

# C# Lambdas

Inline, anonymous functions for concise, expressive code

Press Space for next page →

# Overview

Lambdas are anonymous functions you can pass around like values.

```
// (parameters) => expression  
// (parameters) => { statements; return value; }  
Func<int, int> doubleIt = x => x * 2;  
Action<string> log = m => Console.WriteLine(m);
```

# Action vs Func

`Action` delegates return void; `Func` delegates return a value.

```
// Action: no return value
Action sayHi = () => Console.WriteLine("Hi");
Action<string> logMsg = msg => Console.WriteLine(msg);
Action<int, int> addLog = (a, b) => Console.WriteLine(a + b);

// Func: last generic type is the return type
Func<int> getRandom = () => Random.Shared.Next();
Func<int, int> square = x => x * x;
Func<int, int, int> add = (a, b) => a + b;

// Predicate<T> is a shorthand for Func<T, bool>
Predicate<string> nonEmpty = s => !string.IsNullOrEmpty(s);
```

Guidelines:

- Use `Action` for side effects
- Use `Func` when you need a result

# How To Call / Use

Assign lambdas to variables or pass them directly to methods.

```
// Assign to a variable
Func<int, bool> isOdd = n => (n & 1) == 1;
Console.WriteLine(isOdd(3)); // True

// Pass inline
var top3 = scores
    .OrderByDescending(s => s)
    .Take(3)
    .ToList();

// With events
button.Click += (s, e) => Console.WriteLine("Clicked!");

// Closures (captures external variable)
var threshold = 10;
var big = nums.Where(n => n > threshold).ToList();
threshold = 5; // capture is by reference; later changes affect calls
```

Tips:

- Prefer expression bodies for short logic

# Why They Exist

Lambdas make code shorter, clearer, and more composable.

- Express callbacks and small functions inline (no separate method)
- Power functional-style APIs (LINQ: `Select` , `Where` , etc.)
- Enable event handlers and async continuations
- Capture context via closures for flexible behavior
- Back expression trees for providers like Entity Framework

```
var evens = numbers.Where(n => n % 2 == 0).ToList();  
var names = people.Select(p => p.Name).ToArray();
```

# Exercise: CreateAdder

Return a function that adds `a` to its input using a closure.

Goal:

- Implement `CreateAdder(int a) => Func<int,int>` so that `CreateAdder(5)(10) == 15` .
- Hint: Return a lambda that captures `a` .

```
// Signature
public static Func<int, int> CreateAdder(int a)
{
    // TODO: return x => a + x;
    throw new NotImplementedException();
}
```

# Solution: CreateAdder

```
public static Func<int, int> CreateAdder(int a)
{
    return x => a + x; // captures 'a' in the closure
}

// Usage
var add5 = CreateAdder(5);
Console.WriteLine(add5(10)); // 15
```

# Exercise: SortByLength

Sort strings by length ascending using a lambda key selector.

Requirements:

- If `items` is `null`, return `Enumerable.Empty<string>()`.
- Use `OrderBy` with a lambda selecting the length.

```
public static IEnumerable<string> SortByLength(IEnumerable<string> items)
{
    if (items == null) return Enumerable.Empty<string>();
    // TODO: return items.OrderBy(s => s.Length);
    throw new NotImplementedException();
}
```

# Solution: SortByLength

```
public static IEnumerable<string> SortByLength(IEnumerable<string> items)
{
    if (items == null) return Enumerable.Empty<string>();
    return items.OrderBy(s => s?.Length ?? 0); // handle possible null elements safely
}

// Usage
var sorted = SortByLength(new[] { "pear", "fig", "pineapple" });
// fig, pear, pineapple
```

# Exercise: TransformToUpper

Transform each string to upper-case using `Select` and a lambda.

Requirements:

- If `items` is `null`, return `Enumerable.Empty<string>()`.
- Prefer `ToUpperInvariant()` for culture-agnostic behavior.

```
public static IEnumerable<string> TransformToUpper(IEnumerable<string> items)
{
    if (items == null) return Enumerable.Empty<string>();
    // TODO: return items.Select(s => s.ToUpperInvariant());
    throw new NotImplementedException();
}
```

# Solution: TransformToUpper

```
public static IEnumerable<string> TransformToUpper(IEnumerable<string> items)
{
    if (items == null) return Enumerable.Empty<string>();
    return items.Select(s => s?.ToUpperInvariant());
}

// Usage
var upper = TransformToUpper(new[] { "a", "Bc" });
// A, BC
```

# Exercise: MakePrinter

Return an `Action<string>` that prints `prefix + message` .

Requirements:

- Capture `prefix` in the returned lambda.
- Use `Console.WriteLine` to print.

```
public static Action<string> MakePrinter(string prefix)
{
    // TODO: return message => Console.WriteLine($"{prefix}{message}");
    throw new NotImplementedException();
}
```

# Solution: MakePrinter

```
public static Action<string> MakePrinter(string prefix)
{
    return message => Console.WriteLine($"{prefix}{message}");
}

// Usage
var warn = MakePrinter("[WARN] ");
warn("Disk space low"); // [WARN] Disk space low
```

# Exercise: Multiply (Expression-bodied)

Provide an expression-bodied property that multiplies two numbers.

Goal:

- Implement `Multiply` as `Func<int,int,int>` using a concise lambda.

```
public static Func<int, int, int> Multiply
// TODO: => (a, b) => a * b;
=> throw new NotImplementedException();
```

# Solution: Multiply (Expression-bodied)

```
public static Func<int, int, int> Multiply => (a, b) => a * b;
```

```
// Usage
```

```
Console.WriteLine(Multiply(3, 4)); // 12
```