the simplest search algorithm in data structure and checks each item in the set of elements until it matches the searched element till the end of data collection. When the given data is unsorted, a linear search algorithm is preferred over other search algorithms.

1. Binary Search

This algo first sort the array and then middlemost items in the data collection. When a match is found, it returns the index of the item. When the middle item is greater than the search item, it looks for a central item of the left sub-array. If, on the other hand, the middle item is smaller than the search item, it explores for the middle item in the right sub-array. It keeps looking for an item until it finds it or the size of the sub-arrays reaches zero.

* It works faster than a linear search algorithm.
* The binary search uses the divide and conquers principle.

A ← sorted array   
n ← size of array   
x ← value to be searched   
   
Set lowerBound = 1   
Set upperBound = n    
   
while x not found   
if upperBound < lowerBound    
EXIT: x does not exists.   
   
set midPoint = lowerBound + ( upperBound - lowerBound ) / 2   
   
if A[midPoint] x   
set upperBound = midPoint - 1    
   
if A[midPoint] = x    
EXIT: x found at location midPoint   
end while   
   
end procedure

1. What is a binary tree?

A binary tree has a special condition that each node can have two children at maximum.

1. What is a binary search tree?

A binary search tree is a binary tree with a special provision where a node's left child must have value less than its parent's value and node's right child must have value greater than it's parent value.

1. What is fibonacci series?

generates subsequent number by adding two previous numbers. For example − 0 1 1 2 3 5 8 13

1. What is a heap in data structure?

Heap is a special balanced binary tree data structure where root-node key is compared with its children and arranged accordingly. A min-heap, a parent node has key value less than its childs and a max-heap parent node has value greater than its childs.

4. How to implement a stack using queue?
5. difference between bfs and dfs   
   How to implement a queue using stack?
6. sorting algorithms dsa

Design pattern

Types of design pattern:

Creational – deals with object creation mechanism

* 1. Singleton
  2. Factory
  3. Abstract factory
  4. Builder
  5. Prototype

1. Structural
2. Behavioural

* Singleton Design method:

In this pattern, a class has only one instance in the program that provides a global point of access to it. In other words, a singleton is a class that allows only a single instance of itself to be created

There are various ways to implement a singleton pattern in C#. The following are the common characteristics of a singleton pattern.

* + Private and parameterless single constructor
  + Sealed class.
  + Static variable to hold a reference to the single created instance
  + A public and static way of getting the reference to the created instance.

SOLID principle

Dependency Injection:

* Dependency means object
* It is a process of injecting object of a class in constructor of another class
* The use of an interface or base class to abstract the dependency implementation.
* Registration of the dependency in a service container. .NET provides a built-in service container, [IServiceProvider](https://docs.microsoft.com/en-us/dotnet/api/system.iserviceprovider" \t "_blank). Services are typically registered at the app's start-up and appended to an [IServiceCollection](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.iservicecollection" \t "_blank). Once all services are added, you use [BuildServiceProvider](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.servicecollectioncontainerbuilderextensions.buildserviceprovider" \t "_blank) to create the service container.
* *Injection* of the service into the constructor of the class where it's used. The framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.
* It is an example of singleton design pattern
* https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

--------------------------------

S - Single Responsibility Principle

* every class, or similar structure, in your code should have only one job to do. It should be about 1 topic or should contains code related to only 1 topic. Don't write all code in 1 file or 1 method

O - Open Closed Principle

* open for extension, closed for modification. "Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated. "Closed for modification" means we have already developed a class and it has gone through unit testing. We should then not alter it until we find bugs. As it says, a class should be open for extensions, we can use inheritance to do this.

Suppose we have a Rectangle class with the properties Height and Width.

**public** **class** Rectangle{

**public** **double** Height {**get**;**set**;}

**public** **double** Wight {**get**;**set**; }

}

Our app needs the ability to calculate the total area of a collection of Rectangles. Since we already learned the Single Responsibility Principle (SRP), we don't need to put the total area calculation code inside the rectangle. So here I created another class for area calculation.

**public** **class** AreaCalculator {

**public** **double** TotalArea(Rectangle[] arrRectangles)

   {

**double** area;

**foreach**(var objRectangle **in** arrRectangles)

      {

         area += objRectangle.Height \* objRectangle.Width;

      }

**return** area;

   }

}

Now we need to calculate area of circle:

**public** **class** Rectangle{

**public** **double** Height {**get**;**set**;}

**public** **double** Wight {**get**;**set**; }

}

**public** **class** Circle{

**public** **double** Radius {**get**;**set**;}

}

**public** **class** AreaCalculator

{

**public** **double** TotalArea(**object**[] arrObjects)

   {

**double** area = 0;

      Rectangle objRectangle;

      Circle objCircle;

**foreach**(var obj **in** arrObjects)

      {

**if**(obj **is** Rectangle)

         {

            area += obj.Height \* obj.Width;

         }

**else**

         {

            objCircle = (Circle)obj;

            area += objCircle.Radius \* objCircle.Radius \* Math.PI;

         }

      }

**return** area;

   }

}

Now we need to calculate area of triangle and calculate it's the area by adding one more "if" block in the TotalArea method of AreaCalculator. But every time we introduce a new shape we need to alter the TotalArea method. So the AreaCalculator class is not closed for modification. How can we make our design to avoid this situation? Generally, we can do this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes. Such interfaces can be fixed once developed so the classes that depend upon them can rely upon unchanging abstractions. Functionality can be added by creating new classes that implement the interfaces. So let's refract our code using an interface.

**public** **abstract** **class** Shape

{

**public** **abstract** **double** Area();

}

Inheriting from Shape, the Rectangle and Circle classes now look like this:

**public** **class** Rectangle: Shape

{

**public** **double** Height {**get**;**set**;}

**public** **double** Width {**get**;**set**;}

**public** **override** **double** Area()

   {

**return** Height \* Width;

   }

}

**public** **class** Circle: Shape

{

**public** **double** Radius {**get**;**set**;}

**public** **override** **double** Area()

   {

**return** Radius \* Radus \* Math.PI;

   }

}

Every shape contains its area with its own way of calculation functionality and our AreaCalculator class will become simpler than before.

**public** **class** AreaCalculator

{

**public** **double** TotalArea(Shape[] arrShapes)

   {

**double** area=0;

**foreach**(var objShape **in** arrShapes)

      {

         area += objShape.Area();

      }

**return** area;

   }

}

L - Liskov Substitution Principle

* the objects of the subclasses should behave in the same way as the objects of the superclass

[(63) The Liskov Substitution Principle (LSP) | LinkedIn](https://www.linkedin.com/pulse/liskov-substitution-principle-lsp-paul-gichure-ctfl/)

I - Interface Segregation Principle

clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.

D - Dependency Inversion Principle

* The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.

https://dev.to/tamerlang/understanding-solid-principles-dependency-inversion-1b0f

C# concept –

1. C#-

It is object oriented programming language developed by Microsoft use to create web applications

1. What is object-oriented programming?

Object-oriented programming (OOP) - software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior. OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them.

1. What is garbage collection in C#?

Garbage collection is the process of freeing up memory that is captured by unwanted objects. When you create a class object, automatically some memory space is allocated to the object in the heap memory. Now, after you perform all the actions on the object, the memory space occupied by the object becomes waste. It is necessary to free up memory. Garbage collection happens in three cases:

* If the occupied memory by the objects exceeds the pre-set threshold value.
* If the garbage collection method is called
* If your system has low physical memory

You can call method by calling system.GC.collect().

The CLR (Common Language Runtime) adopts garbage collection as a mechanism to clean up the resources consumed by your application. Note that when you create objects in .Net, they are stored in the managed heap, and when you are done using them, you need not worry about cleaning them -- the runtime would do it for you.

The CLR organizes the managed heap into generations. The three generations into which the managed heap is organized are: Generation 0, Generation 1, and Generation 2. The GC is adept at reclaiming the memory occupied by managed objects. However, you should follow certain guidelines to facilitate faster garbage collection so as to improve your application's performance.

**Should I use the GC.Collect() method?**

We mostly don't need to call manually. When you make a call to GC.Collect() method, the runtime performs a stack walk to determine the objects that are reachable and those that aren't. It also freezes the main thread (and also any child threads it created) of the application. In other words, when the GC.Collect() method is called, the runtime performs a blocking garbage collection of all generations.

I would always prefer not to use GC.Collect() unless there is a specific reason to use it.

GC.Collect() – used to collect objects present in the generations 0, 1, 2

GC.Collect(0) – used to collect objects present in generation 0

GC.Collect(1) – used to collect objects present in generations 0 and 1

GC.GetTotalMemory()  - To determine how much memory has been freed by making a call to the GC.Collect() method.

System.GC.GetGeneration(obj) - To determine obj object belongs to which generation

GC.WaitForPendingFinalizers()  - Note that objects that have finalizers (and if a call to SuppressFinalize method hasn’t been made) would not be collected when a call to GC.Collect() method is made. Rather, such objects would be placed in the finalization queue. If you would like to collect those objects as well, you would need to make a call to the GC.WaitForPendingFinalizers() method so that those objects are cleaned up when the next GC cycle runs. In essence, reclaiming the memory occupied by objects that have finalizers implemented requires two passes since such objects are placed in the finalization queue rather than being reclaimed in the first pass when the garbage collector runs.

Generation 0. This is the youngest generation and contains short-lived objects like temporary variable. Garbage collection occurs most frequently in this generation. Newly allocated objects form a new generation of objects and are implicitly generation 0 collections, unless they are large objects, in which case they go on the large object heap in a generation 2 collection.

Generation 1. This generation contains short-lived objects and serves as a buffer between short-lived objects and long-lived objects.

Generation 2. This generation contains long-lived objects like static data that is live for the duration of the process.

1. Static class - Static class, defined by the keyword ‘static’ does not allow inheritance and object creation of class. The advantage of using a static class is that**the compiler can check to make sure that no instance members are accidentally added**. static classes are only allowed to have static methods. non-static classes may contain both static and instance methods,

Static classes cannot contain an instance constructor. However, they can only contain a static constructor.

1. **Can a non-static class have a static constructor?**

Yes, Non-static classes should define a static constructor if the class contains static members that require initialization.

