1. OOP's Concepts with Example

OOP (Object-Oriented Programming) is a programming paradigm that revolves around the concept of "objects." These objects have properties (data) and methods (behavior).

Key OOP Concepts:

- **Encapsulation:**
  - Bundling data and methods that operate on that data within a single unit (class).
  - Example:
    C#
    ```
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public void Greet()
        {
            Console.WriteLine("Hello, my name is " + Name);
        }
    }
    ```

- **Inheritance:**
  - Creating new classes (child classes) based on existing classes (parent classes).
  - Example:
    C#
    ```
    public class Employee : Person
    {
        public string Designation { get; set; }
        public double Salary { get; set; }
    }
    ```

- **Polymorphism:**
  - The ability of objects to take on many forms.
  - **Method Overloading:** Defining multiple methods with the same name but different parameters.
  - **Method Overriding:** Redefining a method in a derived class to provide a specific implementation.
- **Abstraction:**
  - Focusing on essential features and hiding implementation details.
  - Example:
    C#
    ```
    public abstract class Shape
    {
        public abstract double Area();
    }
    ```

2. Custom Exception

A custom exception is a user-defined exception class that inherits from the Exception class or one of its derived classes. It allows you to create specific exception types to handle particular error conditions.

```
public class NegativeNumberException : Exception
{
    public NegativeNumberException(string message) : base(message) { }
}
```

3. Exception Class, Exception Base Class

- **Exception Class:** The base class for all exceptions in C#. It provides properties like Message, StackTrace, and InnerException.
- **Exception Base Class:** The Exception class itself is the base class for all exceptions.

4. ICollection Interface

The ICollection interface defines a general-purpose list. It provides methods for adding, removing, and searching elements, as well as properties for determining the number of elements and whether the collection is read-only.

5. How a For Loop Works

A for loop iterates over a sequence of values. It has three parts:

1. **Initialization:** Executes once before the loop starts.
2. **Condition:** Checked before each iteration. If true, the loop continues.
3. **Increment/Decrement:** Executed after each iteration.

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

6. What is a Constructor and Its Types

A constructor is a special method that is automatically called when an object of a class is created. It initializes the object's state.

- **Default Constructor:** A constructor with no parameters.
- **Parameterized Constructor:** A constructor that takes parameters to initialize the object's state.

7. Difference Between Abstract Classes and Interfaces

Feature	Abstract Class	Interface
Inheritance	Single inheritance	Multiple inheritance
Methods	Can have both abstract and concrete methods	Only abstract methods
Access Modifiers	Can have any access modifier	Only public, static, and abstract
Constructors	Can have constructors	Cannot have constructors

8. Overloading vs. Overriding

- **Overloading:** Defining multiple methods with the same name but different parameters within the same class.
- **Overriding:** Redefining a virtual or abstract method in a derived class to provide a specific implementation.

9. Reversing a String

```
string ReverseString(string str)
{
    char[] charArray = str.ToCharArray();
    Array.Reverse(charArray);
    return new string(charArray);
}
```

Using Built-in Function:

```
string ReverseString(string str)
{
    return new string(str.Reverse().ToArray());
}
```

10. Working of a Try-Catch Block

A try-catch block is used to handle exceptions.

1. **Try Block:** Contains code that might throw an exception.
2. **Catch Block:** Handles the exception if it occurs.

```
try
{
    int result = 10 / 0;
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
```

11. Finally Block

A finally block is executed regardless of whether an exception is thrown or caught. It's often used for cleanup operations.

```
try
{
    // Code that might throw an exception
}
catch (Exception ex)
{
    // Handle the exception
}
finally
{
    // Code to be executed regardless of exception
    Console.WriteLine("Finally block executed.");
}
```

12. Student Class, List, Array, and Printing

```
public class Student
{
    public string Name { get; set; }
    public int RollNumber { get; set; }

    public Student(string name, int rollNumber)
    {
        Name = name;
        RollNumber = rollNumber;
    }
}

class Program
{
    static void Main()
    {
```

```
        List<Student> students = new List<Student>();
        students.Add(new Student("Alice", 1));
        students.Add(new Student("Bob", 2));
        students.Add(new Student("Charlie", 3));

        // Printing students using a foreach loop
        foreach (Student student in students)
        {
            Console.WriteLine($"Name: {student.Name}, Roll Number:
{student.RollNumber}");
        }

        // Counting students
        int count = students.Count;
        Console.WriteLine($"Total Students: {count}");
    }
}
```

13. Properties, Getters, and Setters

- **Properties:** Provide a way to access and modify the values of private fields.
- **Getters:** Used to retrieve the value of a private field.
- **Setters:** Used to set the value of a private field.

14. How to Debug Your Code

- **Visual Studio Debugger:**
  - Set breakpoints to pause execution at specific lines.
  - Step through code line by line.
  - Inspect variables to see their values.
  - Use the Immediate Window to evaluate expressions.

15. Multithreading, Benefits

- **Multithreading:** Allows multiple threads of execution within a single process.
- **Benefits:**
  - Improved performance and responsiveness.
  - Efficient utilization of multi-core processors.
  - Better user experience (e.g., background tasks).

16. Types of Exceptions

- **System Exceptions:**
  - NullReferenceException
  - IndexOutOfRangeException
  - DivideByZeroException
  - InvalidOperationException
  - ArgumentException
- **User-Defined Exceptions:**
  - Custom exceptions created to handle specific error conditions.

17. Built-in Functions for List, String, Array

- **List:**
  - Add, Remove, Contains, IndexOf, Clear, Sort, Reverse
- **String:**
  - Length, ToUpper, ToLower, Substring, IndexOf, LastIndexOf, Trim, Split
- **Array:**
  - Length, Sort, Reverse, IndexOf, LastIndexOf, Copy

18. Benefits of StringBuilder

- **Performance:** More efficient for frequent string manipulations than string concatenation.
- **Flexibility:** Allows for dynamic string building.

