C# Objects and Classes

Introduction to Object-Oriented Programming in C#

C# is an object-oriented programming (OOP) language, which means that it is designed around the concept of objects and classes. This paradigm allows for the modeling of real-world entities and relationships, making it easier to manage and organize code.

C# Object

Definition

In C#, an **Object** is a real-world entity that has a distinct identity, state, and behavior. Examples of objects include:

Chair, Car, Pen, Mobile phone,Laptop

Characteristics

State: Represents the data or attributes of the object (e.g., color, size).

Behavior: Represents the functionality or methods of the object (e.g., drive, write).

Runtime Entity: Objects are created during the execution of a program (runtime).

Instance of a Class: An object is an instance of a class, meaning it can access all the members (fields, methods) defined in that class.

Example of Creating an Object

To create an object in C#, you typically use the `new` keyword followed by the class name. Here's an **example:**

```
Student s1 = new Student(); // Creating an object of Student
```

In this example:

`Student` is the class type.

`s1` is the reference variable that holds the reference to the instance of the `Student` class.

The `new` keyword allocates memory for the object at runtime.

C# Class

Definition

In C#, a **Class** is a blueprint or template for creating objects. It defines the properties (fields) and behaviors (methods) that the objects created from the class will have.

Example of a C# Class

Here's a simple example of a C# class that defines a Student with two fields:

```
public class Student
{
int id; // Field or data member
string name; // Field or data member
// Constructor to initialize the fields
public Student(int studentId, string studentName)
{
id = studentId;
name = studentName;
}
// Method to display student details
public void DisplayInfo()
{
Console.WriteLine($"ID: {id}, Name: {name}");
}
}
```

Key Components of the Class Example

Fields:

int id: Represents the student's ID.

string name: Represents the student's name.

Constructor:

public Student(int studentId, string studentName): Initializes the id and name fields when a new Student object is created.

Method:

public void DisplayInfo(): A method that prints the student's details to the console.

C# Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```
using System;
  public class Student
   {
      int id;//data member (also instance variable)
      String name;//data member(also instance variable)

    public static void Main(string[] args)
      {
         Student s1 = new Student();//creating an object of Student
         s1.id = 101;
         s1.name = "Sonoo Jaiswal";
         Console.WriteLine(s1.id);
         Console.WriteLine(s1.name);
      }

   }
```

C# Class Example 2: Having Main() in another class

Let's see another example of class where we are having Main() method in another class. In such case, class must be public.

```
using System;
public class Student
{
public int id;
public String name;
}
class TestStudent{
public static void Main(string[] args)
{
Student s1 = new Student();
s1.id = 101;
s1.name = "Sonoo Jaiswal";
Console.WriteLine(s1.id);
Console.WriteLine(s1.name);   }}}
```

C# Constructor

In C#, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C# has the same name as class or struct.

There can be two types of constructors in C#.

- Default constructor
- Parameterized constructor

C# Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

C# Default Constructor Example: Having Main() within class

```
using System;

public class Employee
{
public Employee()
{
Console.WriteLine("Default Constructor Invoked");
}
public static void Main(string[] args)
{
Employee e1 = new Employee();
Employee e2 = new Employee();
}
}
```

C# Default Constructor Example: Having Main() in another class

Let's see another example of default constructor where we are having Main() method in another class.

```
using System;
public class Employee
{
public Employee()
```

```
{
Console.WriteLine("Default Constructor Invoked");
}
}
class TestEmployee{
public static void Main(string[] args)
{
Employee e1 = new Employee();   //Default Constructor Invoked
Employee e2 = new Employee();  //Default Constructor Invoked
}
}
```

C# Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

```
1.  using System;
2.  public class Employee
3.  {
4.  public int id;
5.  public String name;
6.  public float salary;
7.  public Employee(int i, String n,float s)
8.  {
9.  id = i;
10. name = n;
11. salary = s;
12. }
13. public void display()
14. {
15. Console.WriteLine(id + " " + name+" "+salary);
16. }
17. }
18. class TestEmployee{
19. public static void Main(string[] args)
20. {
21. Employee e1 = new Employee(101, "Sonoo", 890000f);
```