1. Partial class: Partial class, defined by the keyword ‘partial’ allows its members to partially divide or share source (.cs) files.
2. Sealed class: Sealed classes are classes that cannot be inherited. Eg: Struct

**class** X

{

**protected** **virtual** **void** F()

    {

        Console.WriteLine("X.F");

    }

**protected** **virtual** **void** F2()

    {

        Console.WriteLine("X.F2");

    }

}

**class** Y : X

{

**sealed** **protected** **override** **void** F()

    {

        Console.WriteLine("Y.F");

    }

**protected** **override** **void** F2()

    {

        Console.WriteLine("X.F3");

    }

}

**class** Z : Y

{

    // Attempting to override F causes compiler error CS0239.

    //

**protected** **override** **void** F()

    {

         Console.WriteLine("C.F");

    }

    // Overriding F2 is allowed.

**protected** **override** **void** F2()

    {

        Console.WriteLine("Z.F2");

    }

}

1. **Abstract class:**

The abstract modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, not instantiated on its own. Members marked as abstract must be implemented by non-abstract classes that derive from the abstract class.

Abstract classes are classes that cannot be instantiated where you cannot create objects. Abstract classes work on the OOPS concept of abstraction. Abstraction helps to extract essential details and hide the unessential ones.

Abstract classes have the following features:

* An abstract class cannot be instantiated.
* An abstract class may contain abstract methods and accessors.
* It is not possible to modify an abstract class with the [sealed](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed) modifier because the two modifiers have opposite meanings. The sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.
* A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.
* It is an error to use the [static](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static) or [virtual](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual) modifiers in an abstract method declaration.
* It is an error to use the abstract modifier on a static property.
* **We cannot create instance of abstract class because it does not have a complete implementation**.

1. What is the difference between an abstract class and an interface?

* abstract classes can have both abstract and non-abstract methods but all the methods of an interface are abstract methods.
* Since abstract classes can have both abstract and non-abstract methods, we need to use the Abstract keyword to declare abstract methods. But in the interface, there is no such need.
* An abstract class has constructors while an interface encompasses none.

1. Like a class, Interface can have methods, properties, events, and indexers as its members. But interfaces will contain only the declaration of the members. ... Interface cannot contain fields because they represent a particular implementation of data.

**Interface cannot contain a field**

interface IEmployee {

string Name {

get;

set; }

int Counter { get; } }

1. C# doesn’t support multiple inheritances. Instead, you can use interfaces to inherit the properties using the class name in the signature.
2. What is Boxing and Unboxing in C#?

Boxing: Boxing converts value type (int, char, etc.) to reference type (object) which is an implicit conversion process using object value.

int num = 23; // 23 will assigned to num   
Object Obj = num; // Boxing

Unboxing: Unboxing converts reference type (object) to value type (int, char, etc.) using an explicit conversion process.

*Example:*int i = (int)Obj;

1. Difference between the Equality Operator (==) and Equals() Method in C#?

Although both are used to compare two objects by value, still they both are used differently.

Equality operator (==) is a reference type which means that if equality operator is used, it will return true only if both the references point to the same object.     
The == Operator compares the reference identity while the Equals() method compares only contents.

Equals() method: Equals method is used to compare the values carried by the objects. int x=10, int y=10. If x==y is compared then, the values carried by x and y are compared which is equal and therefore they return true.

Equality operator: Compares by reference   
   
Equals(): Compares by value

1. What are the different ways in which a method can be Overloaded in C#?

Overloading means when a method has the same name but different parameters. Only the main() method cannot be overloaded.

In order to overload methods in C#,

* Change the number of parameters in a method, or
* Change the order of parameters in a method, or
* Use different data types for parameters

1. What is the difference between a struct and a class in C#?

 Struct

* The struct is a value type in C# and it inherits from System.Value Type.
* Struct is usually used for smaller amounts of data.
* Struct can’t be inherited from other types.
* A structure can't be abstract.
* No need to create an object with a new keyword.
* Do not have permission to create any default constructor.
* We cannot define default constructor in structure. If we try to define then the compiler will produce an error . We can define static constructor and parameterised constructor.

// Static Constructor, valid

**static** Student()

            {

                Console.WriteLine("Static Constructor");

            }

            // Struct does not contain Default Constructor

            // It will produce an error as "Structs cannot contain explicit parameterless constructor"

**public** Student()

            {

                Console.WriteLine("Default Constructor");

            }

            // Parameterised Constructor

            // Valid, This constructor should return all the values of the struct members and must contain all the arguments

**public** Student(**int** roll, **string** name, **string** mobile, **int** serial)

            {

               Serial\_No = serial;

            }

// Destructor

            // Struct cannot contain destructors. It will produce a compile error as "Only class types can contain destructors"

            ~Student()

            {

                Console.WriteLine("Destructor");

            }

Class

* The class is a reference type in C# and it inherits from the System.Object Type.
* Classes are usually used for large amounts of data.
* Classes can be inherited from other classes.
* A class can be an abstract type.
* We can create a default constructor.

1. **When to use Structures? Is There a Difference between Class and Structure?**

 1) Structures provide better performance when we have small collections of value-types that you want to group together.

 3) In C#, using a value type instead of a reference type will result in fewer objects on the managed heap, which results in a lesser load on the garbage collector (GC), less frequent GC cycles, and consequently better performance. However, value types have their downsides too. Passing around a big struct is definitely costlier than passing a reference, that's one obvious problem.

4) A class is a reference type. When an object of the class is created, the variable to which the object is assigned holds only a reference to that memory. When the object reference is assigned to a new variable, the new variable refers to the original object. Changes made through one variable are reflected in the other variable because they both refer to the same data. A struct is a value type. When a struct is created, the variable to which the struct is assigned holds the struct's actual data. When the struct is assigned to a new variable, it is copied. The new variable and the original variable, therefore, contain two separate copies of the same data. Changes made to one copy do not affect the other copy. In general, classes are used to model more complex behavior or data that is intended to be modified after a class object is created. Structs are best suited for small data structures that contain primarily data that is not intended to be modified after the struct is created.

5) A struct is a value type. If you assign a struct to a new variable, the new variable will contain a copy of the original.

**public** **struct** IntStruct

{

**public** **int** Value { **get**; **set**; }

}

var struct1 = **new** IntStruct() { Value = 0 }; // original

var struct2 = struct1; // A copy is made

var struct3 = struct2; // A copy is made

// NOTE: A "copy" will occur when you pass a struct into a method parameter.

// To avoid the "copy", use the ref keyword.

// Although structs are designed to use less system resources

// than classes. If used incorrectly, they could use significantly more.

A class is a reference type. When you assign a class to a new variable, the variable contains a reference to the original class object.

**public** **class** IntClass

{

**public** **int** Value { **get**; **set**; }

}

Execution of the following results in only one instance of the class object in memory.

var class1 = **new** IntClass() { Value = 0 };

var class2 = class1; // A reference is made to class1

1. What is the difference between constant and readonly in C#?

Const is nothing but "constant", a variable of which the value is constant but at compile time. It's mandatory to assign a value to it. By default, a const is static and we cannot change the value of a const variable throughout the entire program.

Readonly is the keyword whose value we can change during runtime or we can assign it at run time but only through the non-static constructor.

1. What is the difference between the dispose and finalize methods in C#?

Both are used to free unmanaged resources that are not in use, like files, database connections in the application domain and more. But the difference is finalize is the Internal process, it is called by Garbage Collector and can’t be called manual by user code or any service. While dispose has to be called manualy

1. Difference between array and arraylist

Array stores only specific data type variables while arraylist stores all type of variable as an object

When declaring an array the size of the items is fixed therefore, the memory allocation is fixed. But with ArrayList, it can be increased or decreased dynamically.

While arrays cannot accept null, ArrayList can accept null values.

1. Static Constructor

A constructor can be static. You create a static constructor to initialize static fields. Static constructors are not called explicitly with the new statement. They are called when the class is first referenced. There are some limitations of the static constructor as in the following;

* Static constructors are parameterless.
* Static constructors can't be overloaded.
* There is no accessibility specified for Static constructors.

1. **Destructor:**

* Destructors are parameterless.
* A Destructor can't be overloaded.
* Destructors are not inherited.
* Destructors can cause performance and efficiency implications.

1. Difference between String & string?

[string](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/string) is an alias in C# for [System.String](https://docs.microsoft.com/en-us/dotnet/api/system.string" \t "_blank).   
So technically, there is no difference. It's like [int *vs.* System.Int32](https://stackoverflow.com/questions/62503/c-int-or-int32-should-i-care).

1. **Extension Method:**

* Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.
* Extension methods are static methods
* The most common extension methods are the LINQ standard query operators that add query functionality to the existing [System.Collections.IEnumerable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable) and [System.Collections.Generic.IEnumerable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1) types. To use the standard query operators, first bring them into scope with a using System.Linq directive. Then any type that implements [IEnumerable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1" \t "_blank) appears to have instance methods such as [GroupBy](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.groupby" \t "_blank), [OrderBy](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.orderby" \t "_blank), [Average](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.average), and so on.
* Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on. The parameter is preceded by the [this](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/this) modifier. Extension methods are only in scope when you explicitly import the namespace into your source code with a using directive.
* You can use extension methods to extend a class or interface, but not to override them.
* namespace ExtensionMethods

{   
    public static class MyExtensions   
    {   
        public static int WordCount(this String str)   
        {   
            return str.Split(new char[] { ' ', '.', '?' },StringSplitOptions.RemoveEmptyEntries).Length;   
        }   
    }}

The WordCount extension method can be brought into scope with this using directive:

using ExtensionMethods; 

And it can be called from an application by using this syntax:

string s = "Hello Extension Methods";

int i = s.WordCount(); 

Or

string s = "Hello Extension Methods";   
int i = MyExtensions.WordCount(s);

* Extension methods cannot access private variables in the type they are extending.
* While it's still considered preferable to add functionality by modifying an object's code or deriving a new type whenever it's reasonable and possible to do so, extension methods have become a crucial option for creating reusable functionality throughout the .NET ecosystem. For those occasions when the original source isn't under your control, when a derived object is inappropriate or impossible, or when the functionality shouldn't be exposed beyond its applicable scope, Extension methods are an excellent choice.
* When using an extension method to extend a type whose source code you aren't in control of, you run the risk that a change in the implementation of the type will cause your extension method to break.