19. Life Cycle of a Thread

1. **New:** Thread is created.
2. **Runnable:** Thread is ready to run, but waiting for CPU time.
3. **Running:** Thread is executing.
4. **Blocked:** Thread is waiting for a resource or event.
5. **Dead:** Thread has finished execution.

20. Find Duplicates in an Array

```
public static void FindDuplicates(int[] arr)
{
    HashSet<int> uniqueNumbers = new HashSet<int>();
    foreach (int num in arr)
    {
        if (!uniqueNumbers.Add(num))
        {
            Console.WriteLine(num + " is a duplicate.");
        }
    }
}
```

21. Multiple Interfaces

A class can implement multiple interfaces to inherit multiple sets of methods and properties. This allows for more flexibility and reusability.

```
interface IShape
{
    void Draw();
}

interface IColor
{
    string Color { get; set; }
```

```
}

class Circle : IShape, IColor
{
    public string Color { get; set; }

    public void Draw()
    {
        Console.WriteLine("Drawing a circle of color " + Color);
    }
}
```

22. Real-World Overriding

Consider a Shape base class with a virtual Draw() method. Derived classes like Circle and Rectangle can override this method to provide specific implementations.

23. Stack, Queue, Graph

- **Stack:**
  - LIFO (Last-In-First-Out) data structure.
  - Operations: Push, Pop, Peek.
  - Real-world example: Undo/Redo functionality.
- **Queue:**
  - FIFO (First-In-First-Out) data structure.
  - Operations: Enqueue, Dequeue, Peek.
  - Real-world example: Print queue.
- **Graph:**
  - A non-linear data structure consisting of nodes (vertices) and edges.
  - Used to represent networks, relationships, and dependencies.
  - Types: Directed and Undirected.
  - Algorithms: DFS, BFS, Dijkstra's, etc.

24. Difference Between For Loop and Foreach Loop

Feature	For Loop	Foreach Loop
Iteration	Explicit index-based iteration	Implicit iteration over elements
Collection Type	Arrays, Lists, any collection implementing IEnumerable	Any collection implementing IEnumerable
Flexibility	More control over iteration, can modify collection	Simpler syntax for

Feature	For Loop	Foreach Loop
	elements	read-only iteration

25. Searching, Finding Extremes, Sum, and Second Extremes

```
int[] numbers = { 5, 2, 9, 1, 7 };

// Search for a specific number
int searchValue = 7;
bool found = Array.Exists(numbers, x => x == searchValue);

// Find smallest and largest
int smallest = numbers.Min();
int largest = numbers.Max();

// Calculate sum
int sum = numbers.Sum();

// Find second smallest and second largest
int[] sortedNumbers = numbers.OrderBy(x => x).ToArray();
int secondSmallest = sortedNumbers[1];
int secondLargest = sortedNumbers[numbers.Length - 2];
```

26. Difference Between Property, Field, and Static Method

Feature	Property	Field	Static Method
Access	Through getter and setter	Direct access	Accessed directly using the class name
Scope	Instance-level	Instance-level or static	Class-level
Behavior	Can have logic in getter and setter	Simple data storage	Can be called without creating an instance of the class

27. How to Access const and static

- **const:** Can be accessed directly using the class name or an instance of the class.
- **static:** Can be accessed directly using the class name.

28. Concat, + in String

- **+:** Concatenates strings.
- **Concat:** More efficient for multiple string concatenations, especially in loops.

29. Singleton Class

A class that ensures only one instance of itself exists.

```
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();

    private Singleton() { }

    public static Singleton Instance
    {
        get { return instance; }
    }
}
```

30. Covariant, Static, Constant Parameter

- **Covariant Return Type:** A derived class can override a base class method and return a more derived type.
- **Static Parameter:** A parameter passed to a static method.
- **Constant Parameter:** A parameter whose value cannot be changed within the method.

31. Object

An object is an instance of a class. It has properties (data) and methods (behavior). Objects represent real-world entities or abstract concepts.

32. Class

A class is a blueprint for creating objects. It defines the properties and methods that objects of that class will have.

33. Why Do We Need Polymorphism?

Polymorphism allows objects of different types to be treated as if they were of the same type. This enables code reusability, flexibility, and easier maintenance.

34. Object vs. Dynamic vs. Var

- **Object:** A reference type that can hold any object. It's less type-safe and can lead to runtime errors if not used carefully.
- **dynamic:** A type that defers type checking to runtime, allowing for more flexibility but potentially less performance.
- **var:** A type inference keyword that lets the compiler determine the type of a variable based on its initialization.

35. Params Keyword

The params keyword allows a method to accept a variable number of arguments of the same type.

```
public void PrintNumbers(params int[] numbers)
{
    foreach (int number in numbers)
    {
        Console.WriteLine(number);
    }
}
```

36. Access Specifiers, File Handling

- **Access Specifiers:**
  - public: Accessible from anywhere.
  - private: Accessible only within the class.
  - protected: Accessible within the class and its derived classes.
  - internal: Accessible within the same assembly.
- **File Handling:**
  - **Reading:** StreamReader
  - **Writing:** StreamWriter
  - **Binary I/O:** BinaryReader, BinaryWriter

37. Primary Key, Foreign Key

- **Primary Key:** A unique identifier for a record in a table.
- **Foreign Key:** A column in one table that references the primary key of another table.

