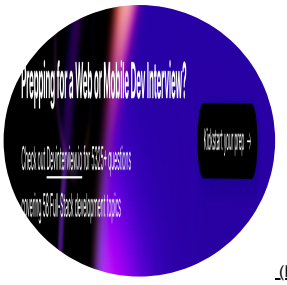
100 Must-Know C# Interview Questions



(https://devinterview.io/questions/web-and-mobile-development/)

You can also find all 100 answers here 👉 Devinterview.io - C# (https://devinterview.io/questions/web-and-mobile-development/c-sharp-interview-questions)

1. What is *C#* and what are its key features?

C#, pronounced "C-sharp," is an object-oriented, multi-paradigm programming language developed by **Microsoft** as part of its .NET initiative. It's widely used for developing applications targeting the Windows ecosystem.

Key Features of C#

1. **Simplicity**: C# streamlines complex tasks, making common programming patterns more manageable.
2. **Type Safety**: It employs a robust type system that mitigates many common programming errors at compile time.
3. **Object-Oriented Design**: It encourages modular, reusable code structures through classes and interfaces.
4. **Scalability**: C# supports both small, quick programs and large, intricate applications.
5. **Cross-Language Support**: It integrates seamlessly with other languages in the .NET ecosystem, offering advantages such as unified debugging and code reuse.
6. **Language Interoperability**: Developers can leverage code from other languages integrated into .NET.
7. **Memory Management**: It uses automatic memory management, reducing the burden of manual memory allocation and deallocation.
8. **Asynchronous Programming**: C# provides powerful tools to create responsive applications, such as the `async` and `await` keywords.
9. **Integrated Development Environment (IDE)**: Visual Studio is a robust and popular tool for C# development, offering features like IntelliSense and debugging.
10. **Library Support**: C# is backed by a vast standard library that simplifies various tasks.
11. **Modern Features**: C# continues to evolve, leveraging contemporary software development concepts and practices.
12. **Suitability for Web Development**: C# is used in conjunction with ASP.NET for developing dynamic web applications and services.
13. **LINQ (Language-Integrated Query)**: C# provides LINQ, enabling developers to query collections akin to SQL.
14. **Built-In Security Features**: C# includes functionalities such as Code Access Security (CAS) and Role-Based Security.
15. **Native Code Execution**: C# applications are executed through the Common Language Runtime (CLR), offering platform independence.
16. **Exception Handling**: It utilizes a structured error-handling approach, making it easier to reason about potential code issues.
17. **Polymorphism and Inheritance**: These essential object-oriented concepts are foundational to C#.

Memory Management in C#

C# uses the **Garbage Collector (GC)** for automatic memory management. The GC identifies and removes objects that are no longer in use, optimizing memory consumption. However, improper memory handling can lead to memory leaks or inefficient GC performance, underscoring the need for developer awareness.

Strong Types in C#

One of the defining characteristics of C# is its strong type system. It enforces type checking at compile time, reducing the likelihood of data-related errors. This stringency extends to both **primitive types** (e.g., `int`, `float`, `bool`) and **user-defined types**. C# 7.0 onward introduced **"pattern matching"**, enhancing the language's handling of types and enabling straightforward type-based operations.

Asynchronous Programming Support

C# incorporates a task-based model for efficient asynchronous execution. This approach, facilitated by the `async` and `await` keywords, mitigates thread-related overhead typically associated with multithreading, bolstering application performance.

Code Access Security

Historically, C# applications implemented Code Access Security (CAS). This feature defined and enforced permissions for varying levels of privilege within an application. CAS has been gradually phased out in newer versions of .NET, but C# remains renowned for its robust security features.

2. Explain the basic structure of a *C# program*.

Let's look at the basic structure of a C# program, including its key elements like **Namespaces**, **Classes**, **Methods**, **Variables**, **Keywords for types**, **Statements**, **Directives** and the **Main method**.