1. **Binding Extension Methods at Compile Time**

* You can use extension methods to extend a class or interface, but not to override them. An extension method with the same name and signature as an interface or class method will never be called. At compile time, extension methods always have lower priority than instance methods defined in the type itself. In other words, if a type has a method named Process(int i), and you have an extension method with the same signature, the compiler will always bind to the instance method. When the compiler encounters a method invocation, it first looks for a match in the type's instance methods. If no match is found, it will search for any extension methods that are defined for the type, and bind to the first extension method that it finds. The following example demonstrates how the compiler determines which extension method or instance method to bind to.
* Example
* The following example demonstrates the rules that the C# compiler follows in determining whether to bind a method call to an instance method on the type, or to an extension method. The static class Extensions contains extension methods defined for any type that implements IMyInterface. Classes A, B, and C all implement the interface.
* The MethodB extension method is never called because its name and signature exactly match methods already implemented by the classes.
* When the compiler can't find an instance method with a matching signature, it will bind to a matching extension method if one exists.

// Define an interface named IMyInterface.

namespace DefineIMyInterface

{

    using System;

    public interface IMyInterface

    {

        // Any class that implements IMyInterface must define a method

        // that matches the following signature.

        void MethodB();

    }

}

// Define extension methods for IMyInterface.

namespace Extensions

{

    using System;

    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any

    // class that implements IMyInterface.

    public static class Extension

    {

        public static void MethodA(this IMyInterface myInterface, int i)

        {

            Console.WriteLine

                ("Extension.MethodA(this IMyInterface myInterface, int i)");

        }

        public static void MethodA(this IMyInterface myInterface, string s)

        {

            Console.WriteLine

                ("Extension.MethodA(this IMyInterface myInterface, string s)");

        }

        // This method is never called in ExtensionMethodsDemo1, because each

        // of the three classes A, B, and C implements a method named MethodB

        // that has a matching signature.

        public static void MethodB(this IMyInterface myInterface)

        {

            Console.WriteLine

                ("Extension.MethodB(this IMyInterface myInterface)");

        }

    }

}

// Define three classes that implement IMyInterface, and then use them to test

// the extension methods.

namespace ExtensionMethodsDemo1

{

    using System;

    using Extensions;

    using DefineIMyInterface;

    class A : IMyInterface

    {

        public void MethodB() { Console.WriteLine("A.MethodB()"); }

    }

    class B : IMyInterface

    {

        public void MethodB() { Console.WriteLine("B.MethodB()"); }

        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }

    }

    class C : IMyInterface

    {

        public void MethodB() { Console.WriteLine("C.MethodB()"); }

        public void MethodA(object obj)

        {

            Console.WriteLine("C.MethodA(object obj)");

        }

    }

    class ExtMethodDemo

    {

        static void Main(string[] args)

        {

            // Declare an instance of class A, class B, and class C.

            A a = new A();

            B b = new B();

            C c = new C();

            // For a, b, and c, call the following methods:

            //      -- MethodA with an int argument

            //      -- MethodA with a string argument

            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to

            // the extension method that has a matching signature.

            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)

            a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // A has a method that matches the signature of the following call

            // to MethodB.

            a.MethodB();            // A.MethodB()

            // B has methods that match the signatures of the following

            // method calls.

            b.MethodA(1);           // B.MethodA(int)

            b.MethodB();            // B.MethodB()

            // B has no matching method for the following call, but

            // class Extension does.

            b.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // C contains an instance method that matches each of the following

            // method calls.

            c.MethodA(1);           // C.MethodA(object)

            c.MethodA("hello");     // C.MethodA(object)

            c.MethodB();            // C.MethodB()

        }

    }

}

/\* Output:

    Extension.MethodA(this IMyInterface myInterface, int i)

    Extension.MethodA(this IMyInterface myInterface, string s)

    A.MethodB()

    B.MethodA(int i)

    B.MethodB()

    Extension.MethodA(this IMyInterface myInterface, string s)

    C.MethodA(object obj)

    C.MethodA(object obj)

    C.MethodB()

\*/

1. What's the Difference between the Is and As operator

* "is" operator to check the object type. If two objects are of the same type, it returns true, else it returns false.
* The "as" operator behaves in a similar way as the "is" operator. The only difference is it returns the object if both are compatible with that type. Else it returns a null.

1. What is Serialization in C#?

Serialization in C# is the process of converting an object into a stream of bytes to store the object to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

1. What are the benefits of OOP?

* **Modularity.** Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
* **Reusability.** Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.
* **Productivity.** Programmers can construct new programs quicker through the use of multiple libraries and reusable code.
* **Easily upgradable and scalable.**

1. Clone() Vs CopyTo() Array

* The **Clone()** method returns a new array object containing all the elements in the original array.
* The **CopyTo()** method copies the elements into another existing array. It copies the elements of one array to another **pre-existing** array starting from a given index (usually 0).

1. **Generic type:**

* It improves reusability
* ArrayList is of non-generic type as it stores all data in object form

// Declare the generic class.

public class GenericList<T>

{

    public void Add(T input) { }

}

class TestGenericList

{

    private class ExampleClass { }

    static void Main()

    {

        // Declare a list of type int.

        GenericList<int> list1 = new GenericList<int>();

        list1.Add(1);

        // Declare a list of type string.

        GenericList<string> list2 = new GenericList<string>();

        list2.Add("");

        // Declare a list of type ExampleClass.

        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();

        list3.Add(new ExampleClass());

    }

}

1. Anonymous Types

* Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first.
* The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.
* You create anonymous types by using the [new](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/new-operator) operator together with an object initializer

var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following

// statement to verify that their inferred types are int and string.

Console.WriteLine(v.Amount + v.Message);

* Anonymous types typically are used in the [select](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/select-clause) clause of a query expression to return a subset of the properties from each object in the source sequence.
* Anonymous types contain one or more public read-only properties. No other kinds of class members, such as methods or events, are valid. The expression that is used to initialize a property cannot be null, an anonymous function, or a pointer type.
* var productQuery = from prod in products select new { prod.Color, prod.Price };

1. Reflection:

**Reflection** objects are used for obtaining type information at runtime. The classes that give access to the metadata of a running program are in the **System.Reflection** namespace.or get the type from an existing object

int i = 42; 

Type type = i.GetType(); 

//The output is: System.Int32.

1. Abstract Class:

* An abstract class is an incomplete class or special class we can't be instantiated. The purpose of an abstract class is to provide a blueprint for derived classes and set some rules what the derived classes must implement when they inherit an abstract class.
* An abstract class can inherit from a class and one or more interfaces.
* Abstract class can have abstract or non-abstract method
* An abstract class cannot be inherited from by structures or we cannot inherit struct class in abstract class.
* An abstract class cannot support multiple inheritance.

**abstract** **class** Container : B //But we can't inherit like this : A,B

{

}

1. Difference between IQuerable and Ienumerable:

IQuerable:

1. IQueryable is suitable for querying data from out-memory (like remote database, service) collections.
2. While querying data from a database, IQueryable executes a "select query" on server-side with all filters.

IEnumerable:

1. IEnumerable is suitable for querying data from in-memory collections like List, Array and so on.
2. While querying data from the database, IEnumerable executes "select query" on the server-side, loads data in-memory on the client-side and then filters the data.
3. What is the use of ‘using’ statement in C#?

The ‘using’ block is used to obtain a resource and process it and then automatically dispose of when the execution of the block completed.

1. **No, we can't use “this” keyword** inside a static method. “this” refers to current instance of the class. But if we define a method as static , class instance will not have access to it, only CLR executes that block of code. Hence we can't use “this” keyword inside static method
2. Can a private virtual method can be overridden?

No, because they are not accessible outside the class.

1. What is the base class in .net from which all the classes are derived from?

System.Object

1. Why can’t you specify the accessibility modifier for methods inside the interface?

In an interface, we have virtual methods that do not have method definition. All the methods are there to be overridden in the derived class. That’s why they all are public.

1. How can we set the class to be inherited, but prevent the method from being over-ridden?

Declare the class as public and make the method sealed to prevent it from being overridden.

1. What is difference between the “throw” and “throw ex” in .NET?

“Throw” statement preserves original error stack whereas “throw ex” have the stack trace from their throw point. It is always advised to use “throw” because it provides more accurate error information.

1. **Question: How can you describe object-oriented concepts in detail?**

**Encapsulation**: defines the binding together code and the data and keeps it safe from any manipulation done by other programs and classes. It is a container that prevents code and data from being accessed by another program that is defined outside the container.

**Abstraction**: this concept of object-oriented protects everything other than the relevant data about any created object in order to increase efficiency and security within the program.

**Inheritance**: Inheritance is applied in such a way where one object uses the properties of another object.

**Polymorphism**: is a feature that allows one interface to act as a base class for other classes. This concept is often expressed as a "single interface but multiple actions".

1. **Destructor**

A destructor is a member that works just the opposite of the constructor. Unlike constructors, destructors mainly delete the object. The destructor name must match exactly with the class name just like a constructor. A destructor block always starts with the tilde (~) symbol.