```
22. Employee e2 = new Employee(102, "Mahesh", 490000f);
23. e1.display();   // 101 Sonoo 890000
24. e2.display();   // 102 Mahesh 490000
25. }
26. }
```

Destructors in C#

Introduction

In C#, a **Destructor** is a special method of a class used to destroy objects or instances of classes. Destructors are invoked automatically by the **Garbage Collector** when an instance of a class becomes unreachable, ensuring that resources are released properly.

Key Properties of Destructors:

- **Class-Only Usage:** Destructors can only be defined in classes, not in structs.
- **Single Destructor per Class:** A class can contain only one destructor.
- **Tilde (~) Operator:** Destructors are represented using the tilde (~) operator followed by the class name.
- **No Parameters or Access Modifiers:** Destructors do not accept parameters and cannot have access modifiers.
- **Automatic Invocation:** The Garbage Collector automatically calls the destructor when the class instance is no longer needed.

Syntax

The syntax for defining a destructor in a class is as follows:

```
class User
{
   // Destructor
   ~User()
   {
      // Your cleanup code here
   }
}
```

Example

Below is an example demonstrating the use of a destructor alongside a constructor:

```
using System;

class User
{
    public User()
    {
        Console.WriteLine("An instance of the class created");
    }

    // Destructor
    ~User()
    {
        Console.WriteLine("An instance of the class destroyed");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Details();
        // Force garbage collection to demonstrate destructor invocation
        GC.Collect();
        GC.WaitForPendingFinalizers();
        Console.ReadLine();
    }

    public static void Details()
    {
        // Create an instance of the class
        User user = new User();
    }
}
```

Output:

An instance of the class created
An instance of the class destroyed

Explanation:

1. **Instance Creation:**
    - When the Details method is called, a new instance of the User class is created.
    - The **constructor** (public User()) is invoked, printing:
      An instance of the class created
2. **Destructor Invocation:**
    - After the Details method completes, the user object goes out of scope and becomes eligible for garbage collection.
    - The GC.Collect() method is called to force garbage collection, ensuring that the destructor is invoked promptly.

- The **destructor** (~User()) is then called by the Garbage Collector, printing:
  An instance of the class destroyed

3. **Program Termination:**
   - The Console.ReadLine() statement keeps the console window open, allowing you to see the output.

The this Keyword in C#

Introduction

In C#, the this keyword is a special reference that points to the current instance of the class. It is commonly used to resolve naming conflicts, pass the current instance to other methods, and declare indexers. This document explores the three primary uses of the this keyword in C# with examples.

Key Uses of the this Keyword:

1. **Referring to Current Class Instance Variables:**
   The this keyword is often used to refer to the current class instance's fields or methods, especially when there is a naming conflict between instance variables and method parameters.
2. **Passing the Current Object as a Parameter:**
   this can be used to pass the current object to another method or constructor, allowing for object chaining or further operations on the same instance.
3. **Declaring Indexers:**
   The this keyword is used to define indexers in classes or structs, allowing objects to be indexed in a similar way to arrays.

Example: Referring to Current Class Instance Variables

Below is an example that demonstrates how the this keyword is used to distinguish between instance variables and method parameters when they have the same name.

Code Example

```
using System;

public class Employee
{
    public int id;
    public string name;
    public float salary;

    // Constructor using 'this' to refer to instance variables
    public Employee(int id, string name, float salary)
```

```
    {
        this.id = id;       // 'this.id' refers to the instance variable, while 'id' refers to the constructor parameter
        this.name = name;
        this.salary = salary;
    }

    // Method to display employee details
    public void display()
    {
        Console.WriteLine(id + " " + name + " " + salary);
    }
}

class TestEmployee
{
    public static void Main(string[] args)
    {
        // Creating instances of Employee
        Employee e1 = new Employee(101, "Sonoo", 890000f);
        Employee e2 = new Employee(102, "Mahesh", 490000f);

        // Displaying the details of each employee
        e1.display();
        e2.display();
    }
}
```

Output:

101 Sonoo 890000
102 Mahesh 490000

Explanation:

- **Constructor Usage:**
  The Employee class constructor accepts parameters id, name, and salary. The this keyword is used to refer to the instance variables (this.id, this.name, and this.salary) to differentiate them from the parameters with the same names.
- **Object Creation:**
  Two instances of the Employee class are created with different values, and the display() method is called to print the details of each employee.
- **Display Method:**
  The display() method outputs the id, name, and salary for each Employee instance, showing how the this keyword correctly assigns values to the instance variables.

The static Keyword in C#

Introduction

In C#, the static keyword is used to indicate that a member belongs to the type itself, rather than to any instance of the type. This means that static members are shared across all instances of a class and can be accessed without creating an object of the class. The static keyword can be applied to fields, methods, constructors, classes, properties, operators, and events.

Key Characteristics of static Members:

- **Type-Level Association:** Static members are associated with the type itself, not with any specific object instance.
- **No Instance Required:** Static members can be accessed without creating an instance of the class.
- **Memory Efficiency:** Since static members are shared among all instances, they help save memory by avoiding duplication of the same data across multiple instances.

Important Notes:

- **Non-Static Members:** Indexers and destructors cannot be declared as static.
- **Singleton Behavior:** Static fields and methods can be used to implement the Singleton design pattern, ensuring that only one instance of a class is created.

Advantages of the static Keyword

Memory Efficiency

Static members do not require an instance of the class to be accessed. This characteristic saves memory since the static members are shared across all instances of the class. As a result, the memory footprint of static members is significantly smaller compared to instance members, especially in scenarios with many instances.

Static Field

A **static field** is a variable that is shared across all instances of a class. Unlike instance fields, which get memory allocated every time an object is created, a static field is allocated memory only once and is shared among all objects of the class.

Example: Static Field in C#

