**Concepts to C#**

**Access Modifier**

In C#, access modifiers are keywords used to specify the accessibility level of classes, methods, variables, and other members within a program. They control which other parts of the program can access these members. C# provides five access modifiers:

1. **public**: Members with this access modifier are accessible from any other part of the program, regardless of the assembly in which they reside.
2. **private**: Members with this access modifier are accessible only within the same class or struct. They are not accessible from outside the class or struct, including derived classes.
3. **protected**: Members with this access modifier are accessible within the same class or struct and by derived classes. They are not accessible from outside the class or struct.
4. **internal**: Members with this access modifier are accessible within the same assembly, but not from outside the assembly.
5. **protected** internal: This access modifier combines the behavior of both protected and internal. Members with this modifier are accessible within the same assembly or by derived classes, regardless of the assembly in which they reside.

**Data Types**

In C#, data types are used to define the type of data that a variable can hold. C# is strongly typed because it enforces type safety at compile time. Here's why:

* **Type Checking at Compile Time**: In C#, every variable and expression has a type known at compile time. The compiler verifies that types are used correctly, ensuring that operations are performed only on compatible data types. This helps catch type-related errors early in the development process.
* **No Implicit Type Conversion**: C# generally does not perform implicit type conversions between different data types unless there's a clear and safe conversion defined. This prevents accidental loss of data or unintended behavior.
* **Explicit Casting**: When you need to convert a value from one type to another, you typically need to explicitly cast it. This makes the code more explicit and reduces the likelihood of errors.
* **Strong Typing Enhances Code Readability and Maintainability**: By enforcing strict type rules, C# code tends to be more readable and understandable. It's easier to reason about the behavior of the code when you know exactly what types are involved.
* **Prevents Common Bugs**: Many common programming errors, such as passing a string where an integer is expected or performing arithmetic operations on incompatible types, are caught by the compiler in C#. This leads to more robust and bug-free software.

In C#, data types are primitive and non-primitive.

Primitive Data Types: (Pre defined)

* Primitive data types are the most basic data types provided by the programming language.
* They are built into the language and directly supported by the compiler.
* Primitive data types are used to represent simple values, such as integers, floating-point numbers, characters, and boolean values.
* They are also referred to as fundamental data types.
* Examples of primitive data types in C# include int, float, double, char, bool, etc.

Non-Primitive Data Types(user defined)

* Non-primitive data types are also known as reference types.
* They do not store the actual data; instead, they store a reference to the data's memory location.
* They are typically more complex data structures and can be composed of primitive or other non-primitive data types.
* Non-primitive data types include classes, interfaces, arrays, delegates, etc.
* They are usually user-defined types or collections provided by libraries or frameworks.
* Examples of non-primitive data types in C# include:
  + Classes
  + Interfaces
  + Arrays
  + Strings
  + Delegates

Data types in C# is mainly divided into three categories

1. Value Data Types
2. Reference Data Types
3. Pointer Data Type

**Value Data Types** : In C#, the Value Data Types will directly store the variable value in memory and it will also accept both signed and unsigned literals.

Following are different Value Data Types in C# programming language :

**->Signed & Unsigned Integral Types :** There are 8 integral types which provide support for 8-bit, 16-bit, 32-bit, and 64-bit values in signed or unsigned form.

| **Alias** | **Type Name** | **Type** | **Size(bits)** | **Range** | **Default Value** |
| --- | --- | --- | --- | --- | --- |
| sbyte | System.Sbyte | signed integer | 8 | -128 to 127 | 0 |
| short | System.Int16 | signed integer | 16 | -32768 to 32767 | 0 |
| Int | System.Int32 | signed integer | 32 | -231 to 231-1 | 0 |
| long | System.Int64 | signed integer | 64 | -263 to 263-1 | 0L |
| byte | System.byte | unsigned integer | 8 | 0 to 255 | 0 |
| ushort | System.UInt16 | unsigned integer | 16 | 0 to 65535 | 0 |
| uint | System.UInt32 | unsigned integer | 32 | 0 to 232 | 0 |
| ulong | System.UInt64 | unsigned integer | 64 | 0 to 263 | 0 |

**->Floating Point Types :**There are 2 floating point data types which contain the decimal point.

**Float:** It is 32-bit single-precision floating point type. It has 7 digit Precision. To initialize a float variable, use the suffix f or F. Like, float x = 3.5F;. If the suffix F or f will not use then it is treated as double.