38. Create a Table

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Age INT
);
```

39. Index
An index is a data structure that improves the speed of data retrieval operations on a database table. It creates a sorted copy of specific columns, allowing for faster searches and sorting.

40. Join

A join is a SQL operation that combines rows from two or more tables, based on a related column between them.

- **Inner Join:** Returns rows that have matching values in both tables.
- **Left Join:** Returns all rows from the left table, and the matched rows from the right table.
- **Right Join:** Returns all rows from the right table, and the matched rows from the left table.
- **Full Outer Join:** Returns all rows when there is a match in either left or right table.

41. Varchar vs Char

Both VARCHAR and CHAR are data types used to store character strings in SQL.

- **VARCHAR:** Variable-length character string. The storage size is based on the actual length of the data entered.
- **CHAR:** Fixed-length character string. The storage size is fixed, regardless of the actual length of the data.

42. SQL View

A SQL view is a virtual table based on the result-set of an SQL statement. It doesn't store data directly but provides a different way to view the data in the underlying tables.

43. LINQ Methods

LINQ (Language Integrated Query) provides a unified way to query data sources like databases, XML documents, and collections. Common LINQ methods include:

- **Filtering:** Where
- **Projection:** Select
- **Ordering:** OrderBy, OrderByDescending
- **Grouping:** GroupBy
- **Quantifiers:** Any, All
- **Set Operations:** Union, Intersect, Except

44. Delegates and Its Types

A delegate is a type that represents a reference to a method.

- **Singlecast Delegate:** Can refer to only one method.
- **Multicast Delegate:** Can refer to multiple methods, which are invoked sequentially.

45. Events

An event is a notification mechanism that allows one object to signal another object that something has happened. Events are often used to implement observer patterns.

46. Normalization

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. It involves breaking down a large table into smaller tables and defining relationships between them.

47. Namespace

A namespace is a declarative region that holds declarations of types, such as classes, interfaces, structs, enums, delegates, and modules. It helps to organize code and avoid naming conflicts.

48. Enum

An enum (enumeration) is a user-defined data type that consists of a set of named constants. It's used to define a set of related values.

49. SDLC (Software Development Life Cycle) and Its Types

SDLC is a framework that defines the stages involved in developing software.

Common SDLC Models:

- **Waterfall Model:** A linear, sequential approach.
- **Agile Model:** Iterative and incremental development.
- **Iterative Model:** Combines elements of both waterfall and prototype models.
- **Spiral Model:** Risk-driven approach with iterative phases.
- **V-Model:** Emphasizes verification and validation at each stage.

STRING

String Functions:

1. Length:

- **Purpose:** Returns the number of characters in the string.
- **Syntax:** string.Length
- **Example:**

```
string str = "Hello, World!";
int length = str.Length; // length will be 13
```

2. CompareTo:

- **Purpose:** Compares the current string with another string.
- **Syntax:** string.CompareTo(string other)
- **Returns:**
  - -1 if the current string is less than the other string.
  - 0 if the strings are equal.
  - 1 if the current string is greater than the other string.
- **Example:**
- ```
  string str1 = "Apple";
  string str2 = "Banana";
  int result = str1.CompareTo(str2); // result will be -1
  ```

3. Equals:

- **Purpose:** Determines whether two strings are equal.
- **Syntax:** string.Equals(string other)
- **Returns:** True if the strings are equal, otherwise false.
- **Example:**

```
string str1 = "Hello";
string str2 = "Hello";
bool areEqual = str1.Equals(str2); // areEqual will be true
```

4. Contains:

- **Purpose:** Checks if a substring exists within the current string.
- **Syntax:** string.Contains(string value)
- **Returns:** True if the substring exists, otherwise false.
- **Example**

```
string str = "Hello, World!";
bool contains = str.Contains("World"); // contains will be true
```

5. StartsWith:

- **Purpose:** Checks if the current string starts with a specified substring.
- **Syntax:** string.StartsWith(string value)
- **Returns:** True if the string starts with the substring, otherwise false.
- **Example:**

```
string str = "Hello, World!";
bool startsWith = str.StartsWith("Hello"); // startsWith will be true
```

6. EndsWith:

- **Purpose:** Checks if the current string ends with a specified substring.
- **Syntax:** string.EndsWith(string value)
- **Returns:** True if the string ends with the substring, otherwise false.
- **Example:**

```
string str = "Hello, World!";
bool endsWith = str.EndsWith("World!"); // endsWith will be true
```
```

7. IndexOf:

- **Purpose:** Finds the index of the first occurrence of a specified character or substring within the current string.
- **Syntax:** string.IndexOf(char value) or string.IndexOf(string value)
- **Returns:** The index of the first occurrence, or -1 if not found.
- **Example:**

```
string str = "Hello, World!";
int index = str.IndexOf('o'); // index will be 4
```

8. LastIndexOf:

- **Purpose:** Finds the index of the last occurrence of a specified character or substring within the current string.
- **Syntax:** string.LastIndexOf(char value) or string.LastIndexOf(string value)
- **Returns:** The index of the last occurrence, or -1 if not found.
- **Example:**

```
string str = "Hello, World!";
int index = str.LastIndexOf('o'); // index will be 7
```

9. Substring:

- **Purpose:** Extracts a substring from the current string.
- **Syntax:** string.Substring(int startIndex) or string.Substring(int startIndex, int length)
- **Returns:** The extracted substring.
- **Example:**

```
string str = "Hello, World!";
string substring = str.Substring(7); // substring will be "World!"
```

10. Replace:

- **Purpose:** Replaces all occurrences of a specified character or substring with another string.
- **Syntax:** string.Replace(char oldChar, char newChar) or string.Replace(string oldString, string newString)
- **Returns:** The modified string.
- **Example:**

```
string str = "Hello, World!";
string replaced = str.Replace("World", "Universe"); // replaced will
be "Hello, Universe!"
```

11. ToUpper:

- **Purpose:** Converts all characters in the current string to uppercase.
- **Syntax:** string.ToUpper()
- **Returns:** The uppercase string.
- **Example:**

```
string str = "Hello, World!";
string upper = str.ToUpper(); // upper will be "HELLO, WORLD!"
```

12. ToLower:

- **Purpose:** Converts all characters in the current string to lowercase.
- **Syntax:** string.ToLower()
- **Returns:** The lowercase string.
- **Example:**

```
string str = "Hello, World!";
string lower = str.ToLower(); // lower will be "hello, world!"
```

13. Trim:

- **Purpose:** Removes leading and trailing white space characters from the current string.
- **Syntax:** string.Trim()
- **Returns:** The trimmed string.
- **Example:**

```
string str = "  Hello, World!  ";
string trimmed = str.Trim(); // trimmed will be "Hello, World!"
```

