Classes: The Core of Object-Oriented Programming

A **class** is the most common kind of **reference type** in C#. It serves as a blueprint for creating objects, which are instances of the class. A class encapsulates data (fields) and behavior (methods) into a single, cohesive unit.

The simplest possible class declaration looks like this:

```
class YourClassName { }
```

A more complex class can optionally include:

- **Attributes and class modifiers** before the `class` keyword. contoh -> public
- **Generic type parameters** after the class name.
- **Class members** within its braces, such as fields, methods, properties, constructors, and more.

Common **non-nested class modifiers** include `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe`, and `partial`.

Class Members

Within a class's braces, you define its members. Let's explore the key members:

Fields

A **field** is a variable that is a member of a class or struct. Fields store data associated with an object.

```
class Octopus
{
    string name;        // A private field
    public int Age = 10; // A public field initialized to 10
}
```

Fields can have various **modifiers**:

- **`static`**: The field belongs to the class itself, not to individual instances.
- **Access modifiers** (`public`, `internal`, `private`, `protected`): Control visibility.
- **`new`**: For hiding inherited members.

- **unsafe**: For pointers (advanced).
- **readonly**: Prevents modification after construction.
- **volatile**: For multi-threading (advanced).

Naming Conventions: For private fields, popular conventions are camelCased (e.g., firstName) or _camelCased (e.g., _firstName). The latter (_firstName) helps distinguish private fields from method parameters and local variables.

The readonly Modifier

The readonly modifier ensures that a field can only be assigned a value during its declaration or within the enclosing type's constructor. Once set, its value cannot be changed.

Field Initialization

Field initialization is optional. If a field is not explicitly initialized, it will automatically receive a **default value** (e.g., 0 for numeric types, '\0' for char, null for reference types, false for bool).

Field initializers run *before* constructors. An initializer can contain expressions and call methods.

```
public int Age = 10; // Field initializer
static readonly string TempFolder = System.IO.Path.GetTempPath(); //
Initializer with method call
```

Declaring Multiple Fields

For convenience, you can declare multiple fields of the same type in a comma-separated list. This allows them to share the same attributes and field modifiers.

```
static readonly int legs = 8, eyes = 2;
```

Constants

A **constant** is a special type of field whose value is determined **statically at compile time**. The compiler literally substitutes the constant's value directly into the code wherever it's used, similar to a macro in C++.

Constants can only be bool, char, string, any built-in numeric type, or an enum type. They are declared using the const keyword and *must* be initialized with a value.

Naming constant use UPPERCASE_WITH_UNDERSCORE

```
public class Test
{
    public const string ~~Message~~ = "Hello World"; // Constant declaration
}                       MESSAGE
```

Constants vs. static readonly Fields

Constants serve a similar role to static readonly fields but are much more restrictive.

- **Compile-time evaluation:** Constants are evaluated at compile time. For instance, 2 * System.Math.PI would be replaced by its calculated value (6.2831853071795862) directly in the compiled code. This makes sense for values like PI, which are truly constant.
- **Runtime evaluation:** A static readonly field's value can be determined at runtime and potentially differ each time the program runs. For example, static readonly DateTime StartupTime = DateTime.Now; will capture the exact time the program starts.
- **Versioning implications:** When exposing values to other assemblies, a static readonly field is advantageous over a const. If assembly X exposes a const (e.g., public const decimal ProgramVersion = 2.3;) and assembly Y references it, the value 2.3 is "baked in" to assembly Y at compile time. If assembly X later changes the constant to 2.4 and is recompiled, assembly Y will *still* use the old 2.3 value until it is also recompiled. A static readonly field avoids this binary compatibility issue because its value is looked up at runtime. Any value that might change should not be a constant.
- Constants can also be declared local to a method.

Constant -> valuenya gabakalan berubah
kalau static readonly -> setiap kita manggil
class/fungsinya itu valuenya bisa berubah
isinya, tapi cuman bisa diread aja gabisa
diubah setelah diinit

Methods

A **method** defines an action or behavior that a class or object can perform. Methods can accept input data through **parameters** and return output data via a **return type**. A `void` return type indicates that the method does not return any value. Methods can also output data using `ref` or `out` parameters.