Main Program Components

- **Namespaces**: Serve as containers for related classes; they help prevent naming conflicts. The most commonly used namespace is `System`, which provides access to fundamental types, such as strings and exceptions. Examples: `System`, `System.Collections`, and `System.Text`.
- **Class**: Acts as a blueprint for objects. A program can have multiple classes, but **only one of them should have the Main method.** Such a class is referred to as the "startup" class. Here is a basic example of the structure of the startup class.

```
namespace MyApp
{
    class Program
    {
        static void Main()
        {
            // Program execution starts here.
        }
    }
}
```

- **Method**: Represents a collection of statements that are grouped together to perform a specific operation. The Main method is the entry point for a C# application.
- **Variables**: They store data that can be manipulated throughout the program. For instance, you can use a variable to keep track of score in a game or to store a user's input.
- **Console Input/Output**: The `System.Console` class provides methods to read input from and write output to the console.
- **Control Statements**: Such as `if-else` and `while`, help in altering the program's flow based on conditions.
- **Comments**: These are non-executing lines used for documentation. In C#, a single line comment begins with `//`, and a multi-line comment starts with `/*` and ends with `*/`.
- **Directives**: These are special instructions that tell the compiler to perform specific tasks. For instance, the `#include` directive is used to add the contents of a file to the source file during compilation, and `#define` is used to create symbolic constants.

Code Example: Bricks Count

Here is the C# code:

```
// Count the number of bricks in a pyramid
using System;

class Program
{
    static void Main()
    {
        Console.Write("Enter the number of levels in the pyramid: ");
        int levels = int.Parse(Console.ReadLine());

        int totalBricks = CalculateBricks(levels);
        Console.WriteLine($"Total bricks in the {levels}-level pyramid: {totalBricks}");
    }

    static int CalculateBricks(int levels)
    {
        int bricksAtBase = levels * levels;
        return (bricksAtBase * (bricksAtBase + 1) * (2 * bricksAtBase + 1)) / 6;
    }
}
```

3. What are the different types of *data types* available in *C#*?

C# features a variety of data types, each tailored to specific data needs. These types are grouped under different categories that include:

- **Value Types**: They store their own data and are distinct from one another. Examples include basic types like **int** and **float**, as well as **struct**.
- **Reference Types**: These types store references to their data. **Classes**, **arrays**, and **delegates** are classic examples.
- **Integral Types**: These are whole numbers, either signed or unsigned.
- **Floating-Point Types**: These accommodate non-integer numbers.
- **High-Precision Decimal Type**: Ideal for precise monetary and scientific calculations.
- **Boolean Type**: Represents truth values.
- **Character Types**: Specifically designed for holding characters and described in two flavors: **Uniocode** and **ASCII**.
- **Text Types**: Specifically tailored for text like strings.

Basic Types

These are the ones that are used most commonly in regular programming tasks.

`bool` (Boolean)

It signifies either **true** or **false** in logical operations.

- Storage Size: Not precise; generally 1 byte
- Default Value: **false**
- `True` Equivalents: Any non-zero numerical value; for instance, 1 is the same as **true**. Non-null object references are also **true**.
- `False` Equivalents: **0**, **null**, or an empty object reference

`char` (Character)

This represents a single Unicode character and is enclosed in **single quotes**.

- Storage Size: 2 bytes or 16 bits.
- Minimum Value: '0'
- Maximum Value: 'uffff' or '65,535'
- Default Value: **null**

`sbyte`, `byte`, `short`, `ushort`, `int`, and `uint` (Integral Types)

`byte`, `short`, and `int` are the most often used members in most applications.

Data Type	Size in Bytes	Range for Signed Values	Default Value
`sbyte`	1	-128 to 127	0
`byte`	1	0 to 255	0
`short`	2	-32,768 to 32,767	0

Data Type	Size in Bytes	Range for Signed Values	Default Value
`ushort`	2	0 to 65,535	0
`int`	4	-2,147,483,648 to 2,147,483,647	0
`uint`	4	0 to 4,294,967,295	0

`long` and `ulong`

These types are for very larger numbers, and they are usually associated with memory consumption.

- **Size in Bytes**: **8** (64 bits)
- **Range for Signed Values**: Approximately ±9.22 x 10^{18}
- **Default Value**: **0** for `long` and `ulong`

`float` and `double` (Floating-Point Types)

Both `float` and `double` can accommodate fractional values, but their precision varies. `double` allows for more significant digits.

- **Size in Bytes**: 4 for `float` and 8 for `double`
- **Precision**: 7 decimal places for `float` and 15 decimal places for `double`
- **Default Value**: 0.0

`decimal` (High-Precision Decimal Type)

This type is used for precision financial and monetary calculations.

- **Size in Bytes**: 16 (128 bits)
- **Precision**: Up to 28 decimal places for both integral and fractional parts.
- **Default Value**: 0.0M

String and Object Types

The **object** type is a **base type for all other types**, while the **string** type is specifically tailored for text storage.