String Methods:

1. Split:

- **Purpose:** Splits the current string into a string array based on a specified delimiter.
- **Syntax:** string.Split(char[] separator) or string.Split(char[] separator, StringSplitOptions options)
- **Returns:** The string array.
- **Example:**

```
string str = "Hello,World,How,Are,You";
string[] words = str.Split(','); // words will be ["Hello", "World",
"How", "Are", "You"]
```

2. Join:

- **Purpose:** Joins the elements of a string array into a single string using a specified delimiter.
- **Syntax:** string.Join(string separator, string[] values)
- **Returns:** The joined string.
- **Example:**

```
string[] words = {"Hello", "World", "How", "Are", "You"};
string joined = string.Join(", ", words); // joined will be "Hello,
World, How, Are, You"
```

3. Format:

- **Purpose:** Formats a string using placeholders and corresponding values.
- **Syntax:** string.Format(string format, object[] args)
- **Returns:** The formatted string.
- **Example**

```
string name = "Alice";
int age = 30;
string formatted = string.Format("Hello, {0}! You are {1} years old.",
name, age);
// formatted will be "Hello, Alice! You are 30 years old."
```

4. PadLeft:

- **Purpose:** Pads the left side of the current string with a specified character to a specified length.
- **Syntax:** string.PadLeft(int totalWidth) or string.PadLeft(int totalWidth, char paddingChar)
- **Returns:** The padded string.
- **Example:**

```
string str = "Hello";
string padded = str.PadLeft(10, '-'); // padded will be "-----Hello"
```

5. PadRight:

- **Purpose:** Pads the right side of the current string with a specified character to a specified length.
- **Syntax:** string.PadRight(int totalWidth) or string.PadRight(int totalWidth, char paddingChar)
- **Returns:** The padded string.
- **Example:**

```
string str = "Hello";
string padded = str.PadRight(10, '-'); // padded will be "Hello-----"
```

6. Remove:

- **Purpose:** Removes a specified number of characters from the current string, starting at a specified index.
- **Syntax:** string.Remove(int startIndex) or string.Remove(int startIndex, int count)
- **Returns:** The modified string.
- **Example:**

```
string str = "Hello, World!";
string removed = str.Remove(7, 5); // removed will be "Hello, !"
```

7. Insert:

- **Purpose:** Inserts a specified string into the current string at a specified index.
- **Syntax:** string.Insert(int startIndex, string value)
- **Returns:** The modified string.
- **Example**

```
string str = "Hello, World!";
string inserted = str.Insert(7, "Beautiful "); // inserted will be
"Hello, Beautiful World!"
```

8. ToCharArray:

- **Purpose:** Converts the current string into a character array.
- **Syntax:** string.ToCharArray()
- **Returns:** The character array.
- **Example:**

```
string str = "Hello, World!";
char[] chars
```

9. ToCharArray:

- **Purpose:** Converts the current string into a character array.
- **Syntax:** string.ToCharArray()
- **Returns:** The character array.
- **Example:**

```
string str = "Hello, World!";
char[] chars = str.ToCharArray(); // chars will be ['H', 'e', 'l',
'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd', '!']
```

10. Intern:

- **Purpose:** Retrieves the string from the string pool if it already exists, otherwise adds it to the pool and returns a reference to it.
- **Syntax:** string.Intern(string str)
- **Returns:** The interned string.
- **Example:**

```
string str1 = "Hello";
string str2 = string.Intern(str1);
// str1 and str2 will refer to the same object in the string pool
```

11. Copy:

- **Purpose:** Copies the current string to a new character array.
- **Syntax:** string.CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count)
- **Returns:** None.
- **Example:**

```
string str = "Hello, World!";
char[] chars = new char[13];
str.CopyTo(0, chars, 0, 13); // chars will be ['H', 'e', 'l', 'l',
'o', ',', ' ', 'W', 'o', 'r', 'l', 'd', '!']
```

12. Normalize:

- **Purpose:** Normalizes the current string according to the specified Unicode normalization form.
- **Syntax:** string.Normalize(NormalizationForm form)
- **Returns:** The normalized string.

- **Example:**

```
string str = "café";
string normalized = str.Normalize(NormalizationForm.FormC);
// normalized will be "cafe" (normalized form)
```

13. IsNormalized:

- **Purpose:** Checks if the current string is normalized according to the specified Unicode normalization form.
- **Syntax:** string.IsNormalized(NormalizationForm form)
- **Returns:** True if the string is normalized, otherwise false.
- **Example:**

```
string str = "café";
bool isNormalized = str.IsNormalized(NormalizationForm.FormC);
// isNormalized will be false
```

14. IsNullOrEmpty:

- **Purpose:** Checks if the specified string is null or empty.
- **Syntax:** string.IsNullOrEmpty(string str)
- **Returns:** True if the string is null or empty, otherwise false.
- **Example:**

```
string str = null;
bool isNullOrEmpty = string.IsNullOrEmpty(str);
// isNullOrEmpty will be true
```

15. IsNullOrWhiteSpace:

- **Purpose:** Checks if the specified string is null, empty, or consists only of white space characters.
- **Syntax:** string.IsNullOrWhiteSpace(string str)
- **Returns:** True if the string is null, empty, or consists only of white space characters, otherwise false.
- **Example:**

```
string str = "   ";
bool isNullOrWhiteSpace = string.IsNullOrWhiteSpace(str);
// isNullOrWhiteSpace will be true
```

ARRAY

Functions and Methods:

Indexer

- **Purpose:** Access or modify elements of an array by their index.
- **Syntax:** array[index]
- **Example:**
  C#
```
int[] numbers = { 1, 2, 3, 4, 5 };
int secondElement = numbers[1]; // Accesses the second element
(index 1)
numbers[3] = 10; // Modifies the fourth element (index 3)
```