**Syntax:**

~class\_name()   
{   
//code   
}

A destructor is **called automatically:**

1. when the program finishes its execution.
2. Whenever a scope of the program ends that defines a local variable.
3. Whenever you call the delete operator from your program.
4. Types of parameter:
   1. [params](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/params) specifies that this parameter may take a variable number of arguments.
   2. [in](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/in-parameter-modifier) specifies that this parameter is passed by reference but is only read by the called method.
      1. Its reference type but ensures the argument is not modified.
      2. It is just readonly but cannot modified by the called method.
      3. Variables passed as in arguments must be initialized before being passed in a method call.
      4. Whereas ref arguments may be modified, out arguments must be modified by the called method

int readonlyArgument = 44; 

InArgExample(readonlyArgument);

Console.WriteLine(readonlyArgument);     // value is still 44 

void InArgExample(in int number)   
{ 

   // Uncomment the following line to see error CS8331

    //number = 19;   
}

* + 1. Although in, out, and ref parameter modifiers are considered part of a signature, members declared in a single type cannot differ in signature solely by in, ref and out. Therefore, methods cannot be overloaded if the only difference is that one method takes a ref or out argument and the other takes an in argument. The following code, for example, will not compile:

class CS0663\_Example   
{

 // Compiler error CS0663: "Cannot define overloaded 

 // methods that differ only on in, ref and out". 

    public void SampleMethod(in int i) { }

  publicvoid SampleMethod(ref int i) { } 

} 

Overloading based on the presence of in is allowed:

class InOverloads   
{ 

    public void SampleMethod(in int i) { }

  publicvoid SampleMethod(int i) { }

}

* 1. [ref](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref) - We are passing the address of memory location and both argument and parameter points to same memory location.
     1. Variables passed as in arguments must be initialized before being passed in a method call.
     2. To use a ref parameter, both the method definition and the calling method must explicitly use the ref keyword,

void Method(ref int refArgument)   
{

    refArgument = refArgument + 44;   
}   
 

int number = 1; 

Method(ref number); 

Console.WriteLine(number);   
// Output: 45

* 1. [out](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/out-parameter-modifier) - Any operation on the parameter is made on the argument. It is like the [ref](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref) keyword, except that ref requires that the variable be initialized before it is passed.

int initializeInMethod;

OutArgExample(out initializeInMethod);

Console.WriteLine(initializeInMethod);     // value is now 44 

void OutArgExample(out int number)   
{ 

   number = 44;   
}

* 1. Declaring a method with out arguments is a classic workaround to return multiple values. Beginning with C# 7.0, consider [value tuples](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples) for similar scenarios. The following example uses out to return three variables with a single method call. The third argument is assigned to null. This enables methods to return values optionally.

void Method(out int answer, out string message, out string stillNull) 

{ 

  answer = 44;

  message = "I've been returned";

    stillNull = null;   
} 

int argNumber; 

string argMessage,

argDefault;

Method(out argNumber, out argMessage, out argDefault);

Console.WriteLine(argNumber); 

Console.WriteLine(argMessage);

Console.WriteLine(argDefault == null); 

// The example displays the following output:   
//      44   
//      I've been returned   
//      True

* 1. a variable named number before it is passed to the [Int32.TryParse](https://docs.microsoft.com/en-us/dotnet/api/system.int32.tryparse#system-int32-tryparse(system-string-system-int32@)) method, which attempts to convert a string to a number

string numberAsString = "1640";

   
if (Int32.TryParse(numberAsString, out var number))

    Console.WriteLine($"Converted '{numberAsString}' to {number}");

else   
    Console.WriteLine($"Unable to convert '{numberAsString}'");

// The example displays the following output:   
//       Converted '1640' to 1640

* 1. Difference between ref and out:

Ref needs to be initialized before passed to method while out doesn't require any initialization before being passed in a method call. However, the called method is required to assign a value before the method returns.

1. Which class acts as a base class for all the data types in .net?

 Object Type is the ultimate base class for all data types.The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion

1. Difference between dynamic type and object type variable

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

1. Properties and fields

Properties expose fields. Fields should (almost always) be kept private to a class and accessed via get and set properties. Properties provide a level of abstraction allowing you to change the fields while not affecting the external way they are accessed by the things that use your class.

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

public class MyClass   
{   
    // this is a field.  It is private to your class and stores the actual data.   
    private string \_myField;   
    // this is a property. When accessed it uses the underlying field,

    // but only exposes the contract, which will not be affected by the underlying field

public string MyProperty   
{   
get{   
   return \_myField;   
   }   
set{   
   \_myField = value;   
   }}

}}

1. **Constructor Overloading: C# supports overloading of constructors, that means we can have constructors with different set of parameters. So our class can be like the following code snippet:**   
   **public** **class** mySampleClass

{

**public** mySampleClass()

    {

    // no parameter constructor method.// First Constructor

    }

**public** mySampleClass(**int** Age)

    {

    // constructor with one parameter.// Second Constructor

    }

**public** mySampleClass(**int** Age, **string** Name)

    {

    // constructor with two parameters.// Third Constructor

    }

}

Well, note here that call to the constructor now depends on the way you instantiate the object.   
For example:   
mySampleClass obj = **new** mySampleClass()

// At this time the code of no parameter // constructor (First Constructor) will be executed

mySampleClass obj = **new** mySampleClass(12)

// At this time the code of one parameter // constructor(Second Constructor) will be // executed.

Consructor chaining:

**public** **class** mySampleClass

{

**public** mySampleClass(): **this**(10)

    {

    // No parameter constructor method.// First Constructor

    }

**public** mySampleClass(**int** Age)

    {

    // Constructor with one parameter.// Second Constructor}

    }

}

First of all let us see what is this syntax:

*public mySampleClass(): this(10)*

Here this refers to the same class, so when we say this(10), we actually mean execute the *public SampleClass (int Age)*method. The above way of calling the method is called initializer. We can have at the most one initialize in this way in the method.   
   
Another thing which we must know is the execution sequence i.e., which method will be executed when. Here if I instantiate the object as:

mySampleClass obj = **new** mySampleClass()

Then the code of *public mySampleClass(int Age)*will be executed before the code of mySampleClass(). So practically the definition of the method:   
**public** mySampleClass(): **this**(10)

{

   // This is the no parameter constructor method.// First Constructor

}

is equivalent to:   
**public** mySampleClass()

{

    mySampleClass(10)

     // This is the no parameter constructor method.// First Constructor

}

This is sometimes called Constructor chaining.

Oops concept –

* Inheritance

It is to inherit one class into another

* Abstraction

This provides essential features without describing any background details. Abstraction is important because it can hide unnecessary details from reference objects to names. It is also necessary for the construction of programs. Instead of showing how an object is represented or how it works, it focuses on what an object does. Therefore, data abstraction is often used for managing large and complex programs.

Abstraction allows the programmer to display only the necessary details to the world while hiding the others.

Abstraction is achieved in C# by using the Abstract class and interface.

**Example:**

|  |
| --- |
| **class** Program      {  **static** **void** Main(**string**[] args)             {                    Hyundai hyn = **new** Hyundai();                    String descp = hyn.Describe();                    Console.WriteLine(descp);                    Console.ReadKey();              }        }  **abstract** **class** Car   {  **public** **void** Describe()            {  **return** "Description of the car";             }     }    **class** Hyundai : Car  {  } |

**The output of the following is:**

Description of the car

|  |
| --- |
| **class** Program  {  **static** **void** Main(**string**[] args)             {                  Hyundai hyn = **new** Hyundai();                  String descp = hyn.Describe();                  Console.WriteLine(descp);                  Console.ReadKey();              }   }  **abstract** **class** Car  {  **public** **virtual** **string** Describe()                {  **return** "Description of the car";                 }     }  **class** Hyundai : Car  {  **public** **override** **string** Describe()           {  **return** "Description of the car is now Hyundai";            }   } |

**So, if you execute this code now, it will give the following output:**

Description of the car is now Hyundai

The derived class method overrides the base class method. In this way, you can create a different derived class such as Ferrari, Porsche, BMW, etc. with their own implementation methods.

If you look closely, you can see that our description method in the abstract class doesn’t contain any implementation.

***Then, why are we defining empty methods?***

This is because an Abstract class provides a signature of the methods and makes it an obligation for the subclasses to create an implementation for all of these methods. This allows sharing of the base class but at the same time, it also keeps a check on the method implementation of the derived class.

Interface

In C#, the interface is the blueprint of a class. The interface is similar to an abstract class and is used to achieve a hundred percent abstraction. All the methods described inside the interface are abstract by default. It doesn’t have any method body and it cannot be instantiated.

The interface is mainly used to achieve multiple inheritance and full abstraction. All the method signature declared inside the interface should be provided with implementation from the class or the struct that implements it.

**Example:**

|  |
| --- |
| **class** Program  {  **static** **void** Main(**string**[] args)            {                   Hyundai hyn = **new** Hyundai();                   String descp = hyn.Describe();                   Console.WriteLine(descp);                   Console.ReadKey();              }  }  **interface** Car   {  **string** Describe();      }  **class** Hyundai : Car    {  **public** **string** Describe()           {  **return** "Description of the car is now Hyundai";             }    } |

**The output of the above code will be:**

Description of the car is now Hyundai

* Encapsulation

This binds the member function and data member into a single class. This also allows for abstraction. Within OOP, encapsulation can be achieved through creating classes. Those classes then display public methods and properties. The name encapsulation comes from the fact that this class encapsulates the set of methods, properties, and attributes of its functionalities to other classes.

Encapsulation is an object-oriented programming concept that allows programmers to wrap data and code snippets inside an enclosure. By using the encapsulation program, you can hide the members of one class from another class. It’s like encircling a logical item within a package. It allows only relevant information available and visible outside and that too only to specific members.

Encapsulation is implemented by using access specifiers. Access Specifier is used for defining the visibility and accessibility of the class member in C#.

**C# contains the following access specifiers.**

* Public
* Private
* Protected
* Internal

The access specifiers define the visibility of the class and its values. It allows you to make data visible for a particular part of the code and hide it from another part. The most commonly used visibility is public and private.

* Polymorphism

**Polymorphism is basically divided into two parts:**

1. Compile-time Polymorphism
2. Run time polymorphism

*#1) Static or Compile Time Polymorphism*

Compile-time polymorphism is also known as Static polymorphism. the method calling decision is made at the time of compiling.

It is achieved by keeping the method name the same but passing different sets of parameters.

**Example:**

|  |
| --- |
| **class** Program {  **void** print(**int** i, **int** j) {  Console.WriteLine("Printing int: {0}", (i+j) );  }  **void** print(**string** a, **string** b) {  Console.WriteLine("Printing String: " , (a+b));  }    **static** **void** Main(**string**[] args) {  Program prog = **new** Program();    // Call print for sum of integers  prog.print(5, 6);    // Call to concatenate strings  prog.print("Hello","World");  Console.ReadKey();  }  } |

In the above example, we have called the same method “print” twice using different parameters. At first, we pass two integers as parameters, then we have passed two strings as parameters. There are two “print” methods with the same name.

When we pass a parameter with the integers system, it will look for the method named “print” which accepts two integer parameters and it will execute that ignoring other methods with the same name.

In the second part, we passed the string parameter. Again the system will look for the method that accepts two string parameters. Hence, based on the parameters passed, the first method will add two integers and the next one will concatenate two strings.