- **Storage Size**: Variable
- **Example**:

```
object obj = "Hello";                    // An object
string str = "Hello";                    // A string
```

Other Types

`struct`, `enum`, `Nullable`, `tuple`, `valueTuple` are among other types in C#. **Cards.dll** is the main Dynamic-Linked Library (DLL) file for the **Solitaire** game. The game needs access to visual assets and data configurations. The **Cards.dll** file provides the visual resources and supports specific functionalities for the game, such as drawing playing cards on the screen. This DLL might also contain card-specific details like symbols, suits, or card images.

The **Cards.dll** file isn't intended for use in standalone applications but supports the Solitaire game. The DLL might include card-related resources created using tools like **Visual Studio** or third-party drawing software.

Integrating Visual Cues

Visual representations, such as card images, can enhance user experience, making the game more engaging and intuitive. The DLL is likely key to linking these visual cues with code logic.

Code Example

Here's a code snippet for providing a deck of cards using the Cards.dll:

```
// Import the Cards.dll assembly
using Cards;

// Create a new deck of cards from the DLL
Deck myDeck = new Deck();

// Deal a card
Card firstCard = myDeck.DrawACard();
```

4. What is the difference between *value types* and *reference types*?

Let's look at the difference between **Value Types** and **Reference Types**.

Key Distinctions

Data Storage

- **Value Types**: Store the data in their memory space. Examples include primitive types like integers and floating-point numbers.
- **Reference Types**: Store a reference to the data, which resides in the managed heap, and have dynamic storage size.

Memory Management

- **Value Types**: Managed by the stack. Memory is allocated and deallocated automatically within the scope of where they are defined.
- **Reference Types**: Managed by the heap. Garbage Collector automatically allocates memory and reclaims it when the object is no longer referenced.

Assignment Behavior

- **Value Types**: Directly manipulate their allocated memory. Assigning one value type to another creates an independent copy.
- **Reference Types**: Store a reference to the object's location in memory. Assigning a reference type to another only creates a new reference to the same object in memory.

Performance

- **Value Types**: Generally faster to access and manipulate because of their stack-based memory management. Ideal for smaller data types.
- **Reference Types**: Slight performance overhead due to indirection (pointer dereferencing) when accessing data in heap memory.

Nullability

- **Value Types**: Cannot be `null`. They always have a defined value.
- **Reference Types**: Can be `null`, especially if it's not been assigned an object.

Type Categories

- **Value Types**: Most primitive data types (int, float, bool, char, etc.), enums, and structs are value types.
- **Reference Types**: Classes, arrays, strings, delegates, and objects of types that are derived from the base type `object` are reference types.

Code Example: Value and Reference Types

Here is the C# code:

```
using System;

class ValueReferenceExample
{
    public static void Main(string[] args)
    {
        int val1 = 10;  // Value type: Example of a primitive type
        int val2 = val1;  // val2 is a separate copy of the value

        // Now modifying val2 will not change val1
        val2 = 20;

        Console.WriteLine("val1: " + val1 + ", val2: " + val2);

        MyStruct struct1 = new MyStruct { Value = 5 };  // Value type: Example of a struct
        MyStruct struct2 = struct1;  // struct2 is a separate copy
        struct2.Value = 10;
        Console.WriteLine("struct1: " + struct1.Value + ", struct2: " + struct2.Value);

        MyClass ref1 = new MyClass { SomeNumber = 7 };  // Reference type: Example of a class
        MyClass ref2 = ref1;  // ref2 is a reference to the same object
        ref2.SomeNumber = 15;
        Console.WriteLine("ref1: " + ref1.SomeNumber + ", ref2: " + ref2.SomeNumber);

        string str1 = "Hello, World!";  // Reference type: Example of a string
        string str2 = str1;
        str2 += " Have a great day!";  // Modifying str2 will not change str1
        Console.WriteLine("str1: " + str1 + ", str2: " + str2);
    }
}

struct MyStruct
{
    public int Value;
}

class MyClass
{
    public int SomeNumber { get; set; }
}
```

5. What are *nullable types* in *C#*?

Nullable types are a special data type in C# that can represent both a regular value and `null`. It is a beneficial feature for representing data that might be absent or unknown.

