Generics:

Generic classes and methods allow for reusability of your code by using type parameters. These allow classes and methods to defer the specification of types until the class or method is declared and instantiated.

GenericList example

```
using System;

namespace Generics
{
     public void Add(Book book)
    {
          throw new NotImplementedException();
    }

    public Book this[int index]
    {
        get { throw new NotImplementedException(); }
    }
}

public class GenericList<T>
{
    public void Add(T value)
    {
    }

    public T this[int index]
    {
        get { throw new NotImplementedException();
    }
}
```

for generics, we specify the parameter as type **T** for **Template / Type**

```
namespace Generics
{
    class Program
    {
        static void Main(string[] args)
        {
            var book = new Book { Isbn = "1111", Title = "C# Advanced" };

            var numbers = new List();
            numbers.Add(10);

            var books = new BookList();
            books.Add(book);
        }
    }
}
```

GenericDictonary example

```
public class GenericList<TKey, TValue>
{
     public void Add(TKey key, TValue value)
    {

    }
}
```

we should prefix our generic parameters with **T** but also give them a descriptive name

Delegates:

- A delegate is a reference or pointer to a function
- it is an object that knows how to call a method or group of methods

Think of delegates as a type-safe way of defining and calling callbacks. They allow methods to be passed as parameters, returned as values from other methods, and stored in variables.

Key Points about Delegates:

- **Type Safety:** Delegates are object-oriented and type-safe. They ensure the signature of the method being called is correct.
- **Multicast Capability:** A single delegate can invoke multiple methods when it's called. This is known as multicasting.
- **Anonymous Methods/Lambda Expressions:** Delegates can point to named methods, anonymous methods, or lambda expressions.

Basic Example:

Let's start with a simple example where we define a delegate to represent methods that can perform an operation on two integers and return an integer.

Delegate example

```
using System;

namespace DelegateDemo
{
    // Step 1: Declare the delegate type
    public delegate int BinaryOperation(int num1, int num2);

    class Program
    {
        static void Main()
        {
            // Step 2: Create delegate instances
            BinaryOperation addOperation = Add;
            BinaryOperation multiplyOperation = Multiply;

            // Step 3: Invoke the delegates
            Console.WriteLine(addOperation(5, 3));       // Outputs: 8
            Console.WriteLine(multiplyOperation(5, 3));  // Outputs: 15
        }

        public static int Add(int a, int b)
        {
            return a + b;
        }

        public static int Multiply(int a, int b)
        {
            return a * b;
        }
    }
}
```

In this example:

- We declare a delegate named BinaryOperation that can point to any method taking two integers as parameters and returning an integer.
- We then create instances of this delegate to point to the Add and Multiply methods.
- Finally, we invoke these delegates, which in turn call the methods they point to.

Multicast Delegates:

Delegates can point to more than one method. This is useful in scenarios like event handling.

Multicast *Delegate example*

```
public delegate void SimpleDelegate();

class Program
{
   static void Main()
   {
      SimpleDelegate del1 = Method1;
      SimpleDelegate del2 = Method2;

      // Combine delegates
      SimpleDelegate multicast = del1 + del2;

      multicast();  // Calls both Method1 and Method2
   }

   public static void Method1()
   {
      Console.WriteLine("Method1");
   }

   public static void Method2()
   {
      Console.WriteLine("Method2");
   }
}
```

Lambda Expressions:

Delegates can also work with lambda expressions, providing a concise way to define methods inline.

example 1
BinaryOperation add = (a, b) => a + b; ***Console.WriteLine(add(5, 3));*** // Outputs: 8

Delegates are foundational to many features in C#, including events and LINQ. They provide a flexible mechanism for defining and invoking methods dynamically.

Lambda Expressions:

- **Anonymous method**
  - **no access modifier**
  - **no name**
  - **no return statement**
  - **similar implementation in C++**

A lambda expression is an anonymous function that you can use to create delegates or expression tree types. It provides a concise way to represent an anonymous method without specifying a name.

The basic syntax of a lambda expression is:

(parameters)* => *expression-or-statement-block
• The lambda operator => reads as "goes to". • On the left side of the lambda operator, you specify input parameters (if any). • On the right side, you provide an expression or a statement block.

Simple Lambda Expression:
Func<int, int, int> add = (a, b) => a + b; ***Console.WriteLine(add(3, 4));*** // Outputs: 7
Here, **Func<int, int, int>** is a delegate type that represents a method that takes two int parameters and returns an int. The lambda expression **(a, b) => a + b** is a concise way to define this method.