*#2) Dynamic Polymorphism or Runtime Polymorphism*

Runtime polymorphism or dynamic polymorphism occurs when both method name and method signature have the same name and parameters. It takes decision during runtime.

**Example:**

|  |
| --- |
| **class** Program   {  **public** **void** print()           {                 Console.WriteLine("Printing from class Program");            }   }  **class** Execute : Program   {  **public** **void** print()           {                   Console.WriteLine("Printing from class Execute");           }    **public** **static** **void** Main(**string**[] args)                  {                          Execute exe = **new** Execute();                           exe.print();                          Console.ReadLine();                  }    } |

**If we run the above program we will get the following output:**

Printing from class Execute

Even though the class Execute has inherited all the methods of the class Program but when we called the method print which is present in both the classes, the method present in the child class will override the method from the parent class.

Dynamic polymorphism is used to implement abstraction. It allows the user to create an abstract class that is used to provide an implementation for an interface when it is inherited by a derived class. The abstract class can contain names/signature of the methods and the derived class can have a more specialized definition for the method.

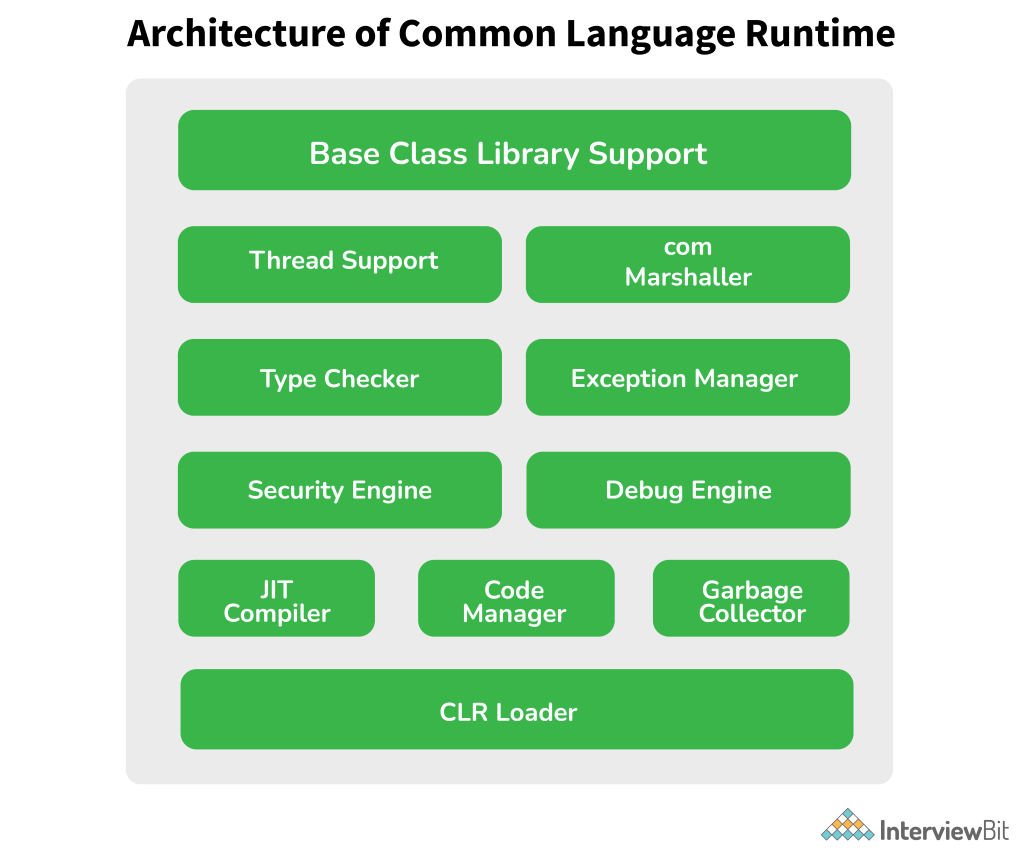
.Net concept –

1. What is a managed and unmanaged code?

Managed code lets you run the code on a managed CLR runtime environment in the .NET framework.    
Benefits: Provides various services like a garbage collector, exception handling, etc.    
   
Unmanaged code is when the code doesn’t run on CLR, it is an unmanaged code that works outside the .NET framework. Such an example is C++.

1. What is Common Language Runtime (CLR)?

CLR handles program execution for various languages including C#. The architecture of CLR handles memory management, garbage collection, security handling, and looks like:



1. .Net core and .Net framework

What is .NET Core?

.NET Core is the latest version of Microsoft’s .NET Framework, which is a free, open-source, general-purpose programming platform. It’s a**cross-platform framework that works with Windows, Mac OS X, and Linux.**The.NET Core Framework may be used to create a variety of apps, including mobile, desktop, online, cloud, IoT, machine learning, microservices, games, and more. .NET Core is written from scratch to make it a modular, lightweight, fast, and cross-platform Framework.

What is .NET Framework?

The framework is a re-usable design platform for software systems, which provides support for code libraries and various scripting languages In simple words a framework is something that makes core programming easy. Microsoft’s Net Framework is a software development platform for building and running Windows applications. Net framework includes developer tools, programming languages, and libraries that are used to develop desktop and web applications. It is also used to build websites, web services, and games.

Key Differences

|  |  |  |
| --- | --- | --- |
| Basis | .NET Core | .NET Framework |
| Open-Source | .NET Core is an open-source platform. | The.Net Framework includes certain open source components. |
| Cross-Platform | It is based on the concept of “create once, run anywhere.” Because it is cross-platform, it is compatible with a variety of operating systems, including Windows, Linux, and Mac OS. | .NET Framework is compatible with Windows OS(operating system) only |
| Application models | The Application Model of .Net Core includes ASP.NET and windows universal apps. | The Application Model of the .NET Framework includes WinForms, ASP.NET, and WPF. |
| Microservices support | .NET Core has support for microservices. , NET Core allows a mix of technologies that can be minimalized for each microservice. | When we talk about the .NET Framework it does not allow for the construction and deployment of microservices in multiple languages. |
| REST services support | .NET Core has no support for WCF( Windows Communication Foundation ) services. You would always need to create a REST API. | When it comes to WCF (Windows Communication Foundation) services, the.NET Framework is a fantastic choice. It also works with RESTful services. |
| Performance and Scalability | .NET core provides high scalability and performance in comparison to .NET Framework because of its architecture. | .NET Framework is less scalable and provides low performance in comparison to .NET Core. |
| Compatibility | Mobile.NET Core is compatible with various operating systems-Windows, Linux, and Mac OS. | On the other .NET Framework is only compatible with Windows OS. |
| Mobile Development | Mobile apps are becoming more important for businesses. .NET Core has some support for mobile apps. It’s compatible with Xamarin and other open-source platforms for mobile applications. | On the other hand, the .NET Framework does not support their development at all, and that is a problem. |

NET Core Vs .NET Framework – Which is Better?

The answer to this question depends on the project requirement, what our project demands for so here are some points that we have to consider to choose best for our project out of the .NET Framework and .NET Core.

Prefer or choose .NET Core if

1. The project demands cross-platform integration.
2. The project requires the development of microservices.
3. Project relies heavily on CLI( Command Line Interface) as .NET Core is suitable for CLI.

Prefer or choose .NET Framework if

1. Applications are already running on .NET Framework.
2. The Applications require technologies like workflow, webforms, or WCF that are not present in .NET Core.
3. Applications are built to run on Windows alone.

**Is .NET core replacing the .NET framework?**

The answer to this question is no as the .NET Core and .NET Framework both have their cons and pros that we discussed earlier and we should choose between them according to the project needs.

Why to use .net core 3.1?

It will support for long time that is for atleast 3 years.

**Why  .Net core is faster than the .NET framework?**

.NET Core is faster than .NET Framework because the architecture of .NET Core is written or restructured from scratch to make it a modular, lightweight, fast, and cross-platform Framework.

**When Not to Use .NET Core**

You should not choose or use .NET Core in the following scenarios

1. The Applications require technologies like workflow, webforms or WCF that are not present in .NET Core.
2. Applications are built to run on Windows alone.
3. Do not want to deal with continuous upgrades and changes.

**When Not to Run .NET Framework**

You should not use or run .NET Framework in the following cases

1. The project demands cross-platform integration.
2. Project requires the development of microservices.
3. Project relies heavily on CLI( Command Line Interface) as .NET Core is suitable for CLI.

1. **SDLC – Software development life cycle:**

Chart, bubble chart

Description automatically generated

Why SDLC?

The following are some reasons why SDLC is important in software development:

* The goal is to deliver high-quality software that meets our customers’ expectations.
* Each stage/phase of the process is followed by a formal review that ensures optimum management control.
* Ensures that all stakeholders in the development process are aware of the project planning process.
* With SDLC, you can generate considerable documentation about your system.
* Several intermediate products are produced, all of which can be reviewed in order to determine if they meet the specifications and the needs of the users.
* In every phase, there are specific deliverables, entry criteria, and exit criteria.
* A sequential development process is ideal for small- to mid-sized projects with clear requirements.
* It allows you to lower project risk and management plan overhead.
* A faster and more efficient development process.

**STLC:**

Diagram

Description automatically generated

Redux –

1. **What is redux?**

Redux is an open-source librarary

primary use in managing and centralizing application state. It is a predictable state container for applications built using JavaScript.

used along with JavaScript libraries, for instance, React or Angular in order to build UIs (User Interfaces).

1. What is flux?

*Flux* is a design pattern which used as a replacement for the more traditional MVC pattern. It is not a framework or a library but a new kind of architecture that complements React and the concept of Unidirectional Data Flow.