**Double:** It is 64-bit double-precision floating point type. It has 14 – 15 digit Precision. To initialize a double variable, use the suffix d or D. But it is not mandatory to use suffix because by default floating data types are the double type.

| **Alias** | **Type name** | **Size(bits)** | **Range (aprox)** | **Default Value** |
| --- | --- | --- | --- | --- |
| float | System.Single | 32 | ±1.5 × 10-45 to ±3.4 × 1038 | 0.0F |
| double | System.Double | 64 | ±5.0 × 10-324 to ±1.7 × 10308 | 0.0D |

**->Decimal Types** : The decimal type is a 128-bit data type suitable for financial and monetary calculations. It has 28-29 digit Precision. To initialize a decimal variable, use the suffix m or M. Like as, decimal x = 300.5m;. If the suffix m or M will not use then it is treated as double.

| **Alias** | **Type name** | **Size(bits)** | **Range (aprox)** | **Default value** |
| --- | --- | --- | --- | --- |
| decimal | System.Decimal | 128 | ±1.0 × 10-28 to ±7.9228 × 1028 | 0.0M |

**-> Character Types** : The character types represents a UTF-16 code unit or represents the 16-bit Unicode character

| **Alias** | **Type name** | **Size In(Bits)** | **Range** | **Default value** |
| --- | --- | --- | --- | --- |
| char | System.Char | 16 | U +0000 to U +ffff | ‘\0’ |

**->Boolean Types** : It has to be assigned either true or false value. Values of type bool are not converted implicitly or explicitly (with casts) to any other type. But the programmer can easily write conversion code.

| **Alias** | **Type name** | **Values** |
| --- | --- | --- |
| bool | System.Boolean | True / False |

**->Enum:** enum stands for enumeration, it is a set of named integer constants, their default integer values start with 0, we can also set any other sequence of the values.

**-> Struct**: Structure is a value type and a collection of variables of different data types under a single unit. It is almost similar to a class because both are user-defined data types and both hold a bunch of different data types.

Important Points about Structures:

* Once the structures go out of scope, it gets automatically deallocated.
* Created much more easily and quickly than heap types.
* Using structure it become easy to copy the variable’s values onto stack.
* A struct is a value type, whereas a class is a reference type.

**Difference Between Structures and Class :**

|  |  |  |
| --- | --- | --- |
| **Category** | **Structure** | **Class** |
| **Data Type** | **Value Type** | **Reference type** |
| **Assignment Operation** | **Copies the value** | **Copies the reference** |
| **Parameterless Constructors** | **Not Allowed** | **Allowed** |
| **Inheritance** | **Not supported** | **Always supported** |

**Reference Data Types** : The Reference Data Types will contain a memory address of variable value because the reference types won’t store the variable value directly in memory.

Examples of reference types include **classes**, **interfaces**, **arrays**, **delegates**, and **strings**.

**->String :** It represents a sequence of Unicode characters and its type name is System.String. So, string and String are equivalent.

**Pointer Data Types:** Pointer types are used to store memory addresses.

They allow direct manipulation of memory, which is typically not permitted in safe managed environments like C#.

Pointers are used in unsafe code blocks and are mainly used for interoperability with unmanaged code or for specific performance optimizations.

Examples of pointer types include int\*, char\*, etc.

**Nullable types**

In C#, the compiler does not allow you to assign a null value to a variable. So, C# 2.0 provides a special feature to assign a null value to a variable that is known as the Nullable type. The Nullable type allows you to assign a null value to a variable. Nullable types introduced in C#2.0 can only work with Value Type, not with Reference Type.

The Nullable type is an instance of System.Nullable<T> struct. Here T is a type which contains non-nullable value types like integer type, floating-point type, a boolean type, etc.

Nullable<data\_type> variable\_name = null;

datatype? variable\_name = null;

You cannot directly access the value of the Nullable type. You have to use GetValueOrDefault() method to get an original assigned value if it is not null. You will get the default value if it is null. The default value for null will be zero.

**Type Casting**

Type conversion happens when we assign the value of one data type to another. There are two types:

1. Implicit Type Casting
2. Explicit Type Casting

->Implicit Type Casting / Automatic Type Conversion- It happens when:

* The two data types are compatible.
* When we assign value of a smaller data type to a bigger data type.

|  |  |
| --- | --- |
| **Convert from Data Type** | **Convert to Data Type** |
| byte | short, int, long, float, double |
| short | int, long, float, double |
| int | long, float, double |
| long | float, double |
| float | double |

Long->decimal->float

            **int** i = 57;

            // automatic type conversion

**long** l = i;

-> Explicit Type Casting- Explicit casting occurs when the conversion between data types needs to be performed manually by the programmer.