Lambda Expression with Multiple Statements:

```
Action<string> greet = name =>
{
    string greeting = "Hello, " + name;
    Console.WriteLine(greeting);
};
greet("Alice");                          // Outputs: Hello, Alice
```

In this example, **Action<string>** is a delegate type that represents a method that takes a string parameter and returns void.

Lambda Expression in LINQ:

```
var numbers = new List<int> { 1, 2, 3, 4, 5 };
var evenNumbers = numbers.Where(n => n % 2 == 0);
foreach (var num in evenNumbers)
{
    Console.WriteLine(num);                  // Outputs: 2 and 4
}
```

Lambda expressions are frequently used in **LINQ** (Language Integrated Query) to provide concise ways to filter, order, and project data.

Here, the lambda expression **n => n % 2 == 0** is used to filter out only even numbers from the list.

Comparison with C++:

```
auto add = [](int a, int b) { return a + b; };
std::cout << add(3, 4) << std::endl;          // Outputs: 7
```

In C++, lambda expressions also use the **[ ] capture clause** to specify which outside variables are available for the lambda.

In C#, there's no need for a capture clause like in C++. Variables from the outer scope are captured automatically, and the compiler determines if they should be captured by value or by reference

Events:

Events are a way for a class to provide notifications to clients of that class when some interesting thing happens to an object. The most familiar use for events is in graphical user interfaces; typically, the classes that represent controls in the interface have events that are notified when the user does something to the control (for example, click a button).

Basics of Events:

- **Delegate:** Before understanding events, you need to understand delegates. A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type.

- **Event:** It's a way for a class to expose a delegate to other classes, but in a way that doesn't allow the other classes to replace the delegate. Only add or remove from it.

Example:

Let's say you have a **Clock** class, and you want other classes to be able to subscribe to an event that tells them the clock has ticked.

Events example

```
// Step 1: Define a delegate type
public delegate void ClockTickHandler(object sender, EventArgs e);

public class Clock
{
    // Step 2: Define an event of that delegate type
    public event ClockTickHandler Tick;

    public void Start()
```

```
    {
        while (true)
        {
            System.Threading.Thread.Sleep(1000); // Sleep for 1 second
            OnTick(); // Raise the event
        }
    }

    // Step 3: Raise the event
    protected virtual void OnTick()
    {
        if (Tick != null)
        {
            Tick(this, EventArgs.Empty);
        }
    }
}
public class ClockSubscriber
{
    public void Subscribe(Clock clock)
    {
        clock.Tick += HandleTick; // Subscribe to the Tick event
    }

    private void HandleTick(object sender, EventArgs e)
    {
        Console.WriteLine("Clock ticked!");
    }
}

public class Program
{
    public static void Main()
    {
        Clock clock = new Clock();
        ClockSubscriber subscriber = new ClockSubscriber();

        subscriber.Subscribe(clock);
```

```
        clock.Start();
    }
}
```

In this example:

- The **Clock** class has an event called **Tick**.
- The **ClockSubscriber** class subscribes to that event and provides a handler (**HandleTick**) that will be called whenever the **Clock** raises the **Tick** event.
- The **Clock** class raises the **Tick** event every second.

When you run the program, you'll see "Clock ticked!" printed to the console every second.

This is a basic example, and in real-world scenarios, you might want to provide more information with the event, like the current time, by using custom **EventArgs**.

Extension Methods:

Allows us to add methods to an existing class without changing source code or creating a new class that inherits from it. They are a special kind of static method, but they can be called as if they were instance methods on the extended type.

How to Create an Extension Method:

1. ***Static Class:*** The class that will contain the extension method must be static.
2. ***Static Method:*** The method itself must also be static.
3. ***this Keyword:*** The type that you want to extend will be specified as the first parameter of the method, preceded by the **this** keyword.

Example:

Let's say you want to add a method to the string type that checks if a string is all uppercase.

Extension method example

```
using System;

namespace ExtensionMethodsDemo
{
    public static class StringExtensions
    {
        public static bool IsAllUpperCase(this string str)
        {
            foreach (char c in str)
            {
                if (char.IsLetter(c) && char.IsLower(c))
                    return false;
            }
            return true;
        }
    }

    class Program
    {
        static void Main()
        {
            string testString1 = "HELLO";
            string testString2 = "Hello";

            Console.WriteLine(testString1.IsAllUpperCase());  // True
            Console.WriteLine(testString2.IsAllUpperCase());  // False
        }
    }
```

In the above example:

- We've created an extension method **IsAllUpperCase** for the **string** type.
- The method checks if all the characters in the string are uppercase and returns a boolean value.
- Even though this method is defined in a static class (**StringExtensions**), we can call it as if it were an instance method on a **string**.