Length Property

- **Purpose:** Returns the total number of elements in an array.
- **Syntax:** array.Length
- **Example:**
  C#
```
int[] numbers = { 1, 2, 3, 4, 5 };
int length = numbers.Length; // length will be 5
```

Rank Property

- **Purpose:** Returns the number of dimensions in an array.
- **Syntax:** array.Rank
- **Example:**
  C#
```
int[] numbers = { 1, 2, 3, 4, 5 }; // One-dimensional array
int rank = numbers.Rank; // rank will be 1
```

GetLength Method

- **Purpose:** Returns the length of a specific dimension in a multidimensional array.
- **Syntax:** array.GetLength(dimension)
- **Example:**

C#
```
int[,] matrix = { { 1, 2 }, { 3, 4 } }; // Two-dimensional array
int length = matrix.GetLength(0); // Length of the first dimension
(rows)
```

GetLowerBound and GetUpperBound Methods

- **Purpose:** Returns the lower and upper bounds of a specific dimension in an array.
- **Syntax:** array.GetLowerBound(dimension) and array.GetUpperBound(dimension)
- **Example:**
  C#
  ```
  int[] numbers = { 1, 2, 3, 4, 5 };
  int lowerBound = numbers.GetLowerBound(0); // Lower bound will be 0
  int upperBound = numbers.GetUpperBound(0); // Upper bound will be 4
  ```

Clone Method

- **Purpose:** Creates a shallow copy of an array.
- **Syntax:** array.Clone()
- **Example:**
  C#
  ```
  int[] numbers = { 1, 2, 3, 4, 5 };
  int[] clonedArray = numbers.Clone();
  ```

CopyTo Method

- **Purpose:** Copies a portion of an array to another array.
- **Syntax:** array.CopyTo(Array destination, int destinationIndex, int startIndex, int length)
- **Example:**
  C#
  ```
  int[] source = { 1, 2, 3, 4, 5 };
  int[] destination = new int[3];
  source.CopyTo(destination, 0, 1, 3); // Copies elements 1, 2, and 3
  to destination
  ```

Clear Method

- **Purpose:** Clears all elements of an array to their default values.
- **Syntax:** array.Clear()
- **Example:**
  C#
  ```
  int[] numbers = { 1, 2, 3, 4, 5 };
  numbers.Clear(); // All elements will be set to 0
  ```

Reverse Method

- **Purpose:** Reverses the order of elements in an array.
- **Syntax:** array.Reverse()
- **Example:**
  C#
  ```
  int[] numbers = { 1, 2, 3, 4, 5 };
  numbers.Reverse(); // numbers will be { 5, 4, 3, 2, 1 }
  ```

Sort Method

- **Purpose:** Sorts the elements of an array.
- **Syntax:** array.Sort() or Array.Sort(array, Comparer)
- **Example:**
  C#
  ```
  int[] numbers = { 3, 1, 4, 1, 5, 9, 2, 6, 5 };
  Array.Sort(numbers); // numbers will be sorted in ascending order
  ```

LIST

Common Functions and Methods:

Add Method

- **Purpose:** Adds an element to the end of the list.
- **Syntax:** list.Add(item)
- **Example:**
  ```
  List<int> numbers = new List<int>();
  numbers.Add(1);
  numbers.Add(2);
  ```

Insert Method

- **Purpose:** Inserts an element at a specified index in the list.
- **Syntax:** list.Insert(index, item)
- **Example:**

```
List<string> fruits = new List<string>() { "Apple", "Banana",
"Orange" };
fruits.Insert(1, "Mango"); // Inserts "Mango" at index 1
```

Remove Method

- **Purpose:** Removes the first occurrence of a specified element from the list.
- **Syntax:** list.Remove(item)
- **Example:**

```
C#
List<int> numbers = new List<int>() { 1, 2, 3, 2, 4 };
numbers.Remove(2); // Removes the first occurrence of 2
```

RemoveAt Method

- **Purpose:** Removes the element at a specified index from the list.
- **Syntax:** list.RemoveAt(index)
- **Example:**

```
C#
List<string> fruits = new List<string>() { "Apple", "Banana",
"Orange" };
fruits.RemoveAt(1); // Removes the element at index 1 (Banana)
```

Clear Method

- **Purpose:** Removes all elements from the list.
- **Syntax:** list.Clear()
- **Example:**

```
C#
List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
numbers.Clear(); // Empties the list
```

Contains Method

- **Purpose:** Checks if a specific element exists in the list.
- **Syntax:** list.Contains(item)
- **Example:**
  C#
  ```
  List<string> fruits = new List<string>() { "Apple", "Banana",
  "Orange" };
  bool contains = fruits.Contains("Mango"); // Contains will be false
  ```

IndexOf Method

- **Purpose:** Returns the index of the first occurrence of a specified element in the list.
- **Syntax:** list.IndexOf(item)
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 1, 2, 3, 2, 4 };
  int index = numbers.IndexOf(2); // Index will be 0
  ```

LastIndexOf Method

- **Purpose:** Returns the index of the last occurrence of a specified element in the list.
- **Syntax:** list.LastIndexOf(item)
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 1, 2, 3, 2, 4 };
  int index = numbers.LastIndexOf(2); // Index will be 3
  ```

Exists Method

- **Purpose:** Checks if any element in the list satisfies a specified condition.
- **Syntax:** list.Exists(predicate)
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
  bool exists = numbers.Exists(x => x > 3); // Exists will be true
  ```

Find Method

- **Purpose:** Finds the first element in the list that satisfies a specified condition.
- **Syntax:** list.Find(predicate)
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
  int firstEven = numbers.Find(x => x % 2 == 0); // firstEven will be
  2
  ```