Key Considerations for Using Nullable Types

- **Memory Consumption**: Data types with explicit sizes, such as `int` or `bool`, need more space in memory when converted to nullable types.
- **Performance**: Nullable types might result in fewer optimizations, particularly with certain operations.
- **Clarity**: They offer transparency about the possible absence of a value.

The "null-conditional" operators

In C#, there are dedicated **"null-conditional" operators** caters to appropriately performing actions for nullable types, ensuring that there is no NullReferenceException. Among these are `?.`, `??`, `??=`, and `?.[]`.

Syntax Requirements for Nullable Types

- Assigning a value: `int? nullableInt = 10;`
- Assigning `null`: `nullableInt = null;`

- Performing Operations: Before utilizing the value, ensure it's not null.

```
if (nullableInt.HasValue) {
    int result = nullableInt.Value;
}
```

Common Application Scenarios

- **Database Interactions**: Adaptability to designate if a field is not set.
- **API Requests**: Effective communication to specify if the endpoint did not return a value as anticipated.
- **User Inputs**: Allowing for the perception of a lack of explicit user input.
- **Workflow Dependencies**: Identifying components that necessitate additional data to proceed.

6. Can you describe what *namespaces* are and how they are used in *C#*?

A key concept in C#, a **namespace** organizes code, allows for better code management, and prevents naming conflicts between different elements (such as classes, interfaces, and enums).

Namespaces are particularly useful in larger projects and team-based development.

Key Features

1. **Granularity**: Namespaces provide a way to group related pieces of code, making it easier to navigate your codebase.
2. **Uniqueness**: Every element in a namespace is guaranteed to have a unique name.
3. **Accessibility Control**: By default, members in a namespace are accessible only within the same namespace. You can use `public` and `internal` access modifiers to control this.

Code Example: Basic Namespace

Here is the C# code:

```
namespace MyNamespace
{
    public class MyClass { /* ... */ }

    internal class MyInternalClass { /* ... */ }

    // This interface will be accessible across the entire project
    public interface IMyInterface { /* ... */ }
}
```

Nested Namespace

You can have nested namespaces to further organize your code.

Here is the C# code:

```
namespace ParentNamespace
{
    namespace ChildNamespace
    {
        public class ChildClass { /* ... */ }
    }
}
```

Consider: ChildNamespace is distinct from a top-level namespace.

Access

Members of a **namespace** can be made more or less accessible using access modifiers.

Code Example: Access Modifiers in a Namespace

Here is the C# code:

```
namespace MyNamespace
{
    // Without access modifier: internal by default
    class MyInternalClass { /* ... */ }

    // Public access modifier: accessible from any other code in the project
    public class MyClass { /* ... */ }

    // Private access modifier: not allowed in a namespace
    // Error: 'private' is not valid in this context
    // private void MyPrivateMethod() { /* ... */ }
}
```

Using Directives

Using `using` directives, you can avoid long, repetitive code.

Code Example: Using Directives

Here is the C# code:

```
using System;
using MyNamespace;  // this is a user-defined namespace

// You can now directly use MyNamespace
class Program
{
    static void Main()
    {
        MyClass myObj = new MyClass();
        Console.WriteLine("MyObject is created");
    }
}
```

Best Practices

- **Consistency**: Decide on a consistent namespace naming convention, and ensure team members adhere to it.
- **Relevance**: Use meaningful and descriptive names for your namespaces. A good practice is a reverse domain name, like `MyCompany.MyProduct`.
- **Modularity**: Keep your namespaces well-structured and avoid making them too large, which can be counterproductive.

Common Built-In .NET Namespaces

- **System**: Fundamental types and base types.
- **System.Collections**: Implementations of standard collection classes (like lists, dictionaries, and queues).
- **System.IO**: Input and output, including file operations.
- **System.Linq**: Language Integrated Query (LINQ) functionality.

Common Namespaces

- **System.Data**: Managed code for accessing data from relational data sources.
- **System.Threading**: Enables multi-threading in applications.
- **System.Xml**: Provides support for processing XML.

7. Explain the concept of *boxing* and *unboxing* in *C#*.

Boxing is the process of converting a **value type** (struct) to a **reference type** (object) so it can be assigned to a variable of type `object` or an interface. Conversely, **unboxing** is the reverse operation.