The workflow between dispatcher, stores and views components with distinct inputs and outputs as follows:

Diagram

Description automatically generated

1. **Difference between Redux and Flux.**

The main differences in the comparison: Redux vs Flux are as follows:

|  |  |  |
| --- | --- | --- |
| COMPARISON PARAMETER | REDUX | FLUX |
| Number of stores per application | Redux includes a single Store per application. Rather than placing state information in multiple Stores across the app, Redux keeps everything in one region of the application | Flux includes multiple Stores per application. |
| Architecture | Redux is an open-source JavaScript library used for creating User Interfaces. | Flux's architecture is designed to build client-side web apps. |
| Place where Business Logic of the Application Resides | In Redux, the business logic of the application resides in the Reducer. | In Flux, the business logic of the application resides in the Store. |

1. **differences between mapStateToProps() and mapDispatchToProps()**

|  |  |
| --- | --- |
| mapStateToProps() | mapDispatchToProps() |
| The mapStateToProps() method is used to render the stored data to the component. | The mapDispatchToProps() method is used to render the action creators with props to the component. |
| The entirety of the results of the mapStateToProps() method is a plain object which is later merged into the component’s prop. | In the mapDispatchToProps() method, each action creator is wrapped in the dispatcher call so that they can be called upon directly and later merged into the component’s prop. |
| This method's use is to connect the redux state to the props of the react component. | This method's use is to connect redux actions to the react props. |

1. **Constants  in Redux.**

First of all, we can store all the constants in a single file in our project named constants.js or something else as follows:

export const ADDING\_TODO = 'ADDING\_TODO';   
export const DELETING\_TODO = 'DELETING\_TODO';   
export const EDITING\_TODO = 'EDITING\_TODO';   
export const COMPLETING\_TODO = 'COMPLETING\_TODO';   
export const COMPLETING\_ALL = 'COMPLETING\_ALL';   
export const CLEARING\_COMPLETED = 'CLEARING\_COMPLETED';

After storing the constants in one place, we can use them in two ways in our project:

During actions creation (in actions.js file of our project):

 import { DELETING\_TODO } from './constants';   
   
export function deletingTodo(text) {   
  return { type: DELETING\_TODO, text };   
}

In Reducers (in reducer.js file of our project):

 import { EDITING\_TODO } from './constants';   
   
export default (state = [], action) => {   
  switch (action.type) {   
    case EDITING\_TODO:   
      return [   
        ...state,   
        {   
          text: action.text,   
          completed: false   
        }   
      ];   
    default:   
      return state   
  }   
};

1. **What are reducers in Redux's architecture?**

Reducers in Redux's architecture are **pure functions**that are used to take the previous state and an action and return the next state. Its syntax is given below:

 (previous\_state, action) => new\_state

const initial\_state = { value: 0 }

function countReducer(state = initial\_state, action)

{ // Checking to see if the reducer cares about this action

if (action.type === 'counter/incremented') {

// If the action is of type "counter" or "incremented", we make a copy of `state`

 return { ...state,

// We also update the copy with the new value

value: state.value + 1 } }

// If not, we return the original state unchanged return state }

1. What is action?

*Actions* are plain JavaScript objects or payloads of information that send data from your application to your store. They are the only source of information for the store. Actions must have a type property that indicates the type of action being performed.

For example, let's take an action which represents adding a new todo item:

{   
type: ADD\_TODO,   
text: 'Add todo item'   
}

1. Explain the typical data flow in an application made using React and Redux (Redux Lifecycle for an application).

The typical data flow in Redux starts with a call back from the User Interface component which dispatches an action with a payload. After that, the reducers intercept and receive the dispatched actions, generating a new application state. After that, the actions are propagated down through a hierarchy of components from the Redux store.

1. A picture containing text, sign

   Description automatically generated
2. What are the core principles of Redux?

Redux follows three fundamental principles:

1. **Single source of truth:** The state of your whole application is stored in an object tree within a single store. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.
2. **State is read-only:** The only way to change the state is to emit an action, an object describing what happened. This ensures that neither the views nor the network callbacks will ever write directly to the state.
3. **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write reducers. Reducers are just pure functions that take the previous state and an action as parameters, and return the next state.
4. How to access Redux store outside a component?

You just need to export the store from the module where it created with createStore(). Also, it shouldn't pollute the global window object.

store = createStore(myReducer)   
 

export default store

1. Can I dispatch an action in reducer?

Dispatching an action within a reducer is an **anti-pattern**. Your reducer should be *without side effects*, simply digesting the action payload and returning a new state object. Adding listeners and dispatching actions within the reducer can lead to chained actions and other side effects.

1. How to dispatch an action on load?

You can dispatch an action in componentDidMount() method and in render() method you can verify the data.

class App extends Component { 

componentDidMount() { 

  this.props.fetchData()   
}

   
render() { 

return this.props.isLoaded   
?

 <div>{'Loaded'}</div>

: <div>{'Not Loaded'}</div>   
}   
}   
 

const mapStateToProps = (state) => ({ 

  isLoaded: state.isLoaded   
})   
 

const mapDispatchToProps = { fetchData }

   
export default connect(mapStateToProps, mapDispatchToProps)(App)

1. How to reset state in Redux?

You need to write a *root reducer* in your application which delegate handling the action to the reducer generated by combineReducers().

1. For example, let us take rootReducer() to return the initial state after USER\_LOGOUT action. As we know, reducers are supposed to return the initial state when they are called with undefined as the first argument, no matter the action.

const appReducer = combineReducers({   
/\* your app's top-level reducers \*/   
})   
 

const rootReducer = (state, action) => {

if (action.type === 'USER\_LOGOUT') {

state = undefined   
} 

return appReducer(state, action)   
}

1. What is Redux Thunk?

*Redux Thunk* middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods dispatch() and getState() as parameters.

1. How to set initial state in Redux?

You need to pass initial state as second argument to createStore:

const rootReducer = combineReducers({

todos: todos,   
visibilityFilter: visibilityFilter   
}) 

const initialState = {   
todos: [{ id: 123, name: 'example', completed: false }]   
}   
 

const store = createStore(   
rootReducer,   
initialState   
)



React –

1. ReactJs:

React is an **open-source frontend JavaScript library** which is used for building user interfaces especially for single page applications.

1. SPA (Single-page application):

An SPA is a web app implementation that loads only a single web document, and then updates the body content of that single document via JavaScript APIs such This therefore allows users to use websites without loading whole new pages from the server,  SPAs don't need to load each new web page from scratch

1. Performance gains

Disadvantages:

 more effort required to maintain state, implement navigation

Multi-Page Application

Multiple-page applications work in a “traditional” way. Every change eg. display the data or submit data back to server requests rendering a new page from the server in the browser.

1. How react work?
   1. Creates virtual DOM in memory.
   2. Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM. React finds out what changes have been made, and changes **only** what needs to be changed.

1. When to use a Class Component over a Function Component?

If the component needs *state or lifecycle methods* then use class component otherwise use function component. *However, from React 16.8 with the addition of Hooks, you could use state , lifecycle  methods and other features that were only available in class component right in your function component.* \*So, it is always recommended to use Function components, unless you need a React functionality whose Function component equivalent is not present yet, like Error Boundaries \*

1. **Functional Components vs Class Components:**
   1. **Functional component:** These are nothing but simple javascript functions

const Car=()=> {return <h2>Hi</h2> }

* 1. **Class components:** These components are just like simple class and must have render function.

class Car extends React.Component {

  render() {    
     return <h2>Hi, I am a Car!</h2>;    } }

|  |  |
| --- | --- |
| **Functional Components** | **Class Components** |
| A functional component is just a plain JavaScript function that accepts props as an argument and returns a React element. | A class component requires you to extend from React Component and create a render function which returns a React element. |
| There is no render method used in functional components. | It must have the render() method returning HTML |
| Also known as Stateless components as they simply accept data and display them in some form, that they are mainly responsible for rendering UI. | Also known as Stateful components because they implement logic and state. |
| React lifecycle methods (for example, componentDidMount) cannot be used in functional components. | React lifecycle methods can be used inside class components (for example, componentDidMount). |
| Hooks can be easily used in functional components.  example: const [name,SetName]= React.useState(‘ ‘) | It requires different syntax inside a class component to implement hooks.  example: constructor(props) {     super(props);     this.state = {name: ‘ ‘}  } |
| Constructors are not used . | Constructor are used as it needs to store state. |



|  |  |
| --- | --- |
| **Real DOM** | **Virtual  DOM** |
| 1. It updates slow. | 1. It updates faster. |
| 2. Can directly update HTML. | 2. Can’t directly update HTML. |
| 3. Creates a new DOM if element updates. | 3. Updates the JSX if element updates. |
| 4. DOM manipulation is very expensive. | 4. DOM manipulation is very easy. |
| 5. Too much of memory wastage. | 5. No memory wastage. |

1. Advantages of React:
   1. Uses virtual DOM which improves performance
   2. Reusable component:  Components are independent and reusable bits of code. These components can be shared across various applications having similar functionality. The re-use of components increases the pace of development.
2. React Components re-renders in the following scenarios:
   1. “setState” is called in Component
   2. “props” values are updated
   3. This.forceUpdate() is called
3. How to inherit class?

è  By writing extends keyword , class Model extends Carè  By calling the super() method in the constructor method, we call the parent's constructor method and gets access to the parent's properties and methods.

1. Arrow function: Arrow functions allow us to write shorter function syntax

Before: hello = function() {  return "Hello World!";}

After: hello = () => {  return "Hello World!";}

With arrow function, no binding required while with normal function, we need to bind function with this keyword       This.hello=this.hello.bind(this)

|  |  |
| --- | --- |
| **Stateful Component** | **Stateless Component** |
| 1. Stores info about component’s state change in memory | 1. Calculates the internal state of the components |
| 2. Have authority to change state | 2. Do not have the authority to change state |
| 3. Contains the knowledge of past, current and possible future changes in state | 3. Contains no knowledge of past, current and possible future state changes |
| 4. Stateless components notify them about the requirement of the state change, then they send down the props to them. | 4. They receive the props from the Stateful components and treat them as callback functions. |

1. What is JSX?
   1. JSX stands for JavaScript XML.
   2. It allows us to write HTML inside JavaScript and place them in the DOM without using functions like appendChild( ) or createElement( ).
2. How to prevent re-renders in React?

To prevent the re-rendering of child component, we use the shouldComponentUpdate( ) method:    
**\*\*Note-**Use shouldComponentUpdate( ) method only when you are sure that it’s a **static**component.

shouldComponentUpdate() {

 return false;

}

1. What is the significance of keys in React?

A key is a special string attribute you **should** include when creating arrays of elements. *Key* prop helps React identify which items have changed, are added, or are removed. Keys are used for identifying unique Virtual DOM Elements with their corresponding data driving the UI. They help React to optimize the rendering by recycling all the existing elements in the DOM. These keys must be a unique number or string, using which React just reorders the elements instead of re-rendering them. This leads to increase in application’s performance. Better not to use index as key because anytime array can get re-order

1. Using *indexes* for *keys* is **not recommended** if the order of items may change. This can negatively impact performance and may cause issues with component state.