FindIndex Method

- **Purpose:** Returns the index of the first element in the list that satisfies a specified condition.
- **Syntax:** list.FindIndex(predicate)
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
  int index = numbers.FindIndex(x => x > 3); // Index will be 2
  ```

ForEach Method

- **Purpose:** Executes a specified action on each element in the list.
- **Syntax:** list.ForEach(action)
- **Example:**
  C#
  ```
  List<string> fruits = new List<string>() { "Apple", "Banana",
  "Orange" };
  fruits.ForEach(Console.WriteLine); // Prints each fruit to the
  console
  ```

TrueForAll Method

- **Purpose:** Checks if all elements in the list satisfy a specified condition.
- **Syntax:** list.TrueForAll(predicate)
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
  ```

```
bool allPositive = numbers.TrueForAll(x => x > 0); // allPositive
will be true
```

GetRange Method

- **Purpose:** Returns a new list containing elements from the specified index to the end of the original list.
- **Syntax:** list.GetRange(startIndex, count)
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
  List<int> sublist = numbers.GetRange(1, 3); // sublist will contain
  { 2, 3, 4 }
  ```

RemoveAll Method

- **Purpose:** Removes all elements from the list that satisfy a specified condition.
- **Syntax:** list.RemoveAll(predicate)
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
  numbers.RemoveAll(x => x % 2 == 0); // Removes all even numbers
  ```

Sort Method

- **Purpose:** Sorts the elements of the list.
- **Syntax:** list.Sort() or list.Sort(comparer)
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 3, 1, 4, 1, 5, 9, 2, 6, 5 };
  numbers.Sort(); // Sorts in ascending order
  ```

ConvertAll Method

- **Purpose:** Converts all elements in the list to a new type using a specified converter function.
- **Syntax:** list.ConvertAll(converter)
- **Example:**
  ```
  List<string> numbers = new List<string>() { "1", "2", "3" };
  List<int> intNumbers = numbers.ConvertAll(int.Parse); // Converts
  strings to integers
  ```

ToArray Method

- **Purpose:** Converts the list to an array.
- **Syntax:** list.ToArray()
- **Example:**
  C#
  ```
  List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
  int[] array = numbers.ToArray();
  ```

Dictionaries

Common Functions and Methods:

Add Method

- **Purpose:** Adds a new key-value pair to the dictionary.
- **Syntax:** dictionary.Add(key, value)
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  ages.Add("Alice", 30);
  ages.Add("Bob", 25);
  ```

ContainsKey Method

- **Purpose:** Checks if a specific key exists in the dictionary.
- **Syntax:** dictionary.ContainsKey(key)
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  bool contains = ages.ContainsKey("Alice"); // Contains will be true
  ```

ContainsValue Method

- **Purpose:** Checks if a specific value exists in the dictionary.
- **Syntax:** dictionary.ContainsValue(value)

- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  bool contains = ages.ContainsValue(30); // Contains will be true
  ```

Remove Method

- **Purpose:** Removes a key-value pair from the dictionary.
- **Syntax:** dictionary.Remove(key)
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  ages.Remove("Alice"); // Removes the key-value pair for "Alice"
  ```

Clear Method

- **Purpose:** Removes all key-value pairs from the dictionary.
- **Syntax:** dictionary.Clear()
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  ages.Clear(); // Empties the dictionary
  ```

Count Property

- **Purpose:** Returns the number of key-value pairs in the dictionary.
- **Syntax:** dictionary.Count
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  int count = ages.Count;
  ```

Keys Property

- **Purpose:** Returns a collection of keys in the dictionary.
- **Syntax:** dictionary.Keys
- **Example:**

C#
```
Dictionary<string, int> ages = new Dictionary<string, int>();
foreach (string key in ages.Keys)
{
    Console.WriteLine(key);
}
```

Values Property

- **Purpose:** Returns a collection of values in the dictionary.
- **Syntax:** dictionary.Values
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  foreach (int value in ages.Values)
  {
      Console.WriteLine(value);
  }
  ```

TryGetValue Method

- **Purpose:** Attempts to retrieve the value associated with a specified key.
- **Syntax:** dictionary.TryGetValue(key, out value)
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  int age;
  if (ages.TryGetValue("Alice", out age))
  {
      Console.WriteLine("Age: " + age);
  }
  ```

ToDictionary Method

- **Purpose:** Converts the dictionary to a new dictionary of a different type.
- **Syntax:** dictionary.ToDictionary(keySelector, elementSelector)
- **Example:**
  C#
  ```
  Dictionary<string, int> numbers = new Dictionary<string, int>();
  numbers.Add("one", 1);
  numbers.Add("two", 2);
  ```

```
Dictionary<int, string> reversed = numbers.ToDictionary(kvp =>
kvp.Value, kvp => kvp.Key);
```

TryAdd Method

- **Purpose:** Attempts to add a new key-value pair to the dictionary.
- **Syntax:** dictionary.TryAdd(key, value)
- **Returns:** True if the key-value pair was added, false if the key already exists.
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  bool added = ages.TryAdd("Alice", 30);
  ```

GetOrAdd Method

- **Purpose:** Attempts to retrieve the value associated with a specified key. If the key doesn't exist, it adds a new key-value pair using a specified function.
- **Syntax:** dictionary.GetOrAdd(key, valueFactory)
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  int age = ages.GetOrAdd("Alice", () => 30);
  ```

RemoveAll Method

- **Purpose:** Removes all key-value pairs from the dictionary that satisfy a specified condition.
- **Syntax:** dictionary.RemoveAll(predicate)
- **Example:**
  C#
  ```
  Dictionary<string, int> ages = new Dictionary<string, int>();
  ages.RemoveAll(kvp => kvp.Value > 30);
  ```

Object-Oriented Programming (OOP)

OOP is a programming paradigm based on the concept of "objects." Objects encapsulate data (fields or properties) and code (methods) into self-contained units. This modular approach promotes code reusability, maintainability, and flexibility.

Key Principles of OOP

1. **Encapsulation:**
   - Bundling data and methods that operate on the data within a single unit.
   - Enforces data integrity and prevents unauthorized access.
   - Example:
     C#
     ```
     public class Car
     {
         private string model; // Private field

         public Car(string model)
         {
             this.model = model;
         }

         public void Drive()
         {
             Console.WriteLine($"{model} is driving.");
         }
     }
     ```

2. **Abstraction:**
   - Simplifying complexity by focusing on essential features and hiding unnecessary details.
   - Promotes code reusability and maintainability.
   - Example:
     - A Shape class with abstract methods like GetArea() and GetPerimeter().
     - Derived classes like Circle and Rectangle implement these methods.
3. **Inheritance:**
   - Creating new classes (derived classes) based on existing classes (base classes).
   - Derived classes inherit properties and methods from the base class.
   - Promotes code reuse and hierarchical relationships between classes.
   - Example:
     ```
     public class Animal
     {
         public virtual void MakeSound()
         {
             Console.WriteLine("Generic animal sound.");
         }
     }

     public class Dog : Animal
     ```

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

4. **Polymorphism:**
    - Allowing objects of different types to be treated as if they were of the same type.
    - Enables method overloading and overriding.
    - Supports dynamic behavior and flexibility.
    - Example:
        - A CalculateArea() method that can accept different shapes and calculate their areas based on their specific implementations.

Properties and Fields

- **Fields:** Directly defined data members within a class.
- **Properties:** Provide controlled access to data using getters and setters.
- **Auto-implemented properties:** Simplify property declaration with default getters and setters.

C#
```
public class Example
{
    private string field; // Field

