Generics in C#

1. What are Generics?

- **Definition**: Generics allow you to define a class, method, or data structure that can work with any data type without specifying the exact type.
- **Purpose**: Introduced in .NET 2.0, generics enable type-safe and reusable code by allowing the same method or class to handle different data types.

2. Key Benefits of Generics

- **Type Safety**: Ensures that only the specified data type can be used, preventing runtime errors.
- **Code Reusability**: Reduces the need to write separate code for each data type.
- **Performance**: Eliminates boxing and unboxing, making code faster for collections and methods that handle value types.

Generic Classes and Methods

Generic Classes

- A **generic class** allows you to define a class with placeholders for data types, allowing it to handle multiple types.
- **Syntax**:

```
public class GenericClass<T>
{
        private T data;
        public GenericClass(T value)
        {
        data = value;
        }
        public T GetData()
        {
        return data;
        }
}
```

Example Usage:

```
GenericClass<int> intObject = new GenericClass<int>(5);

GenericClass<string> stringObject = new GenericClass<string>("Hello");


Console.WriteLine(intObject.GetData()); // Output: 5

Console.WriteLine(stringObject.GetData()); // Output: Hello
```

Generic Methods

- A **generic method** allows you to define a method with placeholders for data types, making it usable with various types.
- **Syntax**:

```
public void Display<T>(T value)

{

    Console.WriteLine(value);

}
```

Example Usage:

```
Display<int>(10);        // Output: 10

Display<string>("C#");   // Output: C#
```

Generics in Collections

1. Why Use Generic Collections?

- Generic collections are type-safe, efficient, and avoid the need for boxing and unboxing. Examples include `List<T>`, `Dictionary<TKey, TValue>`, and `Queue<T>`.

2. Examples of Common Generic Collections

- **List<T>**: A type-safe list that can hold any data type.

```
List<int> numbers = new List<int> { 1, 2, 3 };

numbers.Add(4);

Console.WriteLine(numbers[2]); // Output: 3
```

Dictionary<TKey, TValue>: A collection of key-value pairs, where each key is unique.

```
Dictionary<int, string> studentNames = new Dictionary<int, string>();

studentNames.Add(1, "Alice");

Console.WriteLine(studentNames[1]); // Output: Alice
```

Queue<T> and Stack<T>: FIFO and LIFO collections, respectively, useful for task management.

```
Queue<string> tasks = new Queue<string>();

tasks.Enqueue("Task 1");

tasks.Enqueue("Task 2");

Console.WriteLine(tasks.Dequeue()); // Output: Task 1
```

Constraints in Generics

What are Constraints?

- Constraints restrict the types that can be used with a generic class or method, enhancing flexibility and type safety.

2. Common Types of Constraints

- **where T : struct** — T must be a value type.
- **where T : class** — T must be a reference type.
- **where T : new()** — T must have a parameterless constructor.
- **where T : BaseClass** — T must inherit from `BaseClass`.

3. Example of Constraints

```
public class GenericRepository<T> where T : class, new()
{
    public T CreateInstance()
    {
    return new T();
    }
}
// Usage
GenericRepository<Student> repo = new GenericRepository<Student>();
Student student = repo.CreateInstance();
```

Explanation: This example restricts T to reference types that have a parameterless constructor.

Real-World Use Cases of Generics

1. Creating Type-Safe Data Structures

- Use generics for creating custom data structures like **LinkedList<T>** or **BinaryTree<T>** that can handle any data type.

2. Building Reusable Services

- Generic repositories for data access:

```
public class Repository<T> where T : class
{
    public void Add(T item) { /*...*/ }
    public T Get(int id) { /*...*/ return default(T); }
}
```

Delegates and Events with Generics

- Generic delegates make it easy to work with events and callbacks that may involve different data types.

```
public delegate void MyDelegate<T>(T item);
```

. Improving Code Flexibility and Maintenance

- Generics allow you to write adaptable code that can be updated or reused across projects, reducing redundancy and improving maintainability.

Summary of Generics in C#

1. **Key Benefits of Generics**
   - Generics provide **type safety**, **performance**, and **reusability** by allowing you to work with any data type without needing to rewrite code.
   - Generic classes and methods let you define flexible data structures and algorithms applicable to multiple data types.
2. **Practical Applications**
   - Widely used in collections (`List<T>`, `Dictionary<TKey, TValue>`), data structures, and repository patterns.
   - Ideal for creating reusable, type-safe, and efficient code across a variety of projects.
3. **Best Practices**
   - Use **constraints** to enforce type restrictions when necessary.
   - Prefer generic collections over non-generic alternatives to avoid boxing and ensure type safety.
   - Consider generics when building frameworks, libraries, or tools for broader usability.

Delegates in C#

1. What is a Delegate?

- **Definition**: A **delegate** is a type that represents a reference to a method. Delegates in C# allow you to pass methods as parameters, define callback functions, and handle events.
- **Purpose**: Delegates enable **flexibility** in code by allowing methods to be assigned to variables, passed as parameters, and executed dynamically.

2. Key Characteristics of Delegates

- Delegates are **type-safe**: The method signature (parameters and return type) must match the delegate's signature.
- Delegates allow **encapsulation of method references**.
- They support **multicasting** (i.e., one delegate can reference multiple methods).

Defining and Using Delegates

Syntax of a Delegate

- A delegate is defined with the `delegate` keyword, specifying the return type and parameters.
- **Example Syntax**:

```
public delegate void MyDelegate(string message);
```

- Here, `MyDelegate` is a delegate type that refers to methods with a `void` return type and a `string` parameter.

2. Using a Delegate

- **Assigning a Method**: Create an instance of the delegate and assign it a method that matches its signature.
- **Calling a Delegate**: Use the delegate like a method by passing parameters if required.
- **Example**:

```
public delegate void PrintMessage(string message);

public class Program

{

    public static void ShowMessage(string message)

    {

    Console.WriteLine(message);

    }

    static void Main()

    {

    PrintMessage printer = ShowMessage; // Assign ShowMessage to delegate

    printer("Hello, Delegates!");           // Invoke delegate

    }

}
```

Types of Delegates

Single-Cast Delegate

- A **single-cast delegate** points to a single method.
- **Example**: `PrintMessage` in the previous example is a single-cast delegate that refers to one method, `ShowMessage`.

2. Multicast Delegate

- A **multicast delegate** can refer to multiple methods, executing them in order.
- **Syntax**: You can use the `+=` operator to add methods and `-=` to remove methods.
- **Example**:

```
public delegate void PrintMessage(string message);

public static void ShowMessage(string message)

{

        Console.WriteLine("Show: " + message);

}

public static void LogMessage(string message)

{

        Console.WriteLine("Log: " + message);

}

static void Main()

{

        PrintMessage printer = ShowMessage;

        printer += LogMessage;          // Add LogMessage to the delegate invocation list

        printer("Hello, Multicast!"); // Calls both ShowMessage and LogMessage

}
```

Common Uses of Delegates

Delegates as Callback Functions

- Delegates allow you to pass methods as parameters, enabling callback mechanisms in asynchronous programming.
- **Example**:

```
public delegate void Callback(string result);

public static void ProcessData(Callback callback)

{       // Simulate processing

        string data = "Data processed";

        callback(data); // Invoke callback with result

}

static void Main()

{       ProcessData(result => Console.WriteLine(result));}
```

Delegates in Event Handling

- Delegates are the foundation of **event handling** in C#. Events use delegates to subscribe methods that execute when the event is raised.
- **Example**:

```
public delegate void Notify(); // Declare delegate

public class Process

{

    public event Notify ProcessCompleted; // Declare an event

    public void StartProcess()

    {

    // Process logic here...

    ProcessCompleted?.Invoke(); // Raise event

    }

}

static void Main()

{

    Process process = new Process();

    process.ProcessCompleted += () => Console.WriteLine("Process completed!");

    process.StartProcess();

}
```

Built-In Delegates in C#

1. Action Delegate

- **Definition**: Represents a method that returns `void` and can take 0 to 16 parameters.
- **Example**:

```
Action<string> showMessage = message => Console.WriteLine(message);

showMessage("Hello, Action!"); // Output: Hello, Action!
```

Func Delegate

- **Definition**: Represents a method that returns a value and can take 0 to 16 parameters.
- **Syntax**: `Func<T1, T2, ..., TResult>`
- **Example**:

```
Func<int, int, int> add = (a, b) => a + b;

Console.WriteLine(add(5, 10)); // Output: 15
```

Predicate Delegate

- **Definition**: Represents a method that returns `bool` and takes a single parameter.
- **Example**:

```
Predicate<int> isEven = num => num % 2 == 0;

Console.WriteLine(isEven(4)); // Output: True
```

Summary of Delegates in C#

1. **Key Points**
    - Delegates are type-safe references to methods.
    - Single-cast delegates refer to one method, while multicast delegates can refer to multiple methods.
    - Delegates are the foundation of event handling in C#.
2. **When to Use Delegates**
    - Use delegates for callback functions, event handling, or passing methods as parameters.
    - Built-in delegates (`Action`, `Func`, and `Predicate`) simplify working with delegates.
3. **Best Practices**
    - Prefer built-in delegates (`Action`, `Func`, `Predicate`) when applicable.
    - Use lambda expressions for concise and readable delegate assignments.
    - Delegate chaining (multicast delegates) can be useful for executing multiple methods but be mindful of the execution order and potential side effects.