A method's **signature** must be unique within its type. The signature comprises the method's name and the types of its parameters in order (parameter names and return type are *not* part of the signature).

```
// Examples of method signatures:
void Foo(int x) { ... }
void Foo(double x) { ... } // Different parameter type, different signature
void Foo(int x, float y) { ... } // Different number of parameters, different
signature
void Foo(float x, int y) { ... } // Different order of parameter types,
different signature
```

Methods can have various **modifiers**, including:

- **`static`**: Belongs to the class, not an instance.
- **Access modifiers** (`public`, `internal`, `private`, `protected`).
- **Inheritance modifiers** (`new`, `virtual`, `abstract`, `override`, `sealed`).
- **`partial`**: For partial methods.
- **Unmanaged code modifiers** (`unsafe`, `extern`).
- **`async`**: For asynchronous operations.

Expression-Bodied Methods

A method consisting of a single expression can be written concisely as an **expression-bodied method**. A "fat arrow" (`=>`) replaces the braces and `return` keyword.

```
int Foo(int x) => x * 2; // Expression-bodied method returning a value
void Foo(int x) => Console.WriteLine(x); // Expression-bodied method with void
return type
```

Local Methods

You can define a method *within* another method; these are called **local methods**[44].

```
void WriteCubes()
{
    Console.WriteLine(Cube(3));
    // ...
    int Cube(int value) => value * value * value; // Local method
}
```

A local method is only visible and accessible within its enclosing method. This simplifies the containing type and clearly indicates its limited usage. A significant benefit is that local methods can access the local variables and parameters of the enclosing method (this is called "capturing outer variables"). Local methods can be nested, and can be iterators or asynchronous.

Static Local Methods (C# 8+)

Adding the `static` modifier to a local method (from C# 8) prevents it from accessing local variables and parameters of the enclosing method. This reduces coupling and prevents accidental dependencies.

Local Methods and Top-Level Statements

Methods declared in top-level statements (a feature for simpler console applications) are treated as local methods and can access variables in the top-level statements unless marked `static`.

Overloading Methods

A type can **overload methods** (define multiple methods with the same name) as long as their **signatures are different**.

```
void Foo(int x) { ... }
void Foo(double x) { ... }
void Foo(int x, float y) { ... }
void Foo(float x, int y) { ... }
```

Important: The return type and the `params` modifier are *not* part of a method's signature. Therefore, the following pairs of methods cannot coexist:

```
void Foo(int x) { ... }
float Foo(int x) { ... } // Compile-time error (same signature, different
return type)

void Goo(int[] x) { ... }
void Goo(params int[] x) { ... } // Compile-time error (params is not part of
signature)
```

However, whether a parameter is passed by value or by reference (`ref`/`out`) *is* part of the signature. `Foo(int)` can coexist with `Foo(ref int)` or `Foo(out int)`. But `Foo(ref int)` and `Foo(out int)` cannot coexist.

Instance Constructors

Constructors are special methods that run initialization code when a new instance of a class or struct is created. A constructor's name is the same as its enclosing type, and it has no explicit return type.

```
Panda p = new Panda("Petey"); // Calls the constructor

public class Panda
{
    string name; // Field
    public Panda(string n) // Constructor definition
    {
        name = n; // Initialization code
    }
}
```

Instance constructors support access modifiers (`public`, `internal`, `private`, `protected`) and unmanaged code modifiers (`unsafe`, `extern`). Single-statement constructors can also be expression-bodied. If a parameter name conflicts with a field name, use `this.` to disambiguate the field.

Overloading Constructors

A class or struct can **overload constructors** (have multiple constructors with different parameter lists). To avoid code duplication, one constructor can call another using the `this` keyword, ensuring the called constructor executes first.

```
public class Wine
{
    public decimal Price;
    public int Year;

    public Wine(decimal price) => Price = price;

    // Calls the (decimal price) constructor first
    public Wine(decimal price, int year) : this(price) => Year = year;
}
```

When calling another constructor, expressions passed as arguments can access `static` members but not instance members, as the object isn't fully initialized yet.

Implicit Parameterless Constructors

For classes, the C# compiler automatically generates a `public` parameterless constructor *only if* you don't define any constructors yourself. As soon as you define at least one constructor, this default parameterless constructor is no longer generated automatically.

Constructor and Field Initialization Order

Field initializations (from their declarations) run *before* any constructor code is executed, and they run in the order in which the fields are declared.