Mechanism

- **Boxing**: When a value type is assigned to an `object`-typed variable or when it's passed as a method argument that expects an `object` or an interface type, the CLR boxes the value type, creating a reference-type object that holds the value.
- **Unboxing**: The CLR extracts the original value type from the object, provided the object actually contains such a value. A runtime error occurs if the object holds a different type.

Implications and Performance Considerations

- **Performance**: Boxing and unboxing involve overhead due to memory allocation (when boxing) and type checks during unboxing. They are, therefore, less efficient than direct interactions with value types.
- **Garbage Collection**: More aggressive memory management for short-lived objects could be necessary since **boxed objects reside on the heap**, subject to Garbage Collection.
- **Potential for Errors**: Unboxing operations might fail with an `InvalidCastException` if the object's type doesn't match the expected value type.
- **Type Mismatch**: There's a risk of unexpected behavior if types get mixed up, especially when unboxing.

Code Example: Boxing and Unboxing

Here is the C# code:

```
int number = 42;

// Boxing: Converts integer to object
object boxedNumber = number;

// Unboxing: Converts object to integer
int unboxedNumber = (int)boxedNumber;

// This will throw an InvalidCastException
// double potentialError = (double) boxedNumber;
```

8. What is *Type Casting* and what are its types in *C#*?

Type casting, in the context of C#, refers to the explicit and implicit conversion of data types. While explicit casting might lead to data loss, implicit casting is carried out by the C# compiler, ensuring safe conversion.

Implicit Casting

This form of casting is **automatic** and occurs when there is **no risk of data loss**. For instance, an `int` can be automatically cast to a `long` as there is no possibility of truncation.

Explicit Casting

This form of casting must be done manually and is necessary when there is a risk of data loss. For example, when converting a `double` to an `int`, data loss due to truncation could occur. C# requires you to explicitly indicate such conversion, and if the conversion is not possible, it throws a `System.InvalidCastException`.

Using `as` Operand

If you are sure that an object can be cast to a specific type, use the `as` operand for optimization. If the conversion is possible, the operand returns an object of the specified type; otherwise, it returns `null`.

The `is` Operator for Type Checking

With the `is` operator, you can test whether an object is compatible with a specific type. This tool is very useful to prevent `InvalidCastException` errors.

9. What are *operators* in *C#* and can you provide examples?

Operators in C# are symbols or keywords responsible for specific program actions, such as basic arithmetic, logical evaluations, or assignment.

Basic Operators

Arithmetic Operators

- **Addition**: `+`
- **Subtraction**: `-`
- **Multiplication**: `*`
- **Division**: `/`
- **Modulus**: `%` (remainder of division)

```
int result = 10 % 3;  // Output: 1 (the remainder of 10 divided by 3)
```

Comparison Operators

- **Equal to**: `==`
- **Not equal to**: `!=`
- **Greater than**: `>`
- **Less than**: `<`
- **Greater than or equal to**: `>=`
- **Less than or equal to**: `<=`

```
bool isGreater = 5 > 3;  // Output: true
```

Conditional Operators

- **AND (both conditions are true)**: `&&`
- **OR (at least one condition is true)**: `||`

```
bool conditionMet = (5 > 3) && (7 < 10);  // Output: true, both conditions are true
```

- **Ternary**: `?` and `:`

```
  int max = (5 > 3) ? 5 : 3;  // Output: 5 (if true, takes the first value, if false, takes the second value)
```

- **Null Coalescing**: `??`

```
  string name = incomingName ?? "Default Name";  // Output: The value in incomingName, or "Default Name" if incomingName is null
```

Bitwise Operators

These operators perform actions at the bit level.

- **AND**: `&`
- **OR**: `|`
- **XOR**: `^` (Exclusive OR)
- **NOT**: `~` (Unary complement)
- **Shift left**: `<<`
- **Shift right**: `>>`

```
int bitwiseAndResult = 5 & 3;  // Output: 1 (bitwise AND of 5 and 3 is 1)
```

Assignment Operators

- **Simple assignment**: `=`
- **Add then assign**: `+=`
- **Subtract then assign**: `-=`
- **Multiply then assign**: `*=`
- **Divide then assign**: `/=`
- **Modulus then assign**: `%=`
- **AND then assign**: `&=`
- **OR then assign**: `|=`

```
int num = 5;
num += 3;  // num is now 8
```

10. What is the difference between == *operator* and `.Equals()` *method*?

The == operator in C# is used to compare two objects for reference equality, meaning it checks whether the two objects are the same instance in memory. On the other hand, the `.Equals()` method is used to compare the actual values of the two objects, based on how the method is implemented for a particular class.

