Writer: Kiran Kumar J

The document is to provide Frequently asked question in interviews and in meeting about C# Fundamentals and syntaxes

C# Fundamentals faqs

C# Fundamentals in FAQS

[C# Fundamentals Faq’s 1](#_Toc204461327)

[Value Type and Reference Types 1](#_Toc204461328)

[Difference between Value Tyes and Reference Types 1](#_Toc204461329)

[Can you give examples of C# built-in Value Types and Reference Types? 1](#_Toc204461330)

[Can you explain Type casting in C# programming? 2](#_Toc204461331)

[Flow Controls 3](#_Toc204461332)

[Explain the purpose of if and if-else statements. 3](#_Toc204461333)

[What are the different types of loops available in C#? 3](#_Toc204461334)

[When would you use a switch statement over an if-else if ladder? 4](#_Toc204461335)

[Can you exit a loop prematurely in C#? 5](#_Toc204461336)

[Data Collections 5](#_Toc204461337)

[What are Arrays, Lists, and Dictionaries in C#? What's the main difference between those? 5](#_Toc204461338)

[How do you perform basic CRUD operations (Create, Read, Update, Delete) on a List<T>? 7](#_Toc204461339)

# C# Fundamentals Faq’s

# Value Type and Reference Types

## Difference between Value Tyes and Reference Types

**Value Types** (e.g., int, char, struct, enum) store their actual data directly, usually on the stack.

**Reference Types** (e.g., class, string, object, array) store a reference (memory address) to their data, which is stored on the heap. Value types are copied when passed, reference types pass the reference.

## Can you give examples of C# built-in Value Types and Reference Types?

**Value Types:** int, float, double, bool, char, decimal, DateTime, struct, enum.

int number = 42; // Integral value type

double pi = 3.1415; // Floating-point value type

bool isReady = true; // Boolean value type

char letter = 'A’; // Character value type

enum Weekday { Mon, Tue, Wed }

struct Point { public int X; public int Y; }

**Reference Types:** string, object, class, array, delegate.

class MyClass

{

public int Value;

}

MyClass x = new MyClass();

x.Value = 10;

MyClass y = x; // y is a copy of the reference, not the object

y.Value = 20; // this changes x.Value as well

// x.Value is now 20

## Can you explain Type casting in C# programming?

**Type casting in C#** is the process of converting a value from one data type to another. This can be necessary when data of different types needs to be assigned or passed around in a program.

There are two main types of type casting in C#:

1. **Implicit Casting (Implicit Conversion):**
   * Automatically done by the compiler when converting a smaller or compatible data type to a larger or compatible type without loss of data.
   * Example: Converting int to long, or float to double.
   * No special syntax is required.

int myInt = 9;

double myDouble = myInt;// Implicit casting from int to double

1. **Explicit Casting (Explicit Conversion):**
   * Requires manual intervention using a cast operator because it might cause data loss or might not always succeed.
   * Typical for converting from a larger data type to a smaller one, e.g., double to int.
   * Syntax uses parentheses with the target type in front of the value.

double myDouble = 9.78;

int myInt = (int)myDouble; // Explicit cast: from double to int

# Flow Controls

## Explain the purpose of if and if-else statements.

They execute a block of code conditionally based on whether a specified Boolean expression evaluates to true or false. if-else provides an alternative path when the if condition is false.

int number = 10;

if (number > 0)

{

Console.WriteLine("The number is positive.");

}

else

{

Console.WriteLine("The number is zero or negative.");

}

## What are the different types of loops available in C#?

for loop (fixed iterations), while loop (executes as long as condition is true), do-while loop (executes at least once, then checks condition), foreach loop (iterates over elements in a collection).

**for loop** (fixed iterations) : Runs exactly 5 times, with i taking values from 0 to 4.

for (int i = 0; i < 5; i++)

{

Console.WriteLine("Iteration " + i);

}

**While loop** (Executes as long as condition is true) : Keeps running until count reaches 5

int count = 0;

while (count < 5)

{

Console.WriteLine("Count is " + count);

count++;

}

**Do-while loop** (Executes at least once and checks the condition) : Always runs the code block at least once, even if the condition is false on the first check.

int num = 0;

do

{

Console.WriteLine("Number is " + num);

num++;

} while (num < 5);

**foreach loop** (iterates over elements in a collection): Goes through each element in the colors array and prints it.

string[] colors = { "Red", "Green", "Blue" };

foreach (string color in colors)

{

Console.WriteLine("Color: " + color);

}

## When would you use a switch statement over an if-else if ladder?

A switch statement is generally preferred when you have multiple possible execution paths based on the value of a single variable, making the code more readable and often more efficient than a long if-else if chain.

int day = 3;

switch (day)

{

case 1:

Console.WriteLine("Monday");

break;

case 2:

Console.WriteLine("Tuesday");

break;

case 3:

Console.WriteLine("Wednesday");

break;

case 4:

Console.WriteLine("Thursday");

break;

default:

Console.WriteLine("Another day");

break;

}

Summary: Use a switch when you have one variable with many constant values to check, for better readability and potential performance benefits; use if-else if for complex or non-equality conditions.

## Can you exit a loop prematurely in C#?

Using the break statement to exit the loop entirely, or the continue statement to skip the rest of the current iteration and proceed to the next iteration.

int i = 0;

while (i < 10)

{

if (i == 7)

{

break; // Exit the loop entirely when i is 7

}

if (i == 5)

{

i++; // Must increment here to avoid infinite loop

continue; // Skip printing when i is 5

}

Console.WriteLine(i);

i++;

}

Console.WriteLine("While loop finished.");

# Data Collections

## What are Arrays, Lists, and Dictionaries in C#? What's the main difference between those?

**Array**: Fixed-size, ordered collection of elements of the same type.

**List**: Dynamic-size, ordered collection of elements of a specified type.

**Dictionary**: Stores key-value pairs, where each key is unique. The main difference is their flexibility in size and how elements are accessed (index vs. key).