Nonpublic Constructors

Constructors don't have to be `public`. A common use for a nonpublic constructor is to control instance creation through a static method, perhaps to return objects from a pool or different subclasses based on input.

Deconstructors

A **deconstructor** (or deconstructing method) performs the reverse operation of a constructor: it assigns field values back to a set of variables.

A deconstruction method must be named Deconstruct and have one or more out parameters.

```
class Rectangle
{
    public readonly float Width, Height;
    public Rectangle(float width, float height) { Width = width; Height =
height; }

    public void Deconstruct(out float width, out float height) // Deconstructor
definition
    {
        width = Width;
        height = Height;
    }
}
```

You call a deconstructor using special syntax:

```
var rect = new Rectangle(3, 4);
(float width, float height) = rect; // Deconstruction call
Console.WriteLine(width + " " + height); // Output: 3 4
```

This is equivalent to explicitly calling rect.Deconstruct(out width, out height);. Deconstructing calls support implicit typing (var (width, height) = rect;). You can use the discard symbol (_) for variables you don't care about. If variables are already defined, you can omit their types ((width, height) = rect;), which is called a **deconstructing assignment**.

You can overload the Deconstruct method to offer different deconstruction options. Deconstructors can also be extension methods, useful for deconstructing types you didn't author. From C# 10, you can mix existing and new variables in deconstruction.

Object Initializers

Object initializers provide a convenient syntax to set accessible fields or properties of an object directly after its construction.

```
public class Bunny
{
    public string Name;
    public bool LikesCarrots, LikesHumans;
    public Bunny() { }
    public Bunny(string n) => Name = n;
}

// Using object initializers:
// Note: Parameterless constructors can omit empty parentheses
Bunny b1 = new Bunny { Name = "Bo", LikesCarrots = true, LikesHumans = false };
Bunny b2 = new Bunny("Bo") { LikesCarrots = true, LikesHumans = false };
```

The compiler translates object initializers into code that first calls the constructor, then assigns the specified members. Temporary variables are used internally to ensure that if an exception occurs during initialization, you don't end up with a partially initialized object.

Object Initializers vs. Optional Parameters

While constructors with optional parameters can also initialize objects , object initializers (especially with C# 9's `init` modifier for properties) offer advantages, particularly regarding **immutability** and **backward compatibility** in public libraries. Adding an optional parameter to a `public` constructor breaks binary compatibility for existing consumers. Object initializers, combined with `init`-only properties, allow for non-destructive mutation and better versioning.

For complex initialization, using object initializers to fill in details can simplify constructors, especially in subclassing scenarios.

The `this` Reference

The **`this` reference** refers to the current instance of the class or struct. It is commonly used to:

- Set fields of the current object or pass the current object to another method.
- Disambiguate a local variable or parameter from a field with the same name.

```
public class Panda
{
    public Panda Mate;
    public void Marry(Panda partner)
    {
        Mate = partner;     // Sets the Mate of the current object
        partner.Mate = this; // Sets the Mate of the partner object to
the current object
    }
}
```

The `this` reference is only valid within non-static members of a class or struct.

Properties

Properties combine aspects of fields and methods. From the outside, they look like fields, but internally, they contain logic, similar to methods. This allows for encapsulation, providing controlled access to data.

```
// Using a property looks like accessing a field:
Stock msft = new Stock();
msft.CurrentPrice = 30;
Console.WriteLine(msft.CurrentPrice);
```

A property is declared with `get` and/or `set` accessors.

```
public class Stock
{
    decimal currentPrice; // The private "backing" field

    public decimal CurrentPrice // The public property
    {
        get { return currentPrice; } // The get accessor runs when property is
read
        set { currentPrice = value; } // The set accessor runs when property is
assigned
    }
}
```

The `get` accessor must return a value of the property's type. The `set` accessor has an implicit parameter named `value`, which holds the assigned data, typically assigned to a private backing field. Properties give the implementer complete control over how values are retrieved and set, enabling internal representation changes without affecting the public interface. For example, a `set` accessor could include validation logic.

While public fields are sometimes used for simplicity in examples, **public properties are generally favored over public fields in real applications to promote encapsulation**.

Properties allow various modifiers, similar to methods.