Points to Remember:

1. **Namespace:** To use an extension method, you need to have a using directive for the namespace where the extension method is defined.

2. **No Access to Private Members:** Extension methods cannot access private variables or methods from the type they are extending.

3. **Intellisense:** Extension methods show up in Intellisense, making them easy to discover and use.

4. **Overriding:** If there's an instance method with the same signature as the extension method, the instance method will always be called, effectively "overriding" the extension method.

5. **Extension methods** are particularly powerful when combined with **LINQ**, where they provide a fluent syntax for querying collections.

LINQ:

Stands for **Language Integrated Query**, is a powerful feature in C# that allows you to query collections in a declarative manner, similar to how you'd query databases using SQL. It provides a consistent model for working with data across various kinds of data sources and formats.

LINQ introduces a set of standard query operators that can be used to query, project, and filter data in arrays, enumerable classes, XML, relational databases, and third-party data sources.

Basic LINQ Query:

A basic LINQ query involves three distinct actions:

1. **Obtain the data source.**
2. **Create the query.**
3. **Execute the query.**

Examples:

Querying a List of Integers:

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        // 1. Data source
        List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // 2. Query creation
        var evenNumbers = from num in numbers
                          where num % 2 == 0
                          select num;

        // 3. Query execution
        foreach (var num in evenNumbers)
        {
            Console.WriteLine(num);  // Outputs even numbers: 2, 4, 6, 8, 10
        }
    }
}
```

Querying a List of Strings:

```
List<string> names = new List<string> { "Alice", "Bob", "Charlie", "David" };

// Names that start with 'A' or 'B'
var filteredNames = from name in names
                    where name.StartsWith("A") || name.StartsWith("B")
                    select name;

foreach (var name in filteredNames)
{
    Console.WriteLine(name);  // Outputs: Alice, Bob
}
```

Using Method Syntax:

```
var evenNumbers = numbers.Where(n => n % 2 == 0);

foreach (var num in evenNumbers)
{
    Console.WriteLine(num);                    // Outputs even numbers: 2, 4, 6, 8, 10
}
```

LINQ also provides a method syntax (also known as extension methods) which can be more concise and sometimes more readable:

Querying with Projection:

```
var nameLengths = from name in names
                  select new { Name = name, Length = name.Length };

foreach (var item in nameLengths)
{
    Console.WriteLine($"{item.Name} has {item.Length} characters.");
}
```

You can transform the data as you query it

Ordering:

```
var orderedNames = from name in names
                   orderby name descending
                   select name;

foreach (var name in orderedNames)
{
    Console.WriteLine(name);  // Outputs names in descending order
}
```

You can order the results using **orderby**

Points to Remember:

Deferred Execution: The query variable itself only stores the query commands. The actual query execution happens when you iterate over the query variable, e.g., in a foreach loop. This is called deferred execution.

Immediate Execution: Some **LINQ** methods cause immediate query execution, like **ToArray(), ToList(), Count(),** etc.

Extensibility: LINQ is extensible. For example, **LINQ** to **SQL** allows you to query relational databases, and **LINQ** to **XML** allows you to query **XML** data sources.

In summary, **LINQ** provides a unified, concise, and readable way to query and manipulate data in C#. It integrates seamlessly with the C# language and brings the power of **SQL**-like querying directly into C#.

Nullable Types:

nullable types are a feature that allows value types (like int, double, bool, etc.) to represent a missing or undefined value, which is typically represented by the null reference in reference types.

By default, value types cannot be set to null because they have a default value. For instance, the default value of an int is 0, and the default value of a bool is false. However, there are scenarios where you might want to represent the absence of a value, and that's where nullable types come in.

Syntax:
You can declare a nullable type by appending a **?** to the type name. For example:

- **int?** represents a nullable integer.
- **bool?** represents a nullable boolean.

Examples:

Declaring and Using Nullable Types:

```
int? nullableInt = null;                    // A nullable integer set to null
double? nullableDouble = 3.14;              // A nullable double set to a value

if (nullableInt.HasValue)
{
    Console.WriteLine($"Value of nullableInt: {nullableInt.Value}");
}
else
{
    Console.WriteLine("nullableInt has no value.");       // This will be printed
}
```

Using the ?? Operator:

```
int? anotherNullableInt = null;
int regularInt = anotherNullableInt ?? 0;
```

If **anotherNullableInt** is null, **regularInt** will be set to 0

Using the GetValueOrDefault Method:

```
int? yetAnotherNullableInt = null;
int result = yetAnotherNullableInt.GetValueOrDefault();
```

Nullable types have a method called **GetValueOrDefault** that returns the value if it exists, or the default value for the underlying type if it doesn't

Nullable Types with Value Types:

```
 struct Point
{
    public int X { get; set; }
    public int Y { get; set; }
}

Point? nullablePoint = new Point { X = 1, Y = 2 };
```

All value types have a nullable counterpart. This includes custom structs

Points to Remember:

Performance: Nullable types have a slight overhead compared to regular value types because they essentially wrap the value type and a boolean flag indicating whether the value is present.

Use Cases: Nullable types are particularly useful when dealing with databases, where fields might not have a value (i.e., the value might be NULL in SQL terms).

Nullable Reference Types: Starting with C# 8.0, reference types can also be made nullable or non-nullable based on the context, but this is a different feature that aims to help developers avoid null reference exceptions.

In summary, nullable types in C# provide a way to extend value types to represent absent or undefined values, making it easier to model real-world scenarios where a value might be missing.

Dynamic:

The **dynamic** keyword was introduced with C# 4.0 as part of the language's support for dynamic typing. While C# is primarily a statically-typed language, where the type of a variable is known at compile-time, the dynamic keyword allows you to bypass compile-time type checking and resolve type information at runtime.

Key Points about dynamic:

- **Runtime Type Resolution:** The type of a **dynamic** object is determined at runtime, not at compile time.
- **No Intellisense:** When you're using **dynamic** types in Visual Studio, you won't get Intellisense support because the type information is not available until runtime.
- **RuntimeBinderException:** If you try to access a property or method on a **dynamic** object that doesn't exist, you'll get a **RuntimeBinderException** at runtime.
- **Performance Overhead:** Using **dynamic** introduces a runtime overhead because of the runtime type resolution. It's generally slower than using statically-typed variables.

Examples:

Basic Usage:
dynamic dynamicVar = 100; ***Console.WriteLine(dynamicVar);*** // Outputs: 100 ***dynamicVar = "Hello, World!";*** ***Console.WriteLine(dynamicVar);*** // Outputs: Hello, World!

Working with Objects:

```
dynamic person = new ExpandoObject();
person.Name = "Alice";
person.Age = 30;

Console.WriteLine($"{person.Name}, {person.Age} years old");

// Outputs: Alice, 30 years old
```

Using with Methods:

```
public dynamic DynamicMethod(dynamic param)
{
    return param.Length;  // Assumes param has a Length property
}

// Usage
string testString = "Hello";
Console.WriteLine(DynamicMethod(testString));  // Outputs: 5
```

Potential Pitfalls:

```
dynamic x = "Hello";
Console.WriteLine(x.NonExistentMethod( ));

// This will throw a RuntimeBinderException at runtime
```

When to Use dynamic:

While **dynamic** provides flexibility, it should be used judiciously. Here are some scenarios where **dynamic** might be useful:

- **Interacting with COM objects:** Especially when working with Office Interop, where the type might not be known at compile time.
- **Reflection:** Instead of using complex reflection code to invoke methods or access properties, dynamic can simplify the code.
- **Deserializing JSON or XML:** When you're not sure of the structure of the data you're deserializing, dynamic can be helpful.

However, in most scenarios, it's advisable to use statically-typed variables to leverage the benefits of type safety, Intellisense, and compile-time error checking.

Exception Handling:

Exception handling in C# is a mechanism to gracefully detect and handle runtime errors. It provides a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: try, catch, finally, and throw.

Key Concepts:

- **Exception:** An unwanted or unexpected event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- **Throwing an Exception:** When an error occurs within a method, the method creates an object and hands it off to the runtime system. This object, called an exception object, contains information about the error.
- **Catching an Exception:** To guard against an exception, a portion of code is placed under exception inspection. This is done using the try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception using the catch block and handle it.
- **Finally Block:** This block is used for code that must be executed regardless of whether an exception is thrown or not.

Basic Example:

```
using System;

class Program
{
    static void Main()
    {
        try
        {
            int divisor = 0;
            int result = 10 / divisor;                  // This will cause a DivideByZeroException
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine($"An error occurred: {ex.Message}");
        }
        finally
        {
            Console.WriteLine("This code will execute regardless of an exception.");
        }
    }
}
```

In the above example, a **DivideByZeroException** will be thrown, and the corresponding **catch** block will handle it.

Multiple Catch Blocks:

```
try
{
    // Some code...
}
catch (ArgumentNullException ex)
{
    // Handle ArgumentNullException
}
catch (DivideByZeroException ex)
{
    // Handle DivideByZeroException
}
catch (Exception ex)
{
    // General catch block to handle any exception not caught by the above blocks
}
```

You can have multiple catch blocks to handle different types of exceptions specifically.

Throwing Exceptions:

```
public void SetAge(int age)
{
    if (age < 0)
    {
        throw new ArgumentOutOfRangeException("Age cannot be negative.");
    }

    // Set the age...
}
```

You can also throw exceptions using the throw keyword.

Custom Exceptions:

```
public class InvalidUserAgeException : Exception
{
    public InvalidUserAgeException(string message) : base(message)
    {
    }
}

// Usage:
if (age < 0)
{
    throw new InvalidUserAgeException("Age cannot be negative.");
}
```

You can create custom exception classes by extending the Exception class.

Points to Remember:

- **Avoid Catching General Exceptions:** It's generally a bad practice to catch general exceptions unless it's for a top-level method or for logging purposes. Catching specific exceptions helps ensure you're only handling exceptions you expect and know how to handle.

- **Use finally Judiciously:** Code in the finally block will always run, so it's ideal for cleanup tasks, like closing database connections or files. However, avoid placing large amounts of code or code that can throw exceptions in a finally block.

- **Exceptions are Costly:** Throwing exceptions can be resource-intensive. It's best to use them for exceptional cases and not for regular control flow in your application.

In summary, exception handling in C# provides a robust mechanism to detect and handle runtime errors, ensuring that your application can fail gracefully and can provide useful error information.

Asynchronous programming:

Asynchronous programming is a means of parallelizing tasks in a way that can significantly improve the performance of your application, especially in applications that require many I/O-bound operations. In C#, the async and await keywords make asynchronous programming almost as straightforward as synchronous programming.

Key Concepts:

async Keyword: Indicates that a method, lambda expression, or anonymous method is asynchronous. Methods marked with async will return a Task or Task<T> (or void, but this is generally discouraged except for event handlers).

await Keyword: Used to implicitly wait for a Task or Task<T> to complete and extract its result. When await is used, it allows other parts of your application to run without blocking. Once the awaited task completes, the method resumes.

Basic Example:

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        string result = await FetchDataFromAPIAsync();
        Console.WriteLine(result);
    }

    static async Task<string> FetchDataFromAPIAsync()
    {
        using (HttpClient client = new HttpClient())
        {
            string result = await client.GetStringAsync("https://api.example.com/data");
            return result;
        }
    }
}
```

In this example:

- **FetchDataFromAPIAsync** is an asynchronous method that fetches data from an API.
- The **await** keyword in **FetchDataFromAPIAsync** means the method will return immediately, freeing up the thread to do other work. Once the **GetStringAsync** method completes, the rest of the **FetchDataFromAPIAsync** method will execute.
- The **Main** method is also asynchronous, which is a feature available starting with C# 7.1.

Benefits:

Improved Responsiveness: Asynchronous methods allow UI applications to remain responsive when they're doing work. A UI thread can start an asynchronous operation and then return immediately to process user input.

Efficient Use of Threads: Asynchronous code using **async** and **await** is more efficient in terms of threads than equivalent synchronous code. An asynchronous method doesn't occupy a thread while it's waiting for something to complete. It's returned to the thread pool until the awaited task is done.

Points to Remember:

- **Avoid async void:** Always return a **Task** or **Task<T>** from an **async** method unless you're writing an event handler. **async void** methods can't be awaited, and exceptions thrown in an **async void** method can't be caught outside of that method.

- **Use ConfigureAwait:** When you're writing library code or when you want to avoid capturing the synchronization context, use **ConfigureAwait(false)**:

```
string result =
await client.GetStringAsync("https://api.example.com/data").ConfigureAwait(false);
```

- **Exception Handling:** Use try/catch blocks within **async** methods. Exceptions thrown by an async method will be captured in the returned **Task** or **Task<T>**. To handle these exceptions, you'd typically **await** the method inside a try/catch block.

Example with Exception Handling:

```
static async Task Main()
{
    try
    {
        string result = await FetchDataFromAPIAsync();
        Console.WriteLine(result);
    }
    catch (HttpRequestException ex)
    {
        Console.WriteLine($"An error occurred: {ex.Message}");
    }
}
```

In summary, asynchronous programming in C# with **async** and **await** provides a powerful and efficient way to write programs that handle many tasks at once. It allows you to write code that's both simpler to read and often more performant, especially in I/O-bound operations.