.NET

Func/Action/Predicate Delegates

1. Evolution

1.1 Traditional Delegates

- Since C# 1.0 (2002)
- **Type-safe function pointers** that reference methods with specific signatures.
- Must be declared before use i.e. Required explicit delegate type declarations.
- Example:

```
delegate int MathOperation(int a, int b);  // Custom delegate type
int Add(int a, int b) {
  return a + b;
}
MathOperation add = new MathOperation(Add);
```

1.2 Generic Delegates

- Since C# 2.0
- Allows generic parameters
- Examples:

```
delegate void Consumer<T>(T obj);
```

```
Consumer<string> print = Console.WriteLine;
print("Hello, World");
```

1.3 Pre-defined Generic Delegates

- In C# 3.0 (2007)
- Introduced `Func`, `Action`, and `Predicate` to reduce boilerplate.
- Part of the **System** namespace.

2. Generic Delegates (`Func`, `Action`, `Predicate`)

Delegate	Purpose	Signature
`Action`	For void methods	`Action<T1, T2>`
`Func`	For methods with return values	`Func<T1, T2, TResult>`
`Predicate`	For boolean conditions	`Predicate<T>`

2.1 `Action` Delegate

- **Purpose**: For methods that **return void**.
- **Overloads**: `Action`, `Action<T>`, `Action<T1,T2>`, ..., up to 16 parameters.
- **Example**:

```
Action<string> log = Console.WriteLine;
log("Hello, Action!");
```

2.2 `Predicate` Delegate

- **Purpose**: For methods that **return a boolean** i.e. test a condution.

- Legacy delgate: largely replaced by `Func<T, bool>`.
- **Example**:

```
boolean IsEven(int x) {
  return x % 2 == 0;
}
Predicate<int> condition1 = IsEven;
bool flag1 = condition1(4);  // true
Predicate<int> condition2 = x => x % 2 != 0;
bool flag2 = condition2(4);  // false
```

2.3 `Func` Delegate

- **Purpose**: For methods that **return a value**.
- **Last type parameter**: Return type.
- **Example**:

```
int Add(int x, int y) {
  return x + y;
}
Func<int, int, int> add = Add;
int result = add(5, 7);  // 12
Func<int, int, int> subtract = (x,y) => x - y
result = subtract(7, 2); // 5
```

2.4 Comparison with Java's Functional interfaces

C# Delegate	Java Functional Interface	Description

C# Delegate	Java Functional Interface	Description
`Action<T>`	`Consumer<T>`	Method with no return value (void)
`Func<TResult>`	`Supplier<TResult>`	Method with no arguments, returns a value
`Func<T, TResult>`	`Function<T, TResult>`	Method with one argument, returns a value
`Func<T1, T2, TResult>`	`BiFunction<T1, T2, TResult>`	Method with two arguments, returns a value
`Predicate<T>`	`Predicate<T>`	Method with one argument, returns boolean

3. When to Use Which

3.1 Prefer `Func`/`Action` When:

- Needing **quick, inline & standard delegate definitions**.
- Working with **lambda expressions**.
- Using **LINQ** or functional programming patterns.

3.2 Prefer Traditional Delegates When:

- Defining **event handlers** (e.g., `public delegate void EventHandler()`).
- Requiring **explicit naming** for API clarity.
- Handling **specialized signatures** (e.g., `ref`/`out` parameters).
- **Specialized signatures** not covered by `Func`/`Action`.

4. Performance Considerations

- **No runtime performance difference**: All delegates compile to similar IL.
- **Memory**: Traditional delegates may marginally increase metadata size.

5. Best Practices

- ✔ **Use `Func`/`Action`** for lambda expressions and LINQ.
- ✔ **Reserve traditional delegates** for events and public APIs.
- ✔ **Replace `Predicate<T>`** with `Func<T, bool>` for consistency.

6. MSDN References

- Func Delegate
- Action Delegate

Events

1. What are Events?

1.1 Definition

- **Events** are a language-level implementation of the **Observer pattern**, enabling objects to notify others when something happens.
- Built on top of **delegates**, but with added encapsulation and safety.

1.2 Key Characteristics

- ✔ **Publisher-Subscriber Model**:
  - **Publisher**: Raises the event.
  - **Subscriber**: Responds to the event.
    - ✔ **Decoupled Communication**: Publishers don't need to know about subscribers.

2. Event Declaration and Usage

2.1 Step 1: Define a Delegate

```
public delegate void PriceChangedHandler(decimal oldPrice, decimal newPrice);
```

2.2 Step 2: Declare the Event

```
public class Stock {
    // Event declaration (based on the delegate)
    public event PriceChangedHandler PriceChanged;

    private decimal _price;
    public decimal Price {
        get => _price;
        set {
            if (_price == value) return;
            decimal oldPrice = _price;
            _price = value;
            PriceChanged?.Invoke(oldPrice, _price);  // Raise event
        }
    }
}
```

2.3 Step 3: Subscribe to the Event

```
var stock = new Stock { Price = 100 };

// Subscribe
stock.PriceChanged += (oldPrice, newPrice) =>
    Console.WriteLine($"Price changed from {oldPrice} to {newPrice}");

// Trigger event
stock.Price = 150;  // Output: "Price changed from 100 to 150"
```

3. Built-in `EventHandler` Delegate

3.1 Standard Pattern

- Instead of custom delegates, use .NET's standard:

```
public event EventHandler<PriceChangedEventArgs> PriceChanged;
```

- Requires an `EventArgs` subclass:

```
public class PriceChangedEventArgs : EventArgs
{
    public decimal OldPrice { get; }
    public decimal NewPrice { get; }
    public PriceChangedEventArgs(decimal oldPrice, decimal newPrice)
    {
        OldPrice = oldPrice;
        NewPrice = newPrice;
    }
}
```

3.2 Usage with `EventHandler`

```
public class Stock
{
    public event EventHandler<PriceChangedEventArgs> PriceChanged;

    protected virtual void OnPriceChanged(PriceChangedEventArgs e)
    {
        PriceChanged?.Invoke(this, e);  // 'this' is the sender
```

```
    }

    public decimal Price
    {
        set
        {
            if (_price != value)
            {
                OnPriceChanged(new PriceChangedEventArgs(_price, value));
                _price = value;
            }
        }
    }
}
```

4. Event Accessors (`add`/`remove`)

- Override default event behavior (e.g., thread-safe subscriptions):

```
private EventHandler _myEvent;
public event EventHandler MyEvent
{
    add => _myEvent += value;
    remove => _myEvent -= value;
}
```

5. Best Practices

✔ **Use `EventHandler<T>` for consistency.**
✔ **Always check for `null` before invoking**:

```
PriceChanged?.Invoke(this, EventArgs.Empty);
```

✔ **Prefix events with** `On` for raising methods (`OnPriceChanged`).
✔ **Avoid long-running subscribers** (can block publishers).

6. Common Pitfalls

✘ **Memory leaks** (forget to unsubscribe).

```
// Unsubscribe when done:
stock.PriceChanged -= HandlePriceChange;
```

✘ **Overusing events** (simple callbacks may suffice).

7. MSDN References

- Events (C#)
- EventHandler Delegate

Arrays

Introduction

- Arrays represent one of the most fundamental data structures in C#.
- They provide efficient, fixed-size collections of elements that are stored contiguously in memory, offering O(1) access time to elements via indexing.
- The .NET Framework implements arrays as objects derived from System.Array, making them reference types that inherit all members from this base class.

Array Types and Declaration

1. Single-Dimensional Arrays

- The simplest array form stores elements in a linear sequence:

```
int[] numbers = new int[5]; // Declaration with size
string[] names = { "Alice", "Bob", "Charlie" }; // Initialization
// traverse array using index
for(int i=0; i<names.Length; i++)
    Console.WriteLine(names[i]);
// OR - arrays are IEnumerable - traverse using foreach loop
foreach(string name in names)
    Console.WriteLine(name);
```

2. Multi-Dimensional Arrays

- Support matrix-like structures with rectangular:

```
//int[,] matrix = new int[3, 3]; // 3x3 matrix
int[,] matrix = new int[3, 3] {
        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    };
// access the members
for (int i = 0; i < matrix.GetLength(0); i++) {
    for (int j = 0; j < matrix.GetLength(1); j++)
        Console.Write(matrix[i, j] + " ");
    Console.WriteLine();
}
```

- Also supports jagged arrays.

```
int[][] jagged = new int[3][]; // Jagged (array of arrays)

// Initialize each row with an array of a specific length
jagged[0] = new int[] { 1, 2, 3 };
jagged[1] = new int[] { 4, 5 };
jagged[2] = new int[] { 6, 7, 8, 9 };

// Print the elements of the jagged array
for (int i = 0; i < jagged.Length; i++) {
    for (int j = 0; j < jagged[i].Length; j++)
        Console.Write(jagged[i][j] + " ");
    Console.WriteLine();
}
```

3. Specialized Array Types

- The .NET ecosystem includes optimized array variants:
  - `Buffer` for primitive type operations
  - `ArraySegment<T>` for partial array views
  - `Memory<T>` and `Span<T>` for safe memory access

Memory Structure and Performance

1. Storage Characteristics

- Arrays allocate:
  - Contiguous memory blocks on the managed heap
  - Header information including length and sync block
  - Direct element storage (value types) or references (reference types)

2. Access Patterns

- Provide constant-time O(1) for:
  - Index-based read/write operations
  - Length lookup via Length property
  - Boundary checking (with runtime validation)

3. Performance Considerations

- Optimized for:
  - CPU cache locality (sequential access)
  - Vectorized operations via SIMD
  - Low-overhead iteration constructs

System.Array Functionality

1. Core Members

- All arrays inherit essential methods:
  - `Length` property (total element count)
  - `Rank` property (number of dimensions)
  - `Clone()` method (shallow copy)
  - `GetValue()`/`SetValue()` reflection-style access

2. Sorting and Searching

- Static methods provide common algorithms:

```
Array.Sort(myArray); // Array elements should be IComparable<>
```

```
int index = Array.BinarySearch(sortedArray, value);
```

3. Advanced Operations (Refer Docs)

- Constrained copying (CopyTo)
- Value initialization (Clear, Fill)
- Structural operations (Reverse, Resize)

Span and Memory

- Introduced in .NET Core 2.1 for:
    - Stack-allocated array slices
    - Unified memory access patterns
    - Reduced allocation overhead
    - Internally "ref struct" so can be used locally in methods (not as fields).
- Span Example:

```
using System;

public class SpanExample {
    public static void Main(string[] args) {
        // Create a span from an array
        int[] numbers = { 1, 2, 3, 4, 5 };
        Span<int> numberSpan = numbers.AsSpan();

        // Modify the span, which modifies the underlying array
        numberSpan[0] = 10;
        Console.WriteLine($"First element after modification: {numbers[0]}"); // Output: 10

        // Span can also be created using stackalloc
        Span<int> stackSpan = stackalloc int[3] { 6, 7, 8 };
        foreach (int number in stackSpan) // Iterate through span
            Console.WriteLine(number); // Output: 6, 7, 8
```

```
        }
    }
```

Best Practices and Guidelines

1. Size Management

- Prefer known sizes at creation
- Use List for dynamic sizing
- Consider stackalloc for small temporary arrays

2. Type Safety

- Avoid object[] for heterogeneous data
- Prefer generic methods when possible
- Validate array parameters in public APIs

3. Performance Optimization

- Minimize bounds checking in hot paths
- Use fixed buffers for interop scenarios
- Consider parallel operations for large arrays

MSDN References

- Arrays (C# Programming Guide)
- System.Array Class
- Memory and Span

.NET Collections

Introduction

- The initial non-generic collections (ArrayList, Hashtable) were replaced by type-safe generic collections in .NET 2.0 (2005), with further optimizations in .NET Core and modern .NET versions.
- The System.Collections and System.Collections.Generic namespaces contain the core collection types that form the foundation of data management in .NET applications.

Collection Overview

1. **Key Interfaces**
    - IEnumerable: Foundation for iteration capability
    - ICollection: Adds modification operations
    - IList: Provides index-based access
    - IDictionary<TKey,TValue>: Key-value pair storage
    - ISet: Mathematical set operations
2. **Primary Implementations**
    - List: Dynamic array implementation
    - Dictionary<TKey,TValue>: Hash table implementation
    - Queue/Stack: FIFO/LIFO structures
    - LinkedList: Doubly-linked list
    - HashSet/SortedSet: Optimized set operations
3. **Specialized Collections**
    - SortedList<TKey,TValue>: Hybrid list/dictionary
    - BlockingCollection: Thread-safe producer/consumer
    - ImmutableCollections: Thread-safe persistent structures

- Refer hierarchy diagram as well.

Core Collection Types

1. List

The most commonly used collection representing a dynamically resizable array:

- Implements IList, ICollection, IEnumerable
- O(1) indexed access
- O(n) insertion/removal (except at end)
- Automatically handles capacity growth
- Example:

```
List<string> names = new List<string>();
names.Add("Alice");
names.AddRange(new[] {"Bob", "Charlie"});
foreach (string item in names)
    Console.WriteLine(item);
```

2. Dictionary<TKey,TValue>

- Hash-table based key-value store.
- Near O(1) lookup/insert/delete
- Requires good hash distribution
- Implements IDictionary<TKey,TValue>
- Example

```
Dictionary<int, string> employees = new();
employees.Add(101, "John Doe");
employees[102] = "Jane Smith";
int key = 101; // input from user
string name = employees[key];
System.Console.WriteLine(name);
```

3. HashSet

- Contains unique elements only
- O(1) membership testing
- Union/Intersect/Except operations
- Example:

```
HashSet<int> primes = new() { 2, 3, 5, 7 };
primes.Add(11); // Returns true
primes.Add(5);  // Returns
foreach (int item in primes)
    Console.WriteLine(item);
```

4. Queue and Stack

- Queue: Enqueue/Dequeue operations - FIFO collection
- Stack: Push/Pop operations - LIFO collection
- Both provide O(1) operations
- Example:

```
Queue<string> requests = new();
requests.Enqueue("Request1");
string next = requests.Dequeue();
```

Performance Characteristics

1. Time Complexity Considerations

- Array-based (List): Fast indexed access

- Node-based (LinkedList): Efficient inserts
- Hash-based (Dictionary): Best for lookups

2. Memory Overhead Analysis

- Reference types add per-element overhead
- Capacity vs Count management
    - always Capacity >= Count.
    - list.EnsureCapacity(min); // ensures min capacity, doing reallocation if needed.
    - list.TrimAccess(); // release excess allocation, so that Capacity = Count.

3. Thread Safety Options

- Concurrent collections (ConcurrentDictionary)
- Synchronization wrappers (lock statements)
- Immutable collections

Specialized Collection Types

1. Sorted Collections

- SortedList<TKey,TValue>: Memory-efficient ordered dictionary
- SortedSet: Balanced tree implementation
- SortedDictionary<TKey,TValue>: Faster inserts than SortedList

2. Concurrent Collections

- BlockingCollection: Producer/Consumer patterns
- ConcurrentBag: Unordered thread-safe collection
- ConcurrentQueue/Stack: Lock-free implementations

3. Immutable Collections

- Thread-safe by design
- Structural sharing for efficiency
- Builder pattern for batch modifications
- Example

```
using System;
using System.Collections.Immutable;
public class ImmutableListExample {
    public static void Main(string[] args) {
        // Create an immutable list
        ImmutableList<string> colors = ImmutableList.Create("Red", "Green", "Blue");
        // Iterate and print the original list
        Console.WriteLine("Original List:");
        foreach (var color in colors)
            Console.WriteLine(color);
        // Create a new list by adding an item
        ImmutableList<string> newColors = colors.Add("Orange");
        // Print the new list
        Console.WriteLine("\nNew List (with Orange added):");
        foreach (var color in newColors)
            Console.WriteLine(color);
    }
}
```

Best Practices

1. Collection Selection Guidelines

- Prefer generic collections over non-generic
- Choose based on access patterns

1. Access by Index: ArrayList (non-generic), List<>
2. Access by Index with fast Insert/Delete (in middle): LinkedList<>
3. FIFO: Queue<>
4. LIFO: Stack<>
5. Thread Safety: ConcurrentQueue<>, ConcurrentStack<>, ConcurrentBag<>, ...
6. Immutability: ImmutableList<>, ImmutableQueue<>, ImmutableStack<>, ...
7. Key-Value Pairs (Fast lookup): Dictionary<TKey, TValue>, SortedList<TKey, TValue>, NameValueCollection.
8. Uniques: HashSet<>
9. Fixed size, Index access: Arrays

2. Capacity Management

- Pre-size collections when possible
- Monitor growth operations
- Trim excess memory when appropriate

3. Enumeration Safety

- Avoid modification during enumeration - a modification will throw InvalidOperationException.
- Use snapshots when needed

```
List<string> myList = new List<string> { "apple", "banana", "cherry" };

foreach (string item in myList.ToList()) // Creates a copy
{
    if (item == "banana")
        myList.Remove(item); // Modifies the original, but not the copy
}
```

Modern Enhancements and Features

1. Span and Memory Support

- Slice operations on arrays/lists
- Stack-only allocation options
- Unified processing of different memory types

2. Collection Expressions (C# 12)

Simplified initialization syntax:

```
List<int> numbers = [1, 2, 3, 4, 5];
int[] moreNumbers = [..numbers, 6, 7, 8];
```

```
Dictionary<string, int> dict = new() { ["a"] = 1, ["b"] = 2 };
```

3. Performance-Optimized Variants

- Pooled collections
- Struct-based implementations
- SIMD-accelerated operations

MSDN References

- Collections (C# Programming Guide)
- System.Collections.Generic
- Choosing a Collection Class

Class Member Access specifiers

1. **public**: Members declared with public are accessible from anywhere, without any restrictions.
2. **private**: Members declared with private are only accessible from within the same class where they are defined. This is the default access level for class members if no access specifier is specified.
3. **protected**: Members declared with protected are accessible from within the same class and from derived classes (classes that inherit from the current class).
4. **internal**: Members declared with internal are accessible only from within the same assembly (project or DLL).
5. **protected internal**: Members declared with protected internal are accessible from within the same assembly or from derived classes, even if they are in a different assembly.