Read-Only and Calculated Properties

A property is **read-only** if it only has a `get` accessor. It's **write-only** if it only has a `set` accessor (rarely used).

A property doesn't always need a dedicated backing field; it can be **calculated** from other data.

```
decimal currentPrice, sharesOwned;
public decimal Worth // Calculated property
{
    get { return currentPrice * sharesOwned; }
}
```

Expression-Bodied Properties

Read-only properties can be declared tersely as **expression-bodied properties** using the `=>` fat arrow. Set accessors can also be expression-bodied.

```
public decimal Worth => currentPrice * sharesOwned; // Read-only
expression-bodied property
```

Automatic Properties

The most common property implementation simply reads and writes to a private backing field. **Automatic properties** (introduced in C# 3.0) instruct the compiler to generate this backing field and its accessors automatically, reducing boilerplate code.

```
public class Stock
{
    public decimal CurrentPrice { get; set; } // Automatic property
}
```

The compiler creates a private, unnamed backing field. The `set` accessor can be marked `private` or `protected` to make the property read-only to external types.

Property Initializers

You can add a **property initializer** to automatic properties, similar to fields.

```
public decimal CurrentPrice { get; set; } = 123;
```

Automatic properties with only a `get` accessor can also have initializers, making them read-only. They can also be assigned in the constructor, useful for creating **immutable types**.

`get` and `set` Accessibility

Accessors can have different access levels. Typically, the property itself has the more permissive access (e.g., `public`), and the `set` accessor is less accessible (e.g., `private` or `internal`).

```
public class Foo
{
    private decimal x;
    public decimal X
    {
        get { return x; }
        private set { x = Math.Round(value, 2); } // Private setter
    }
}
```

Init-Only Setters (C# 9+)

From C# 9, you can declare a property accessor with `init` instead of `set`. **Init-only properties** behave like read-only properties, but they can be set *only* via an object initializer during construction. After initialization, they cannot be altered.

```
public class Note
{
    public int Pitch { get; init; } = 20; // Init-only property
}

var note = new Note { Pitch = 50 }; // OK: Set via object initializer
// note.Pitch = 200; // Error: Cannot set after initialization
```

`Init`-only setters can modify `readonly` fields in their own class. They are advantageous for public libraries as adding new `init`-only properties does not break binary compatibility, unlike adding optional parameters to constructors. They also facilitate **nondestructive mutation** with records.

CLR Property Implementation

C# property accessors are compiled internally into `get_XXX` and `set_XXX` methods. `Init` accessors are processed similarly to `set` accessors but with an extra flag in their metadata. Simple nonvirtual property accessors are often **inlined** by the Just-In-Time (JIT) compiler, eliminating performance differences between properties and fields. Inlining replaces a method call with the method's body.

Indexers

Indexers provide a natural syntax for accessing elements in a class or struct that internally encapsulate a list or dictionary of values. They are similar to properties but are accessed using an index argument within square brackets, rather than a property name.

The `string` class, for example, has an indexer to access its characters:

```
string s = "hello";
Console.WriteLine(s[0]); // 'h'
```

Indexers can have any type for their index arguments and the same modifiers as properties. They can also be called null-conditionally using `?`.

Implementing an Indexer

To create an indexer, you define a property named `this` and specify the arguments in square brackets.

```
class Sentence
{
    string[] words = "The quick brown fox".Split();

    public string this[int wordNum] // Indexer definition
    {
        get { return words[wordNum]; }
        set { words[wordNum] = value; }
    }
}

// Usage:
Sentence s = new Sentence();
Console.WriteLine(s[3]); // fox
s[3] = "kangaroo"; // Sets the value
```

A type can declare multiple indexers with different parameter types or multiple parameters. If the `set` accessor is omitted, the indexer is read-only. Expression-bodied syntax can be used for read-only indexers.

CLR Indexer Implementation

Indexers compile internally to get_Item and set_Item methods.