1. What is variable?

there are three ways of defining your variables: var, let, and const

1. Var a=5;
   1. If you use var outside of a function, it belongs to the global scope.
   2. If you use var inside of a function, it belongs to that function.
   3. If you use var inside of a block, i.e. a for loop, the variable is still available outside of that block.
   4. var has a ***function* scope, not a *block* scope.**
2. Let a=5;
   1. let has a ***block* scope**
   2. let is limited to the block (or expression) where it is defined.
   3. If you use let inside of a block, i.e. a for loop, the variable is only available inside of that loop.
3. Const a=5;    
   a. const is a variable that once it has been created, its value can never change.    
   b. const has a ***block* scope.**

1. Shallow Comparison :

The values saved in the variable can be either a primitive or reference type. Example **“var a = 10”,**here the value saved in the variable “a” is of primitive type.

 The data stored in Objects and Array can be referred to as Reference type data.**Comparing Primitive Values is not a concern, problems arise when we have reference values during the comparison.**

When we compare two different objects with the same properties, they equate to false. JavaScript looks for the object reference (Starting Address of the Object). Since the references are different, then even if the property values are the same, it results in “false” value. We can see the same in the code below.

1. var initialData = {    
      name: "Mayank Gupta",    
      age: 30    
   }
2. var finalData = {    
      name: "Mayank Gupta",    
      age: 30    
   }

O/P - False 

we have two objects, *userInfo* and *cloneData*. We copy the value of userInfo in cloneData object. Both these variables are now pointing to the same object, since Objects are copied by reference. Updating any of the objects, update the other object as well, since they are referring to the same object. On comparison for equality, it returns true. So for any object, JavaScript look for the base address of the objects being referred.

var userInfo = {    
   name: "Mayank Gupta",    
   age: 30, designation: "Developer"    
}  

var cloneData = userInfo;

Output: true

If we want to copy the object into a new object, we can use the spread operator to create a separate reference for the object.

var userInfo = {    
   name: "Mayank Gupta",    
   age: 30,    
   designation: "Developer"    
}    
var cloneData = { ...userInfo };

Output: false

1. **Why should we not update the state directly?**

If state is updated directly, then it won't re-render the component.

//Wrong    
this.state.message = 'Hello world'

//Correct    
this.setState({ message: 'Hello World' })

1. What is the difference between HTML and React event handling?
   1. In HTML, the event name usually represents in *lowercase* as a convention:

Whereas in React it follows *camelCase* convention:

* 1. In HTML, you can return false to prevent default behavior:

<a href='#' onclick='console.log("The link was clicked.");

 return false;' />

Whereas in React you must call preventDefault() explicitly:

function handleClick(event) {   
  event.preventDefault()   
  console.log('The link was clicked.')   
}

1. What is the use of refs?

The *ref* is used to return a reference to the element. They *should be avoided* in most cases, however, they can be useful when you need a direct access to the DOM element or an instance of a component.

class MyComponent extends React.Component { 

  constructor(props) { 

super(props)   
    this.myRef = React.createRef()   
  } 

  render() {

  return <div ref={this.myRef} />   
  }}

1. Which is preferred option with in callback refs and findDOMNode()?

It is preferred to use *callback refs* over findDOMNode() API. The **legacy** approach of using findDOMNode

1. What is Lifting State Up in React?

When several components need to share the same changing data then it is recommended to *lift the shared state up* to their closest common ancestor. That means if two child components share the same data from its parent, then move the state to parent instead of maintaining local state in both of the child components.

1. What is reconciliation?

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. This process is called *reconciliation*.

1. How to set state with a dynamic key name?

handleInputChange(event) {   
  this.setState({ [event.target.id]: event.target.value })   
}

1. How to enable production mode in React?

 use Webpack's DefinePlugin method to set NODE\_ENV to production, by which it strip out things like propType validation and extra warnings.

1. What is the recommended way for naming components?

It is recommended to name the component by reference instead of using displayName.

Using displayName for naming component:

export default React.createClass({ 

  displayName: 'TodoApp',   
  // ...   
})

The **recommended** approach:

export default class TodoApp extends React.Component {   
  // ...   
}

also

const TodoApp = () => {   
//...   
}   
export default TodoApp;

1. What is strict mode in React?

React.StrictMode is a useful component for highlighting potential problems in an application. Just like <Fragment>, <StrictMode> does not render any extra DOM elements. It activates additional checks and warnings for its descendants. These checks apply for *development mode* only.

import React from 'react'   
   
function ExampleApplication() { 

  return (   
    <div> 

      <Header />   
      <React.StrictMode>

        <div> 

          <ComponentOne /> 

<ComponentTwo />   
        </div>

      </React.StrictMode>

      <Header />   
    </div>   
  )   
}

1. Why should component names start with capital letter?

Because react throws error. You can define component class which name starts with lowercase letter, but when it's imported it should have capital letter. Here lowercase is fine:

class myComponent extends Component { 

  render() {   
    return <div />   
  }   
}   
   
export default myComponent

While when imported in another file it should start with capital letter:

import MyComponent from './MyComponent'

1. What is the difference between super() and super(props) in React using ES6 classes?

When you want to access this.props in constructor() then you should pass props to super() method.

**Using super(props):**

class MyComponent extends React.Component { 

  constructor(props) {   
    super(props)   
    console.log(this.props) // { name: 'John', ... }   
  }   
}

**Using super():**

class MyComponent extends React.Component { 

  constructor(props) {   
    super()   
    console.log(this.props) // undefined   
  }   
}

Outside constructor() both will display same value for this.props.

1. How to re-render the view when the browser is resized?

You can listen to the resize event in componentDidMount() and then update the dimensions (width and height). You should remove the listener in componentWillUnmount() method.

class WindowDimensions extends React.Component {   
  constructor(props){   
    super(props);   
    this.updateDimensions = this.updateDimensions.bind(this);   
  } 

  componentWillMount() {   
    this.updateDimensions()   
  }

   
  componentDidMount() {   
    window.addEventListener('resize', this.updateDimensions)   
  }   
   
  componentWillUnmount() {   
    window.removeEventListener('resize',this.updateDimensions)}

  updateDimensions(){this.setState({width:window.innerWidth,height:window.innerHeight})   
  }   
   
  render() {   
    return <span>{this.state.width} x {this.state.height}</span>   
  }   
}

1. What is the difference between setState() and replaceState() methods?

When you use setState() the current and previous states are merged. replaceState() throws out the current state, and replaces it with only what you provide. Usually setState() is used unless you really need to remove all previous keys for some reason. You can also set state to false/null in setState() instead of using replaceState().

1. Why you can't update props in React?

The React philosophy is that props should be *immutable* and *top-down*. This means that a parent can send any prop values to a child, but the child can't modify received props.

1. How to define constants in React?

You can use ES7 static field to define constant.

class MyComponent extends React.Component { 

  static DEFAULT\_PAGINATION = 10   
}

1. What is React Router?

**React Router** is a standard library for routing in React. It enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL.

* **BrowserRouter:** It is the parent component that is used to store all of the other components and can have only single child
* **Routes:** It’s a new component introduced in the v6 and a upgrade of the component. The main advantages of Routes over Switch are:
  + Routes are chosen based on the best match instead of being traversed in order and pick first match.
* **Route:** Route is the conditionally shown component that renders some UI when its path matches the current URL.

Case1: If you use exact and strict together, then the location.pathname will only match exactly as provided in path props.

Example: <Route exact strict path="/one/" component={About}/> 

Will only match /one/ but not /one and /one/two.

case 2: If you use only strict, then the location.pathname will match which have trailing slash.

Example:

<Route strict path="/one/" component={About}/> 

Will match /one/ and /one/two but not /one.

case 3: If you use only exact, then the location.pathname will match exact location path.

Example:

<Route exact path="/one" component={About}/> 

Will match /one or /one/. The exact props doesn't care for trailing slash. But it will not match /one/two.

**2. path:** Path specifies a pathname we assign to our component.

**3. element:** It refers to the component which will render on matching the path.

* **Link:** Link component is used to create links to different routes and implement navigation around the application. It works like HTML [anchor tag](https://www.geeksforgeeks.org/html-a-tag/).

Link component uses the **to** prop to describe the location where the links should navigate to.

|  |
| --- |
| <li>     <Link to="/">Home</Link>  </li> |

* The <NavLink> is a special type of <Link> that can style itself as “active” when its to prop matches the current location.

<NavLink to="/react" activeClassName="hurray">   
React   
</NavLink>

1. What is the purpose of push() and replace() methods of history?

A history instance has two methods for navigation purpose.

1. push()
2. replace()

If you think of the history as an array of visited locations, push() will add a new location to the array and replace() will replace the current location in the array with the new one.

1. How to get query parameters in React Router v4?

The ability to parse query strings was taken out of React Router v4 because there have been user requests over the years to support different implementation. So the decision has been given to users to choose the implementation they like. The recommended approach is to use query strings library.

const queryString = require('query-string');   
const parsed = queryString.parse(props.location.search);

You can also use URLSearchParams if you want something native:

const params = new URLSearchParams(props.location.search)   
const foo = params.get('name')

1. Why you get "Router may have only one child element" warning?

You have to wrap your Route's in a <Switch> block because <Switch> is unique in that it renders a route exclusively. At first you need to add Switch to your imports:

import { Switch, Router, Route } from 'react-router'

Then define the routes within <Switch> block:

<Router>

  <Switch>   
    <Route {/\* ... \*/} />   
    <Route {/\* ... \*/} />   
  </Switch> 

</Router>

1. How to pass params to history.push method in React Router v4?

While navigating you can pass props to the history object:

this.props.history.push({   
  pathname: '/template',   
  search: '?name=sudheer',   
  state: { detail: response.data }   
})

The search property is used to pass query params in push() method.