```
using System;

class User
{
    // Static Variables
    public static string name, location;

    // Non-Static Variable
    public int age;

    // Non-Static Method
```

```
    public void Details()
    {
        Console.WriteLine("Non-Static Method");
    }

    // Static Method
    public static void Details1()
    {
        Console.WriteLine("Static Method");
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Creating an instance of the User class
        User u = new User();

        // Assigning value to non-static variable
        u.age = 32;

        // Calling non-static method
        u.Details();

        // Assigning values to static variables
        User.name = "Suresh Dasari";
        User.location = "Hyderabad";

        // Displaying the values of static and non-static variables
        Console.WriteLine("Name: {0}, Location: {1}, Age: {2}", User.name, User.location, u.age);

        // Calling static method
        User.Details1();

        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
}
```

Output:

Non-Static Method
Name: Suresh Dasari, Location: Hyderabad, Age: 32
Static Method

Explanation

Static Members:

- **Static Variables (name, location)**:
  These variables are shared among all instances of the User class. They are assigned and accessed using the class name (User.name, User.location) without the need for an object instance.

- **Static Method (`Details1()`)**:
  This method can be called directly using the class name (`User.Details1()`), without requiring an instance of the `User` class.

Non-Static Members:

- **Non-Static Variable (`age`)**:
  This variable is specific to each instance of the `User` class. In this example, it is assigned and accessed using the object instance `u`.
- **Non-Static Method (`Details()`)**:
  This method operates on the instance-specific data and must be called using an object instance (`u.Details()`).

Summary of Execution:

1. A `User` object `u` is created, and the non-static variable `age` is set to `32`.
2. The non-static method `Details()` is called, which prints "Non-Static Method".
3. Static variables `name` and `location` are assigned values and accessed directly via the `User` class.
4. The values of `name`, `location`, and `age` are printed to the console.
5. The static method `Details1()` is called, which prints "Static Method".

C# Static Class

Introduction

A static class in C# is similar to a regular class, but it cannot be instantiated. This means that you cannot create an object of a static class. The primary purpose of a static class is to hold static members that are globally accessible throughout the application.

Key Points

- **Only Static Members:** A static class can only contain static members (fields, methods, properties, etc.).
- **Cannot Be Instantiated:** You cannot create an instance of a static class.
- **Sealed Class:** A static class is implicitly sealed, meaning it cannot be inherited.
- **No Instance Constructors:** A static class cannot have instance constructors; it only has a static constructor (if needed).

Example

Below is an example of a static class that contains a static field and a static method.

```
using System;
public static class MyMath
{
    // Static Field
    public static float PI = 3.14f;

    // Static Method
    public static int cube(int n)
    {
        return n * n * n;
    }
}

class TestMyMath
{
    public static void Main(string[] args)
    {
        // Accessing static field
        Console.WriteLine("Value of PI is: " + MyMath.PI);

        // Accessing static method
        Console.WriteLine("Cube of 3 is: " + MyMath.cube(3));
    }
}
```

Output

Value of PI is: 3.14
Cube of 3 is: 27

Explanation

Static Field (PI)

- The field PI is a static field within the MyMath class. Since it is static, it belongs to the class itself rather than any instance of the class. It is accessible directly using the class name (MyMath.PI).

Static Method (cube)

- The method cube(int n) is a static method that calculates the cube of a given integer. Like the static field, it is accessed using the class name (MyMath.cube(3)).

Accessing Static Members

- Since MyMath is a static class, we directly use the class name to access its members. There's no need to create an instance of the class to use PI or cube.

Advantages of Using Static Classes

- **Memory Efficiency:** Since static classes cannot be instantiated, they avoid unnecessary memory usage associated with object creation.
- **Global Accessibility:** Static members are globally accessible within the application, making them convenient for utility functions and constants.