For value types like `int`, `bool`, `double`, and `structs`, the == operator compares the values, whereas for reference types like `string` and custom classes, it compares the references.

It's important to note that the behavior of the == operator can be overridden for reference types by overloading the operator, allowing it to compare values instead of references.

The `.Equals()` method can also be overridden in classes to provide custom equality comparisons. By default, it behaves like the == operator for reference types but can be customized to compare values instead.

11. What is the purpose of the `var` *keyword* in *C#*?

Introduced in C# 3.0 along with the Language-Integrated Query (LINQ) features, the `var` keyword primarily serves to **reduce redundancy**.

Key Benefits

- **Code Conciseness**: It streamlines code by inferring types, especially with complex types or generic data structures.
- **Dynamic and Anonymous Types Support**: It's useful when using dynamic types or initializing objects with an anonymous type; a feature useful in LINQ queries.

Avoid Misusing `var`

It's important to use `var` with caution to prevent these issues:

- **Type Clarity**: Incorporating explicit type declarations can enhance code clarity, especially for beginners or when working with inherited code.
- **Code Readability for Method Chaining**: Avoid using `var` excessively when chaining methods to maintain readability.

Best Practices and Common Use-Cases

- **Initialization**: Use `var` while initializing; it helps adapt to object type changes, reduces redundancy, and aligns with the DRY (Don't Repeat Yourself) principle.
- **For-Each Loops with Collections**: It's standard practice to use `var` in for-each loops when iterating through collections**.
- **Complex or Generic Types**: `var` brings clarity and brevity when working with such types.

Code Example: Misusing `var`

Here is the c# code:

```
var user = GetUser();
var customer = LookUpCustomer(user); // What's the type of customer?

// This would be better for clarity:
Customer customer = LookUpCustomer(user);
```

This code shows the potential confusion that can stem from not explicitly declaring types. In this example, using `var` might not communicate the "Customer" type as clearly or immediately as an explicit type declaration.

12. What are the differences between `const` and `readonly` *keywords*?

C# supports two mechanisms for creating **immutable** fields: **`const`** and **`readonly`**.

Key Distinctions

- **Initialization**: `const` fields are initialized when declared, while `readonly` fields can be initialized at the point of declaration or in the class constructor.
- **Visibility**: `readonly` fields allow for differing values within different class instances, whereas `const` ensures the same value across all instances.
- **Type Compatibility**: `const` fields are limited to primitive data types, `String`, and `null`. `readonly` can be utilized with any data type.
- **Compile-Time/Run-Time**: `const` fields are evaluated at compile time. On the contrary, `readonly` fields are evaluated at run-time, fittingly appealing to scenarios that necessitate a run-time reference or state.

Code Example: const vs. readonly

Here is the C# code:

```
public class Example {
    private readonly int readOnlyField;
    public const int constField = 5;

    public Example(int value) {
        readOnlyField = value;
    }

    public void AssignValueToReadOnlyField(int value) {
        readOnlyField = value;  // Will cause a compilation error
    }
}
```

Note that the `readOnlyField` is assigned its value either in the constructor or at declaration. Once it has a value, that value is unalterable. This invariance ensures its immutability. Likewise, `constField` is initialized with a value at the time of declaration and is unalterable for the duration of the program.

13. How does `checked` and `unchecked` *context* affect arithmetic operations?

In C#, the enablers `checked` and `unchecked` ensure more predictable behavior when using specific arithmetic operations. By default, C# employs overflow checking, but you can toggle to overflow wrapping using appropriate blocks.

Context Control Keywords

- **checked**: Forces operations to produce exceptions for overflow.
- **unchecked**: Causes operations to wrap on overflow.

When to Use Each Context

- **checked**: Ideal when you require accurate numeric results to guarantee that potential overflows are detected and handled.
- **unchecked**: Primarily used to enhance performance in scenarios where the logic or the expected input range ensures that overflows are less probable or inconsequential.