**1.Array (Fixed size, accessed by index)**

// Declare and initialize an array of integers with fixed size

int[] numbers = new int[3];

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

// Accessing elements by index

Console.WriteLine(numbers[1]); // Output: 20

**2.List (Dynamic size, accessed by index)**

using System.Collections.Generic;

// Create a dynamic list of strings

List fruits = new List();

fruits.Add("Apple");

fruits.Add("Banana");

fruits.Add("Cherry");

// Accessing elements by index

Console.WriteLine(fruits[2]); // Output: Cherry

// Add or remove as needed

fruits.Remove("Banana");

**3.Dictionary (Key-value pairs, accessed by key)**

using System.Collections.Generic;

// Create a dictionary mapping string keys to integer values

Dictionary stock = new Dictionary();

stock["apples"] = 50;

stock["bananas"] = 30;

stock["oranges"] = 40;

// Accessing elements by key

Console.WriteLine(stock["bananas"]); // Output: 30

Different types of declarations

var dict = new Dictionary<string, int>

{

{ "apple", 3 },

{ "banana", 5 },

{ "orange", 2 }

};

Dictionary<string, int> stock = new Dictionary<string, int>

{

["apples"] = 50,

["bananas"] = 30,

["oranges"] = 50

};

## How do you perform basic CRUD operations (Create, Read, Update, Delete) on a List<T>?

**Create:** list.Add(item).

**Read:** list[index], list.ForEach(), foreach loop, list.Find().

**Update**: list[index] = newItem.

**Delete:** list.Remove(item), list.RemoveAt(index).

class Program

{

static void Main()

{

// Create - Initialize the list and add items

List<Person> people = new List<Person>();

people.Add(new Person { Id = 1, Name = "Alice" });

people.Add(new Person { Id = 2, Name = "Bob" });

people.Add(new Person { Id = 3, Name = "Charlie" });

Console.WriteLine("After Create:");

PrintPeople(people);

// Read - Access elements by index, foreach, Find

Console.WriteLine("\nReading the second person (index 1):");

Console.WriteLine($"Id: {people[1].Id}, Name: {people[1].Name}");

Console.WriteLine("\nUsing foreach to list all people:");

foreach (var person in people)

{

Console.WriteLine($"Id: {person.Id}, Name: {person.Name}");

}

Console.WriteLine("\nFind person with Id = 3:");

var personFound = people.Find(p => p.Id == 3);

if (personFound != null)

Console.WriteLine($"Found: Id {personFound.Id}, Name {personFound.Name}");

// Update - Modify the item at a specific index or found by condition

Console.WriteLine("\nUpdating person with Id = 2:");

int indexToUpdate = people.FindIndex(p => p.Id == 2);

if (indexToUpdate != -1)

{

people[indexToUpdate] = new Person { Id = 2, Name = "Robert" }; // update entire object

}

Console.WriteLine("After Update:");

PrintPeople(people);

// Delete - Remove item by value or by index

Console.WriteLine("\nDeleting person with Id = 1:");

var personToRemove = people.Find(p => p.Id == 1);

if (personToRemove != null)

{

people.Remove(personToRemove);

}

Console.WriteLine("\nDeleting person at index 1:");

if (people.Count > 1)

people.RemoveAt(1);

Console.WriteLine("After Delete:");

PrintPeople(people);

}

static void PrintPeople(List<Person> people)

{

people.ForEach(p => Console.WriteLine($"Id: {p.Id}, Name: {p.Name}"));

}

}

When would you choose a Dictionary<TKey, TValue> over a List<T>?

Key Difference in Usage

Retrieving by ID with List: Slower, must search each time.

Retrieving by ID with Dictionary: Instant lookup with the key.

Summary:

Use a Dictionary when fast key-based access is needed (like, “find by ID quickly”).

Use a List when dealing mainly with ordered collections or when keys aren’t meaningful.

using System;

using System.Collections.Generic;

class Person

{

public int Id { get; set; }

public string Name { get; set; }

}

//Using List<Person> (Good for ordered, sequential access)

class Program

{

static void Main()

{

// Create a dictionary of persons with Id as the key

Dictionary<int, Person> peopleDict = new Dictionary<int, Person>

{

[100] = new Person { Id = 100, Name = "Alice" },

[200] = new Person { Id = 200, Name = "Bob" },

[300] = new Person { Id = 300, Name = "Charlie" }

};

// Access directly by Id (very fast)

Console.WriteLine(peopleDict[200].Name); // Output: Bob

// Safe access

if (peopleDict.TryGetValue(300, out var person))

{

Console.WriteLine(person.Name); // Output: Charlie

}

}

}

//Using Dictionary<int, Person> (Good for fast retrieval by key)

using System;

using System.Collections.Generic;

class Person

{

public int Id { get; set; }

public string Name { get; set; }

}

class Program

{

static void Main()

{

// Create a dictionary of persons with Id as the key

Dictionary<int, Person> peopleDict = new Dictionary<int, Person>

{

[100] = new Person { Id = 100, Name = "Alice" },

[200] = new Person { Id = 200, Name = "Bob" },

[300] = new Person { Id = 300, Name = "Charlie" }

};

// Access directly by Id (very fast)

Console.WriteLine(peopleDict[200].Name); // Output: Bob

// Safe access

if (peopleDict.TryGetValue(300, out var person))

{

Console.WriteLine(person.Name); // Output: Charlie

}

}

}