    public string Property { get; set; } // Auto-implemented Property
}
```

Constructors

- Special methods used to initialize objects.
- Have the same name as the class and no return type.
- **Default constructor:** Initializes fields with default values.
- **Parameterized constructor:** Accepts parameters to initialize fields.
- **Static constructor:** Initializes static fields, called once per type.
- **Private constructor:** Prevents instantiation from outside the class, often used in singleton patterns.
- **Copy constructor:** Creates a new object by copying values from another object.
- **Constructor overloading:** Multiple constructors with different parameters.

Interfaces

- Define contracts that classes or structs must follow.
- Have no implementation, only declarations.
- Support multiple inheritance.
- Members are implicitly public.
- Introduced default members in C# 8+.

```
public interface IShape
{
    double GetArea();
    double GetPerimeter();
}
```

Function Overloading / Method Overloading

- Having multiple methods with the same name but different parameter lists (number, type, order).

Virtual and Override Keywords

- Enable polymorphism by allowing derived classes to modify or extend base class methods.
- **virtual:** Allows a method to be overridden.
- **override:** Provides a new implementation for an inherited virtual or abstract method.

Abstract Classes

- Cannot be instantiated directly.
- Used to define base classes with common methods and enforce derived classes to implement specific functionality.
- Contains both abstract members (must be implemented in derived classes) and concrete members (methods with a body).

C#

```
public abstract class BaseClass
{
    public abstract void AbstractMethod(); // Abstract method
    public void ConcreteMethod() // Concrete method
    {
        Console.WriteLine("This is a concrete method.");
    }
}
```

When to Use Abstract Classes

- When sharing code among several closely related classes.
- To define a base class that should not be instantiated directly.
- When some methods need default implementations while others should be enforced for subclasses.

Delegates

- Types that represent references to methods with a specific parameter list and return type.
- Used for callback methods and event-driven programming.

```
public delegate void MyDelegate(string message);
```

Classes in C#

A class is a blueprint for creating objects in C#. It defines the properties (data members) and methods (functions) that objects of that class will have. Think of a class as a cookie cutter; it defines the shape and characteristics of the cookies that will be produced.

Real-Life Example:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public void Greet()
    {
        Console.WriteLine("Hello, my name is " + Name);
    }
}
```

In this example, the Person class represents a person with properties Name and Age. The Greet method is used to print a greeting message.

Why Write Classes?

- **Encapsulation:** Classes encapsulate data and behavior, promoting modularity and reusability.
- **Object-Oriented Programming (OOP):** C# is an OOP language, and classes are fundamental to its paradigm.
- **Data Modeling:** Classes are used to model real-world entities and concepts.

Naming Convention:

In C#, class names typically follow Pascal casing, where each word starts with an uppercase letter. For example, Person, Customer, and Product.

Constructors

A constructor is a special method that is automatically called when an object of a class is created. It initializes the properties of the object.

Real-Life Example:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age)
    {        Name = name;
```

```
        Age = age;
    }
}
```

This example defines a parameterized constructor for the Person class that takes the person's name and age as arguments.

Access Modifiers

Access modifiers control the visibility of class members (properties, methods, etc.) from other parts of the code.

Real-Life Example:

```
public class BankAccount
{
    private double balance;

    public BankAccount(double initialBalance)
    {
        balance = initialBalance;
    }

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

    public void Withdraw(double amount)
    {
        if (balance >= amount)
        {
            balance -= amount;
        }
        else
        {
            Console.WriteLine("Insufficient funds.");
        }
    }

    public double GetBalance()
    {
        return balance;
    }
}
```

In this example, the balance property is marked as private, ensuring that it can only be

accessed and modified within the BankAccount class.

Passing Any Number of Arguments

The params keyword allows you to pass a variable number of arguments to a method:

Real-Life Example:

```
public class Calculator
{
    public double Add(params double[] numbers)
    {
        double sum = 0;
        foreach (double number in numbers)
        {
            sum += number;
        }
        return sum;
    }
}
```

This example defines a Calculator class with an Add method that can take any number of double values as arguments and return their sum.

Namespaces

Namespaces are used to organize code into logical groups. They help prevent naming conflicts and improve code readability.

Real-Life Example:

```
namespace MyApplication
{
    public class Program
    {
        public static void Main()
        {
            // ...
        }
    }
}
```

In this example, the MyApplication namespace is used to group related classes and code within a project.

Static, Constant, and Readonly

- **static:** Modifies members that belong to the class itself, rather than individual objects. They can be accessed without creating an object.

- **const:** Declares a constant whose value cannot be changed after compilation.
- **readonly:** Declares a read-only member whose value can be assigned only once, either in the declaration or the constructor.

Real-Life Example:

```
public class MathHelper
{
    public const double Pi = 3.14159;

    public static double CalculateArea(double radius)
    {
        return Pi * radius * radius;
    }
}
```

In this example, Pi is a constant value that can be used throughout the application, and CalculateArea is a static method that can be called without creating an instance of the MathHelper class.

Function Overloading

Function overloading allows you to have multiple methods with the same name but different parameter lists. The compiler determines the appropriate method to call based on the arguments provided.

Real-Life Example:

```
public class Shape
{
    public virtual double CalculateArea()
    {
        return 0;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }

    public override double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}

public class Rectangle : Shape
{
    public double Width { get; set; }
```

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

    public override double CalculateArea()
    {
        return Width * Height;
    }
}
```

In this example, the Shape class defines a virtual CalculateArea method. The Circle and Rectangle classes override this method to provide their specific area calculations.

