# C# Short Notes

## C# Types: Value Types vs. Reference Types

### Value Types

* Independent Copies: Each variable holds its own copy.
* Examples: int, float, bool, char, struct
* Behavior: Changing one variable does not affect the other.
* Memory: Stored in the stack.

### Reference Types

* Shared Access: Variables point to the same object.
* Examples: class, array, string, object
* Behavior: Changing one variable affects the other.
* Memory: Object stored in the heap, reference stored in the stack.

### Nullable Value Types

* Non-nullable by default: Value types cannot be null.
* Nullable Value Types: Use ? to make value types nullable (e.g., int?).

## Null Coalescing Operator (??) in C#

* Function: Checks if the first value is null; if it is, uses the second value.
* Purpose: Ensures a fallback value to handle potential nulls, preventing errors and making code robust and readable.
* When to Use
* Default Values: Provide a default value if a variable is null.
* Simplifying Code: Reduces the need for explicit null checks, making code cleaner.
* Avoiding Null Reference Exceptions: Prevents errors from using null objects.
* Practical Scenarios
* User Inputs: Provide a default response for empty or null inputs.
* Configuration Settings: Handle missing or null settings/configuration values.
* Database Values: Manage null fields when retrieving database values.

## Datatype Conversions

### Implicit Conversion

* Done by the compiler
* No loss of information
* No exceptions thrown

### Explicit Conversion

* Required when there is potential data loss or exceptions
* Use cast operator or Convert class

## Difference between Cast Operator and Convert Class

* **Cast Operator:** No exception thrown on error
* **Convert Class:** Throws exception on error

## Parse vs. TryParse

* **Parse():** Throws exception if conversion fails
* **TryParse():** Returns bool indicating success or failure

## Reference Parameters in C#

* **Definition:**
* Pass a variable by reference using the ref keyword.
* The method can modify the original variable.
* **Syntax:**
* public void ExampleMethod(ref int param)
* {
* // Modify param
* }
* **Practical Scenarios in ASP.NET**

1. **State Management**
   * Maintain consistency in session state, query strings, or form data.
2. **Handling Large Data Models**
   * Improve performance by avoiding copying large objects.
3. **Database Operations**
   * Efficiently return multiple updated values or statuses.
4. **Caching and Optimization**
   * Manage and update cached objects efficiently.
5. **Service Layer Methods**
   * Update multiple parameters or objects within service methods.
6. **Error Handling and Logging**
   * Provide updated error or logging information to the calling methods.

## Out Parameters

**Definition**: Passes arguments by reference, allowing methods to return multiple values.

**Key Points**:

* No need to initialize out parameters before passing.
* Must assign a value before the method returns.

**Syntax Example**:

void Method(out int result) { result = 10; // Must assign a value }

int value;

Method(out value);

Console.WriteLine(value); // Outputs: 10

**Practical Scenarios in ASP.NET**:

1. **Returning Multiple Values**: Use for validation methods to return a boolean and error message.
2. **Database Operations**: Return status codes and messages from stored procedures.
3. **Parsing Methods**: Use int.TryParse and similar methods to handle input parsing.

## Parameter Array

**Definition**: Allows methods to accept a variable number of arguments.

**Key Points**:

* Only one params per method, must be the last parameter.
* Can pass an array directly as an argument.
* If no arguments are passed, the array is empty.

## Static Constructors

**Definition**: Initializes static members of a class. Called automatically before any static members are accessed or methods are called.

**Key Points**:

* No access modifiers.
* No parameters.
* Called only once.
* Automatically invoked.

**Usage**: Efficiently initialize and manage resources like settings, logging, cached data, or singleton instances, improving performance and reliability.

## Calling Parametrized Constructor from Default Constructor

**Definition**:

* **Constructor Chaining**: Using this keyword to call a parameterized constructor from a default constructor.

**Practical Scenarios in ASP.NET**:

1. **User Profiles**:
   * **Purpose**: Create default user profiles when no specific details are provided.
2. **Database Connections**:
   * **Purpose**: Use default settings for database connections when specific settings aren't provided.
3. **Logging Events**:
   * **Purpose**: Set up common log entries with default details.

## Inheritance

**Definition**: Allows one class (derived) to inherit properties and methods from another class (base).

**Types Supported**: Single, Multilevel, and Hierarchical.

**Diamond Problem**:

* **Issue**: Ambiguity in multiple inheritance scenarios where a class inherits from two classes that both inherit from a single base class.
* **C# Solution**: Uses interfaces instead of multiple class inheritance to avoid ambiguity and maintain clarity in class hierarchy and method implementation.