C# Code Example

Here is the C# code:

```
using System;

public class Program
{
    public static void Main()
    {
        int a = int.MaxValue;
        int b = 1;

        Console.WriteLine("Using checked context:");
        try {
            checked {
                int c = a + b;
                Console.WriteLine($"Sum: {c}");
            }
        }
        catch (OverflowException) {
            Console.WriteLine("Overflow detected!");
        }

        Console.WriteLine("\nUsing unchecked context:");
        unchecked {
            int d = a + b;
            Console.WriteLine($"Sum (unlike c#): {d}");
        }
    }
}
```

14. What are the different ways to handle *errors* in *C#*?

C# offers multiple mechanisms for handling errors, from traditional **exception handling** to contemporary **asynchronous error management** strategies.

5 Key Methods for Error Handling in C#

1. **Exception Handling**: Based on `try-catch`, primarily for synchronous flows but also compatible with some asynchronous scenarios. Exceptions are caught and processed within `catch` blocks.
2. **Logging and Monitoring**: Involves using services like `ILogger` or specialized tools to capture, evaluate, and report application errors.
3. **HTTP Status Codes**: Typically utilized in web applications to convey the status of a web request. RESTful services also often use these codes for error management.
4. **Return Values**: Methods and functions can provide customized return values to signify different error states.
5. **Task<T> and Task**: Primarily for asynchronous programming, indicating error states through `Task` results (`.Result`) or `Task<T>` members (`.IsFaulted`).

While exceptions represent unexpected errors, the other methods can be better suited for the controlled handling of anticipated issues.

For instance, a missing email in a registration form might be considered a normal scenario and should be conveyed back to the user via a **return value**, rather than triggering an exception.

15. Explain the role of the *garbage collector* in *.NET*.

The **Garbage Collector** (GC) is a key feature in **.NET** that automates memory management. Rather than requiring manual memory deallocation (as seen in languages like C/C++), the GC identifies and recovers memory that's no longer in use.

Benefits of Automatic Memory Management

- **Reduced Memory Leaks**: The GC helps prevent leaks by reclaiming memory that's no longer accessible, even if the programmer forgets to free it.
- **Simplified Memory Management**: Developers are freed from tedious tasks like memory allocation, tracking object lifetimes, and memory deallocation.
- **Protection from Use-after-Free Bugs**: Segmentation faults and other issues caused by accessing memory that's been freed are avoided.

Fundamentals of the Garbage Collector

- **Trigger Mechanism**: The GC is triggered in the background whenever certain memory thresholds are reached, reducing interruption to your program's execution.
- **Process Overview**: It divides memory into three generations, each associated with a different cleanup frequency. Most objects are initially allocated in the first generation (Gen0). When the GC cleans up Gen0, it may promote some objects to Gen1, and so forth. The idea is that, over time, surviving objects are promoted to higher generations and require less frequent cleanup.
- **Mark and Sweep**: During the cleanup cycle, the GC marks all reachable objects and then reclaims the memory occupied by objects that weren't marked. This process ensures that only inaccessible objects are collected.

Recommendations for Handling Memory in .NET

- **Limit High-Frequency Object Instantiation**: Frequent creation and destruction of small, short-lived objects can lead to inefficient GC usage. Consider using object pools for such scenarios.
- **Dispose of Unmanaged Resources**: Certain resources, like file handles or database connections, may not be managed by the GC alone. It's crucial to release them explicitly, typically by implementing the `IDisposable` pattern.
- **Consider Generational Behavior**: The GC's generational approach means that data-heavy long-lived objects (LLDs) could remain in memory for extended periods. Be mindful of LLDs' persistence and their impact on memory consumption.

Code Example: Explicit Disposal with `using`

Here is the C# code:

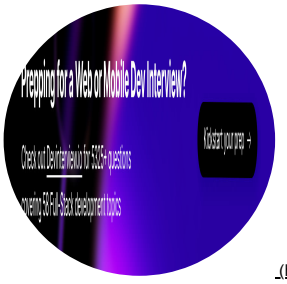
```
using System;

public class ResourceUser : IDisposable
{
    private IntPtr resource;  // Example unmanaged resource

    public ResourceUser()
    {
        resource = SomeNativeLibrary.AllocateResource();
    }

    public void Dispose()
    {
        SomeNativeLibrary.DeallocateResource(resource);
        GC.SuppressFinalize(this); // Omit if Dispose and the finalizer aren't both defined
    }
}

public static class Program
{
    public static void Main()
    {
        using(var resourceUser = new ResourceUser())
        {
            // Use the resource
        }
        // resourceUser is automatically disposed here
    }
}
```



(https://devinterview.io/questions/web-and-mobile-development/)