Properties and Fields

- **Fields:** Direct members of a class that store data.
- **Properties:** Provide a controlled way to access and modify fields, often with additional logic.

Auto-Properties

Auto-properties are a shorthand way to declare properties with backing fields.

Real-Life Example:

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public double Price { get; set; }
}
```

This example defines a Product class with auto-properties for Id, Name, and Price.

Getters and Setters

Getters and setters are used within properties to control access to the underlying field. They can be used to perform validation, calculations, or other operations.

Real-Life Example:

```
public class BankAccount
{
    private double balance;

    public double Balance
    {
        get { return balance; }
        set
        {
            if (value >= 0)
```

```
            {
                balance = value;
            }
            else
            {
                Console.WriteLine("Invalid balance.");
            }
        }
    }
}
```

In this example, the Balance property has a setter that ensures the balance is always non-negative.

Abstract Classes and Interfaces

- **Abstract Class:** A class that cannot be instantiated directly. It can contain abstract methods (methods without implementation) that must be overridden by derived classes.
- **Interface:** A contract that defines the methods and properties that a class must implement. It cannot contain fields.

Real-Life Example:

```
public abstract class Animal
{
    public abstract void MakeSound();
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}

public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meow!");
    }
}
```

In this example, the Animal class is abstract and defines an abstract MakeSound method. The Dog and Cat classes implement this method to provide their specific sounds.

Types of Inheritance

- **Single Inheritance:** A class can inherit from only one base class.
- **Multiple Inheritance:** A class can inherit from multiple base classes (using interfaces).

Real-Life Example:

```
public interface IShape
{
    double CalculateArea();
}

public class Circle : IShape
{
    // ...
}

public class Rectangle : IShape
{
    // ...
}
```

In this example, the Circle and Rectangle classes both implement the IShape interface, allowing them to be treated as shapes without directly inheriting from a common base class.

Why Object-Oriented Programming

- **Reusability:** Code can be reused through inheritance and polymorphism.
- **Maintainability:** Code is easier to maintain and understand when organized into classes and objects.
- **Flexibility:** OOP allows for flexible and adaptable software design.

Data Hiding

Encapsulating data within classes prevents unauthorized access and modification.

Abstraction

Focusing on the essential features of a problem while ignoring irrelevant details.

Polymorphism

The ability of objects of different types to be treated as if they were of the same type.

Delegates, Actions, and Func

- **Delegate:** A type that represents a method signature.
- **Action:** A generic delegate that

Predicate:

A delegate that represents a method that takes an object and returns a boolean value.

Real-Life Example:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public static bool IsAdult(Person person)
{
    return person.Age >= 18;
}

List<Person> people = new List<Person>()
{
    new Person { Name = "Alice", Age = 25 },
    new Person { Name = "Bob", Age = 15 },
    new Person { Name = "Charlie", Age = 30 }
};

List<Person> adults = people.Where(IsAdult).ToList();
```

In this example, the IsAdult predicate is used to filter the list of people to only include those who are adults.

Lambda Expressions

Anonymous functions that can be used to create delegates.

Real-Life Example:

```
List<Person> adults = people.Where(p => p.Age >= 18).ToList();
```

This example uses a lambda expression to achieve the same result as the previous example, without defining a separate IsAdult method.

Finalize/Destructor

A method that is automatically called by the garbage collector before an object is reclaimed.

Real-Life Example:

```
public class ResourceDisposable : IDisposable
{
    private bool isDisposed;

    public void Dispose()
    {
        if (!isDisposed)
        {
            // Release unmanaged resources here
            isDisposed = true;
        }
    }

    ~ResourceDisposable()
    {
        Dispose();
    }
}
```

In this example, the ResourceDisposable class implements the IDisposable interface and provides a Dispose method to release unmanaged resources. The destructor ensures that the Dispose method is called even if the object is not explicitly disposed of.

Managed and Unmanaged Code

- **Managed Code:** Code that is managed by the .NET runtime.
- **Unmanaged Code:** Code that is not managed by the .NET runtime.

Real-Life Example:

```
// Managed code
int[] numbers = { 1, 2, 3 };

// Unmanaged code (using P/Invoke)
[DllImport("user32.dll")]
static extern bool MessageBox(IntPtr hWnd, string lpText, string
lpCaption, uint uType);

MessageBox(IntPtr.Zero, "Hello, world!", "Message", 0);
```

In this example, the numbers array is managed code, while the MessageBox function is

unmanaged code. P/Invoke is used to call the unmanaged function from managed code.

String Functions

C# provides numerous string functions for manipulation and analysis.

Real-Life Example:

```
string text = "Hello, world!";

string upperCaseText = text.ToUpper();
string substring = text.Substring(7);
bool containsWorld = text.Contains("world");
```

List

A generic collection that represents an ordered list of elements.

Real-Life Example:

```
List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };

numbers.Add(6);
numbers.RemoveAt(2);
int firstNumber = numbers[0];
```

Dictionary

A generic collection that represents a key-value pair collection.

Real-Life Example:

```
Dictionary<string, int> ages = new Dictionary<string, int>();

ages.Add("Alice", 25);
ages.Add("Bob", 15);
ages.Add("Charlie", 30);

int aliceAge = ages["Alice"];
```