It involves converting a larger data type to a smaller data type, which could potentially lead to data loss.

To perform explicit casting, you need to use parentheses and specify the target data type.

Explicit casting may result in data loss or unexpected behavior if the value being casted exceeds the range of the target data type.

**double** d = 765.12;

            // Explicit Type Casting

**int** i = (**int**)d;

built-in methods for Type-Conversions

|  |  |
| --- | --- |
| **Method** | **Description** |
| ToBoolean | It will converts a type to Boolean value |
| ToChar | It will converts a type to a character value |
| ToByte | It will converts a value to Byte Value |
| ToDecimal | It will converts a value to Decimal point value |
| ToDouble | It will converts a type to double data type |
| ToInt16 | It will converts a type to 16-bit integer |
| ToInt32 | It will converts a type to 32 bit integer |
| ToInt64 | It will converts a type to 64 bit integer |
| ToString | It will converts a given type to string |
| ToUInt16 | It will converts a type to unsigned 16 bit integer |
| ToUInt32 | It will converts a type to unsigned 32 bit integer |
| ToUInt64 | It will converts a type to unsigned 64 bit integer |

**Boxing and unboxing**

Boxing and unboxing are concepts that involve converting a value type to a reference type (boxing) and vice versa (unboxing). These operations are necessary when dealing with certain scenarios such as storing value types in collections that require reference types or when passing value types as objects.

**Boxing:** Boxing is the process of converting a value type (such as an int, float, or struct) into a reference type (such as object or interface).

When a value type is boxed, a new object is created on the heap, and the value of the value type is copied into that object.

Boxing allows value types to be treated as objects, which is necessary for scenarios like storing them in collections such as ArrayList or passing them as parameters to methods that expect object references.

int i = 10;

object obj = i; // Boxing: int value is boxed into an object

**Unboxing:** Unboxing is the process of converting a reference type (that was previously boxed) back to a value type.

It involves extracting the value from the boxed object and assigning it back to a value type variable.

Unboxing requires explicit casting because the runtime needs to ensure that the boxed object contains a value that can be cast back to the original value type.

int j = (int)obj; // Unboxing: object is unboxed into an int

Boxing and unboxing operations involve additional overhead compared to working directly with value types.

Boxing involves allocating memory on the heap and copying the value, which can impact performance, especially in tight loops or high-performance scenarios.

Unboxing involves type checking and casting, which also adds overhead.

It's generally recommended to avoid unnecessary boxing and unboxing operations in performance-critical code.

int x = 10;

object boxed = x; // Boxing

int y = (int)boxed; // Unboxing

**Constructors**

A constructor is a special method of the class which gets automatically invoked whenever an instance of the class is created. Like methods, a constructor also contains the collection of instructions that are executed at the time of Object creation. It is used to assign initial values to the data members of the same class.

Important points to Remember About Constructors

* Constructor of a class must have the same name as the class name in which it resides.
* A constructor cannot be abstract, final, and Synchronized.
* Within a class, you can create only one static constructor.
* A constructor doesn’t have any return type, not even void.
* A static constructor cannot be a parameterized constructor.
* A class can have any number of constructors.
* Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of Constructors

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Private Constructor
5. Static Constructor

Default Constructor- A constructor with no parameters is called a default constructor. A default constructor has every instance of the class to be initialized to the same values. The default constructor initializes all numeric fields to zero and all string and object fields to null inside a class.

Parameterized Constructor- A constructor having at least one parameter is called as parameterized constructor. It can initialize each instance of the class to different values.

Copy Constructor- This constructor creates an object by copying variables from another object. Its main use is to initialize a new instance to the values of an existing instance.

Private Constructor- If a constructor is created with private specifier is known as Private Constructor. It is not possible for other classes to derive from this class and also it’s not possible to create an instance of this class.

Points To Remember :

* It is the implementation of a singleton class pattern.
* use private constructor when we have only static members.
* Using private constructor, prevents the creation of the instances of that class.

Static Constructor- Static Constructor has to be invoked only once in the class and it has been invoked during the creation of the first reference to a static member in the class. A static constructor is initialized static fields or data of the class and to be executed only once.

Points To Remember :

* It can’t be called directly.
* When it is executing then the user has no control.
* It does not take access modifiers or any parameters.
* It is called automatically to initialize the class before the first instance created.