1. How to implement *default* or *NotFound* page?

A <Switch> renders the first child <Route> that matches. A <Route> with no path always matches. So you just need to simply drop path attribute as below

<Switch>   
  <Route exact path="/" component={Home}/>   
  <Route path="/user" component={User}/>   
  <Route component={NotFound} />   
</Switch>

1. How to perform automatic redirect after login?

The react-router package provides <Redirect> component in React Router. Rendering a <Redirect> will navigate to a new location.

import React, { Component } from 'react'

import { Redirect } from 'react-router'   
export default class LoginComponent extends Component {   
  render() {   
    if (this.state.isLoggedIn === true) {

     return <Redirect to="/your/redirect/page" />   
    }

 else {   
      return <div>{'Login Please'}</div>   
    }   
  }   
}

1. What’s the difference between forceUpdate vs setState

 setState() re-render the component if some state or props of that component is changed. When we call setState() the lifecycle method shouldComponentUpdate() method calls automatically that decide if the component should re-render or not. The shouldComponentUpdate() method exit the update life cycle if there is no reason for re-render.

Whereas the forceUpdate() method re-render the component without even changing the state or props. When we call forceUpdate(), it will re-render the component and skip shouldComponentUpdate() method.

|  |  |
| --- | --- |
| **forceUpdate Method** | **setState Method** |
| It re-render the component without even changing the state or props. | It re-render the component if some state or props of that component changed. |
| It skips the lifecycle shouldComponentUpdate method. | It calls the lifecycle shouldComponentUpdate method. |

1. ES6 features of Javascript

ES6 is the version of JavaScript and there are a lot of features of ES6. Few features are - *Arrow Functions, Let and Const, Class*,*and ‘this’* keyword.

[**Arrow Function**](https://www.geeksforgeeks.org/arrow-functions-in-javascript/)**:** Arrow function allows you to write the shorter syntax for function. It makes your code clean and more readable.

1. What is the purpose of registerServiceWorker in React?

React creates a service worker for you without any configuration by default. The service worker is a web API that helps you cache your assets and other files so that when the user is offline or on a slow network, he/she can still see results on the screen, as such, it helps you build a better user experience, that's what you should know about service worker for now. It's all about adding offline capabilities to your site.

  import React from 'react'; 

import ReactDOM from 'react-dom'; 

import App from'./App'; 

import registerServiceWorker from './registerServiceWorker'; 

   ReactDOM.render(<App />, document.getElementById('root'));

   registerServiceWorker();

1. Component lifecycle:
2. Mounting:
   1. constructor()

Before the start of the mounting phase, we may need to initialize our component using a constructor() method. This is used when we need to initialize state and bind methods to our component. This is the only place where this.state is explicitly assigned. (*Do not use setState here.*)

Again, two important things happen in this step:

* 1. An object is assigned to this.state
  2. Methods are passed/bound to the instance of the class through props
  3. GetDerivedStateFromProps():

The getDerivedStateFromProps() method is used when the state of a component depends on changes of props. getDerivedStateFromProps(props, state) is a static method that is called just before render() method in both mounting and updating phase in React. It takes updated props and the current state as arguments. We have to return an object to update state or null to indicate that nothing has changed.

static getDerivedStateFromProps(props, state) {

**if**(props.name !== state.name){

            //Change in props

**return**{

                name: props.name

            };

        }

**return** **null**; // No change to state

    }

/\* if props changes then after getDerivedStateFromProps

       method, state will look something like

    {

        name: props.name

    }

\*/

* 1. render():

The render() method is the only method that the component is ***required*** to have. It will always be called and its job is to mount the component to the DOM.

* 1. componentDidMount():

 This method will be immediately invoked after the render function is executed. We can make an API call and update the components state based on its response. We can fill in the content from data we load from another endpoint.

1. Updating:
   1. shouldComponentUpdate():

As the name suggests, this method gives you control over whether or not a component should update due to a change in its props or state. By default, a component will always re-render when updated. This method can either return a true or false. Also, this method receives nextProps and nextState as arguments so you can always compare it with the component’s current prop and state values. This method can help with performance optimization but it is also rarely used.

* 1. render():

If shouldComponentUpdate() returns true, the render function is invoked immediately.

* 1. getSnapshotBeforeUpdate():

In this method, we are given access to the props and state value before the update is committed to the DOM. Even though the render function was already called, we are still able to see the previous values. The use of this case is uncommon but can be used to capture any information we may need. It is important to note that this method works for hand in hand with componentDidUpdate(), which is the next lifecycle method that we will discuss.

* 1. componentDidUpdate():

This method is the last one invoked in this phase. Like the previous method, it also receives the former props and state values as arguments but it also receives the return value getSnapshotBeforeUpdate() as a third argument (if present).It is typically used to make more fetch requests on the condition of comparing the current and previous props and state values. Therefore, you may call setState but within a conditional statement.

1. Unmounting:
   1. componentWillUnmount()

This method is executed right before the component is unmounted from the DOM. You can think of this method as a way to clean up anything that is needed to be removed before the component is destroyed.

1. Error handling methods:

There two methods associated:

1. static getDerivedStateFromError():

If an error were to occur in the child component, this method is invoked. The error will be passed as an argument and this method would return a value that will update the state of the class component. This method is useful for rendering a custom fallback UI instead of rendering a broken component.

1. componentDidCatch():

This method receives the error as an argument as well as an object that has information about the error. This method is perfect for logging the error!

Timeline

Description automatically generated with medium confidence

1. What is Babel?

Babel is a JavaScript compiler

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Here are the main things Babel can do for you:

* Transform syntax
* Polyfill features that are missing in your target environment (through a third-party polyfill such as [core-js](https://github.com/zloirock/core-js))

1. Why you should add babel polyfill for every React application?

Because using babel alone does not guarantee that your code will run on all browsers. As you are aware, the latest ES6 features such as promises, arrow functions, etc may not be supported by older browsers, so we use Babel to convert the code to ES5 code which all browsers understand.

When we use create-react-app to create a React application, it includes Babel by default for the conversion of ES6 to ES5 code. But that is not sufficient to support all browsers.

So if your browser understands ES6, then your code will work fine, but for older browsers like Internet explore version 11 or lower, this code will not work.

Therefore, if your application is using promises directly or indirectly using axios or superagent library, then your application will not work on internet explorer or older browsers.

Let’s consider Array.prototype.includes method (added in ES7) which is used to check if a particular element exists in an array.

Graphical user interface, text, application

Description automatically generated

As you can see, for the Array.prototype.includes method, the ES5 code is the same as the input, so your application will not work on Internet Explorer if you are using this method anywhere in your application.

**So long story short**, just using babel is not enough for your application to work because all the latest Javascript features are not supported in all browsers. So to fix this problem, we need to use a polyfill.

1. React.lazy & React.Suspense:

React.lazy will automatically load the bundle containing the OtherComponent when this component is first rendered. The lazy component should then be rendered inside a Suspense component, which allows us to show some fallback content (such as a loading indicator) while we’re waiting for the lazy component to load.

 The React.lazy function lets you render a dynamic import as a regular component. React.lazy takes a function that must call a dynamic import(). This must return a Promise which resolves to a module with a default export containing a React component.

Before: import OtherComponent from './OtherComponent';

After: const OtherComponent = React.lazy(() => import('./OtherComponent'));

import React, { Suspense } from 'react';

   
const OtherComponent = React.lazy(() => import('./OtherComponent')); 

function MyComponent() {   
  return (

    <div>

<Suspense fallback={<div>Loading...</div>}>   
<OtherComponent />   
</Suspense> 

    </div>   
  );   
}

The fallback prop accepts any React elements that you want to render while waiting for the component to load. You can place the Suspense component anywhere above the lazy component. You can even wrap multiple lazy components with a single Suspense component.

      <Suspense fallback={<div>Loading...</div>}> 

        <section>   
          <OtherComponent /> 

          <AnotherComponent />

        </section>   
      </Suspense>

    </div>   
  );   
}

1. What is the use of lazy loading?

The benefits of lazy loading include: **Reduces initial load time** – Lazy loading a webpage reduces page weight, allowing for quicker page load time. Bandwidth conservation – Lazy loading conserves bandwidth by delivering content to users only if it's requested.

Graphical user interface, diagram

Description automatically generated

1. Does React router lazy load?

on demand using React. lazy() and dynamic import() . Using this technique, pages that are not required on the home page can be split out into separate bundles, thereby decreasing **load** time on the initial page and improving performance

const About = React.lazy(() => import("./pages/About"));

const Dashboard = React.lazy(() => import("./pages/Dashboard"));

export default function App() {

  return (

    <div>

    <Routes>

        <Route path="/" element={<Layout />}>

          <Route index element={<Home />} />

          <Route path="about" element={

              <React.Suspense fallback={<>...</>}>

                <About />

              </React.Suspense>

            }

          />

          <Route path="dashboard/\*" element={

              <React.Suspense fallback={<>...</>}>

                <Dashboard />

              </React.Suspense>

            }

          />

          <Route path="\*" element={<NoMatch />} />

        </Route>

      </Routes>

    </div>

  );

}

1. Node:

Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.

1. Node-sass:

Node-sass is a library that provides binding for Node.js to **[LibSass](https://github.com/sass/libsass" \t "_blank), the C version of the popular stylesheet preprocessor, Sass.**

It allows you to natively compile .scss files to css at incredible speed and automatically via a connect middleware.

1. History: The history library lets you easily manage **session**history anywhere JavaScript runs. I. It is an array which maintain all url where user has navigated. You can push or replace url

**<**Push location**=**{{

  pathname**:** '/home',

  search**:** '?the=query',

  hash**:** '#the-hash'

  state**:** { some**:** 'state' }

}}**/>**

 Or

this.props.history.push({

                    pathname: '/',

                    state: 4

                })

Best link –

|  |  |
| --- | --- |
| Topic | Link |
| C# | <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods> |
| Extension Method | <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods> |
| Constructor chaining | https://www.c-sharpcorner.com/UploadFile/825933/constructor-chaining-in-C-Sharp/ |
| SOLID | https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/ |
| LisKov substitution principle | https://www.linkedin.com/pulse/liskov-substitution-principle-lsp-paul-gichure-ctfl/ |
| Dependency inversion | https://dev.to/tamerlang/understanding-solid-principles-dependency-inversion-1b0f |
| Dependency injection | https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection |