Day 3 — Methods & Members

Agenda

- Method Declaration and Syntax
- Method Parameters and Return Types
- Method Overloading
- Named and Optional Parameters
- out and ref Parameters
- params Keyword
- Method Invocation
- Access Modifiers (public, private, protected, internal, protected internal)
- Method Scope
- Static Methods
- Extension Methods
- Local Functions
- Expression-bodied Members

--

1. Method Declaration and Syntax

Definition: A method is a named block of code that performs an action or computes and returns a value.

Short description: Methods encapsulate behavior in types (classes/structs). They have an access modifier, optional `static`, a return type (or `void`), a name, and an optional parameter list.

Variations:

- Instance methods
- Static methods
- Expression-bodied methods

Example:

```
public class Calculator
{
    // declaration: access static return name(parameters)
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

Talking points / Exercises:

- Show anatomy: modifier, return, name, params, body.
- Ask: How does return type affect callers?
- Exercise: Convert `Add` to an expression-bodied method.

References / Diagram:

- Microsoft: Access modifiers & members — https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/members

---

2. Method Parameters and Return Types

Definition: Parameters are inputs to methods; the return type is the type of value a method yields (or `void`).

Short description: Parameters can be value types, reference types, `ref` / `out`, `in`, optional, or `params`. Returns can be single values, tuples, or `Task` for async.

Variations:

- Value vs reference type parameters
- Multiple return patterns: single value, `ValueTuple`, `Task<T>`

Example:

```
public (int sum, int count) Summarize(int[] items)
{
    int sum = 0;
    foreach (var i in items) sum += i;
    return (sum, items.Length);
}

public async Task<string> FetchAsync(string url) => await httpClient.GetStringAsync(url);
```

Talking points / Exercises:

- Show how `ref`, `out`, and `in` change passing semantics.
- Exercise: Create a method that returns both min and max using a tuple.

References / Diagram:

- Returns & tuples — https://learn.microsoft.com/dotnet/csharp/tuples

---

3. Method Overloading

Definition: Multiple methods with the same name but different parameter lists in the same scope.

Short description: Overloading improves API ergonomics. Overloads must differ in parameter types, count, or order (not return type only).

Variations:

- Overload by parameter count
- Overload by parameter types
- Overload combined with `params` or optional parameters (beware ambiguity)

Example:

```
public void Log(string message) { /* ... */ }
public void Log(string format, params object[] args) { /* ... */ }
public void Log(Exception ex, string message) { /* ... */ }
```

Talking points / Exercises:

- Discuss overload resolution and ambiguity with named/optional args.
- Exercise: Add overloads to `Add` for `double` and `decimal`.

References:

- Overload rules — https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/methods#overloading

---

4. Named and Optional Parameters

Definition: Named arguments specify parameter values by name; optional parameters provide default values.

Short description: Named and optional arguments increase call-site readability and reduce the need for many overloads, but can create versioning pitfalls when defaults change.

Variations:

- Positional + named mixing
- Optional with default value expressions

Example:

```
void Configure(int retries = 3, bool verbose = false) { }

// calls
Configure();
Configure(verbose: true); // named
Configure(5, verbose: true);
```

Talking points / Exercises:

- Show how named args allow skipping args in the middle.
- Discuss versioning: changing default values is a breaking change for callers using omitted args.
- Exercise: Call a method mixing named and positional args.

References:

- Named/Optional docs — https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments

---

5. out and ref Parameters

Definition: `ref` and `out` allow a method to modify variables passed by the caller; `out` requires assignment before return, `ref` requires initialization before call.

Short description: Use `ref` to read/write caller data, `out` when the method only needs to return additional values. Newer patterns favor tuples and return types over `out` for clarity, but `out` remains common (TryParse pattern).

Variations:

- `ref` (read/write) vs `out` (write-before-return)
- `in` (read-only by reference) as a related pattern

Example:

```
bool TryParseInt(string s, out int value)
{
    return int.TryParse(s, out value);
}

void Swap(ref int a, ref int b) { int t = a; a = b; b = t; }
```

Talking points / Exercises:

- Explain variable definite assignment rules.
- Exercise: Implement `TryGetMax(int[] items, out int max)`.

References:

- `out` / `ref` docs — https://learn.microsoft.com/dotnet/csharp/language-reference/keywords/out-parameter

---

6. params Keyword

Definition: `params` lets callers pass a variable number of arguments as an array.

Short description: Useful for APIs like `Console.WriteLine` or `string.Format` where the final parameter is a variable-length argument list. A method can have only one `params`, and it must be the last parameter.

Variations:

- `params T[]`
- `params object[]` for heterogenous argument lists

Example:

```
void Write(params string[] lines)
{
    foreach (var l in lines) Console.WriteLine(l);
}

Write("one", "two", "three");
Write(new string[] { "a", "b" });
```

Talking points / Exercises:

- Show how `params` affects overload resolution with arrays.
- Exercise: Create a `Sum(params int[] values)` method.

References:

- `params` docs — https://learn.microsoft.com/dotnet/csharp/language-reference/keywords/params

---

7. Method Invocation

Definition: Invocation is the act of calling a method — either directly, via delegate, or through reflection/dispatch.

Short description: Calls can be instance calls (require an object), static calls (type-level), asynchronous (await), or via delegates (function pointers) and reflection (slow, dynamic).

Variations:

- Direct vs delegate invocation
- Synchronous vs asynchronous invocation

Example:

```
var c = new Calculator();
int s = c.Add(1,2); // instance
int t = Calculator.StaticAdd(3,4); // static
Func<int,int,int> f = (x,y) => x+y; int r = f(1,2); // delegate
```

Talking points / Exercises:

- Demonstrate delegate invocation and `Action` / `Func` types.
- Exercise: Create a simple event or delegate and subscribe/invoke.

References:

- Delegates & events — https://learn.microsoft.com/dotnet/csharp/programming-guide/delegates-and-events/

---

8. Access Modifiers

Definition: Access modifiers control visibility of types and members (public, private, protected, internal, protected internal).

Short description: Use access modifiers to express intent and encapsulate implementation. `internal` restricts access to the assembly; `protected` to derived types; `protected internal` is the union of both.

Variations:

- `public`, `private`, `protected`, `internal`, `protected internal`, `private protected` (C# 7.2)

Example:

```
public class Foo
{
    private void PrivateMethod() {}
    protected void ProtectedMethod() {}
    internal void InternalMethod() {}
    public void PublicMethod() {}
}
```

Talking points / Exercises:

- Diagram who can call which member (class, assembly, derived class, other assemblies).
- Exercise: Show difference between `protected internal` and `private protected`.

References / Diagram:

- Access modifiers overview diagram — https://learn.microsoft.com/dotnet/csharp/language-reference/keywords/access-modifiers

---

9. Method Scope

Definition: Scope determines where method-local names are accessible and the lifetime of local variables.

Short description: Variables declared inside a method are local to that method. Local functions introduce nested scopes; closures capture variables and extend lifetimes.

Variations:

- Local variables vs parameters
- Captured variables (closures) vs static local functions (no capture)

Example:

```
void Outer()
{
    int x = 1;
    void Local() { Console.WriteLine(x); } // captures x
    Local();
}
```

Talking points / Exercises:

- Show captured variable example and potential pitfalls (mutability).
- Exercise: Convert captured local to a static local function and discuss errors.

References:

- Local functions & closures — https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/local-functions

---

10. Static Methods

Definition: A static method belongs to the type itself rather than an instance.

Short description: Static methods cannot access instance members directly. They are commonly used for utility behavior, factory methods, or singletons.

Variations:

- Static classes (only static members)
- Static local functions

Example:

```csharp
public static class MathUtil
{
    public static int Square(int x) => x * x;
}

// usage
int n = MathUtil.Square(4);
```

Talking points / Exercises:

- Discuss thread-safety considerations for static state.
- Exercise: Create a static helper with `ThreadStatic` vs `AsyncLocal` considerations.

References:

- Static members — https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

---

11. Extension Methods

Definition: Extension methods let you "add" methods to existing types without modifying them by declaring static methods where the first parameter is `this T`.

Short description: Extension methods are syntactic sugar — they are static methods invoked using instance method syntax. They are resolved based on `using` directives and can add discoverability to APIs.

Variations:

- Extension methods on interfaces (common with LINQ)
- Generic extension methods

Example:

```csharp
public static class StringExtensions
{
    public static bool IsNullOrEmpty(this string s) => string.IsNullOrEmpty(s);
}

// usage
string? s = null; bool empty = s.IsNullOrEmpty();
```

Talking points / Exercises:

- Show how extension methods participate in overload resolution.
- Exercise: Implement `ToIntOrNull(this string s)`.

References / Diagram:

- Extension methods — https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

---

12. Local Functions

Definition: Local functions are methods declared inside another method.

Short description: They improve encapsulation and readability by keeping helper logic close to the usage site. They can be `static` (no captures) or instance (capture outer variables).

Variations:

- Static local functions (no capture)
- Async local functions

Example:

```
int Fibonacci(int n)
{
    int FibLocal(int k)
    {
        if (k < 2) return k;
        return FibLocal(k-1) + FibLocal(k-2);
    }
    return FibLocal(n);
}
```

Talking points / Exercises:

- Discuss when local functions beat private helpers.
- Exercise: Convert a helper private method into a local function; discuss testability and visibility.

References:

- Local functions — https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/local-functions

---

13. Expression-bodied Members

Definition: Expression-bodied members use `=>` for concise single-expression implementations for methods, properties, and others.

Short description: They enable compact syntax for simple members and were expanded to cover constructors, finalizers, and accessors in recent C# versions.

Variations:

- Methods and properties
- Constructors / deconstructors / finalizers

Example:

```
public int Add(int x, int y) => x + y;
public string Name => _name ?? "<unknown>";
public override string ToString() => $"Person: {Name}";
```

Talking points / Exercises:

- Pros: readability for short logic. Cons: maintainability when logic grows.
- Exercise: Turn a multi-line method into expression-bodied member (if small enough).

References:

- Expression-bodied members — https://learn.microsoft.com/dotnet/csharp/programming-guide/statements-expressions-operators/expression-bodied-members

---

Session Structure & Suggested Timings (2 hours)

1. Intro & agenda (5 min)
2. Anatomy of a method & parameters/returns (15 min)
3. Overloading, named/optional, `params` (20 min)
4. `ref` / `out` / `in` and `params` (15 min)
5. Invocation, delegates, async basics (15 min)
6. Access modifiers & scope, static methods (15 min)
7. Extension methods & local functions (15 min)
8. Expression-bodied members & modern idioms (10 min)
9. Hands-on exercises & discussion (10–20 min)

Exercises (pick 3–4)

- Implement `TryFindMax(int[] numbers, out int max)` and a tuple-returning variant — compare API ergonomics.
- Create a `Logger` class with multiple overloads and `params` and demonstrate ambiguity when mixing named args.
- Implement an extension method `IsPalindrome(this string s)`.
- Convert a helper static method into a local function and discuss captured variables.

Cheat Sheet & Quick Links

- Members & access: https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/members
- Methods: https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/methods
- `ref` / `out`: https://learn.microsoft.com/dotnet/csharp/language-reference/keywords/out-parameter
- Extension methods: https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/extension-methods
- Expression-bodied members: https://learn.microsoft.com/dotnet/csharp/programming-guide/statements-expressions-operators/expression-bodied-members