## Method Hiding

* Uses the new keyword.
* The base class method is hidden, not overridden.
* Method called depends on the reference type.

**Approaches to Call Hidden Methods**

1. **Using the Cast Operator**
2. **Using the base Keyword**
3. **Using a Base Class Reference**

## Polymorphism

**Types of Polymorphism**

1. **Compile-time (Static) Polymorphism**
   * **Method Overloading**: Same method name, different parameters.
   * **Operator Overloading**: Custom behavior for operators.
2. **Runtime (Dynamic) Polymorphism**
   * **Method Overriding**: Derived class provides specific implementation using virtual (base class) and override (derived class).
   * **Interface Implementation**: Different classes implement the same interface.

## Properties

* **Properties**: Encapsulate fields with get (read) and set (write) accessors.
* **Auto-Implemented**: No need for explicit backing field; use { get; set; }.
* **Read-Only**: Omit set; only get.
* **Write-Only**: Omit get; only set.
* **Computed Property**: Use get to calculate value (get { return Length \* Width; }).

## Class Vs Struct

| **Feature** | **Class** | **Struct** |
| --- | --- | --- |
| **Type** | Reference type | Value type |
| **Memory Allocation** | Allocated on the heap | Allocated on the stack or inline |
| **Inheritance** | Supports inheritance (can be derived from other classes) | Does not support inheritance (only implements interfaces) |
| **Default Constructor** | Can have a parameterless constructor | Implicit parameterless constructor (cannot define custom parameterless constructor) |
| **Immutability** | Mutable by default, can be made immutable | Often used as immutable types (using readonly fields) |
| **Copying** | Copies the reference (both variables point to the same object) | Copies the entire data (each variable has its own copy) |
| **Size** | Larger due to heap allocation and garbage collection overhead | More memory-efficient, especially for small data structures |
| **When to Use** | Use for complex data structures, polymorphism, and inheritance | Use for small, simple data structures where performance is critical |

## Interfaces

* **Definition**: An interface in C# is a contract that defines a set of methods, properties, events, or indexers that a class or struct must implement.
* **Syntax**: Interfaces are declared with the interface keyword, and implementing classes use the : symbol.
* **Multiple Implementations**: A class can implement multiple interfaces, providing flexibility and reusability in code.
* **No Implementation**: Interfaces do not provide any implementation themselves (unless using C# 8.0+ with default implementations).

**Explicit Interface Implementation**

* **Purpose**: Used when a class implements multiple interfaces that contain methods with the same signature. Explicit implementation allows the class to provide separate implementations for each interface.
* **Syntax**: The method is implemented using the InterfaceName.MethodName syntax, without an access modifier.
* **Access**: Explicitly implemented methods are only accessible through the interface reference, not directly through the class instance.

**Default Implementation in Interfaces (C# 8.0+)**

* **Definition**: Interfaces can provide default implementations for methods, allowing classes to inherit this implementation if they don't provide their own.
* **Usage**: Helps in evolving interfaces without breaking existing implementations, and reduces the need for boilerplate code in implementing classes.
* **Syntax**: Default methods are defined directly in the interface, with both the method signature and body provided.

**Handling Name Clashes with Explicit Implementation**

* **Scenario**: When a class implements two interfaces with methods having the same signature, explicit implementation is used to avoid conflicts.
* **Accessing Members**: Explicitly implemented methods can use the class's fields or properties to provide context-specific behavior for each interface.
* **Common Implementation**: If the behavior is similar across interfaces, a common private method can be used, with each interface method invoking it with specific parameters.

## Abstract Classes

**Abstract Class**:

* Declared with abstract keyword.
* Cannot be instantiated directly.
* Serves as a blueprint for derived classes.

**Abstract Methods**:

* Declared without implementation in the abstract class.
* Must be overridden in derived classes.

**Non-Abstract Methods**:

* Can have full implementations in the abstract class.
* Available for use in derived classes.

**Inheritance**:

* Derived classes must implement all abstract methods.
* If not fully implemented, the derived class must also be declared abstract.

**Use Case**:

* Used to enforce method implementation while sharing code across derived classes.

abstract class Animal {

public abstract void MakeSound(); // Abstract method

public void Sleep() { Console.WriteLine("Sleeping..."); } // Non-abstract method

}

class Dog : Animal {

public override void MakeSound() { Console.WriteLine("Woof!"); } // Implement abstract method

}