Using Indices and Ranges with Indexers

From C# 8, you can support **indices** (Index) and **ranges** (Range) in your indexers.

```
public string this[Index index] => words[index]; // Indexer for System.Index
public string[] this[Range range] => words[range]; // Indexer for System.Range

// Usage:
Sentence s = new Sentence();
Console.WriteLine(s[^1]);      // fox (last element using Index)
string[] firstTwoWords = s[..2]; // (The, quick) (range from start to index 2, exclusive)
```

Primary Constructors (C# 12+)

From C# 12, you can include a parameter list directly after a class (or struct) declaration. This creates a **primary constructor**.

```
class Person (string firstName, string lastName) // Primary constructor parameters
{
    public void Print() => Console.WriteLine(firstName + " " + lastName);
}

Person p = new Person("Alice", "Jones"); // Instantiates using primary constructor
p.Print(); // Alice Jones
```

Primary constructors are useful for simple scenarios and prototyping. They eliminate the need for explicit field declarations and a separate constructor for basic initialization.

Any additional constructors you explicitly write *must* invoke the primary constructor using this. This ensures that primary constructor parameters are always populated.

While similar to records (which also support primary constructors), records by default generate public init-only properties for each primary constructor parameter. Consider records if this behavior is desired.

Limitations of primary constructors:

- You cannot add extra initialization code directly to a primary constructor.
- Exposing primary constructor parameters as read/write public properties and adding validation logic can be complex.
- Primary constructors displace the default parameterless constructor that C# would otherwise generate.

Primary Constructor Semantics

Unlike ordinary constructor parameters that go out of scope after the constructor finishes, a primary constructor's parameters **remain in scope** throughout the class for the lifetime of the object. They are special C# constructs, not fields, though the compiler may generate hidden fields to store their values.

Primary Constructors and Field/Property Initializers

The accessibility of primary constructor parameters extends to field and property initializers.

```
class Person (string firstName, string lastName)
{
    public readonly string FirstName = firstName; // Field initialized with primary constructor param
    public string LastName { get; } = lastName;   // Property initialized with primary constructor param
}
```

Masking Primary Constructor Parameters

Fields or properties can reuse primary constructor parameter names. In this case, the field/property takes precedence, masking the primary constructor parameter, except on the right-hand side of initializers.

Validating Primary Constructor Parameters

You can perform computations and validation in field initializers using primary constructor parameters. This allows for validation upon construction, such as throwing an `ArgumentNullException` if a parameter is `null`. Code within a field or property initializer executes when the object is constructed.

However, adding complex validation to read/write properties exposed from primary constructor parameters can become cumbersome, potentially leading back to explicit constructor definitions and backing fields.

Static Constructors

A **static constructor** executes once per type, rather than once per instance. A type can have only one static constructor, and it must be parameterless and have the same name as the type.

```
class Test
{
    static Test() // Static constructor
    {
        Console.WriteLine("Type Initialized");
    }
}
```

The runtime automatically invokes a static constructor just before the type is first used, either by instantiating the type or by accessing a static member within it. Only `unsafe` and `extern` modifiers are allowed. If a static constructor throws an unhandled exception, that type becomes unusable for the application's lifetime.

From C# 9, **module initializers** can also be defined, which execute once per assembly when the assembly is first loaded.

Static Constructors and Field Initialization Order

Static field initializers run just before the static constructor is called. If there's no static constructor, they run just before the type is used, or earlier at the runtime's discretion. Static field initializers run in their declared order.

Static Classes

A class marked `static` cannot be instantiated (you can't create objects of it) or subclassed. It must also consist solely of static members. `System.Console` and `System.Math` are classic examples of static classes.

Finalizers

Finalizers (also called deconstructors in some contexts, but distinct from C# deconstructing methods) are class-only methods that execute just before the garbage collector reclaims the memory for an unreferenced object. The syntax is the class name prefixed with `~`.

```
class Class1
{
    ~Class1() // Finalizer
    {
        // Cleanup code here
    }
}
```

This C# syntax is a shorthand for overriding the `Object.Finalize` method. Finalizers are discussed in detail with garbage collection. Single-statement finalizers can use expression-bodied syntax.

Partial Types and Methods

Partial types allow a single class, struct, or interface definition to be split across multiple source files. This is common for auto-generated code (e.g., from UI designers) where you want to add manual augmentations without modifying the generated file.

```
// PaymentFormGen.cs (auto-generated)
partial class PaymentForm { /* ... generated code ... */ }

// PaymentForm.cs (hand-authored)
partial class PaymentForm { /* ... your custom code ... */ }
```

Each part of the definition must have the partial keyword. Participants cannot have conflicting members (e.g., duplicate constructors with the same parameters). All parts must be available at compile time and reside in the same assembly. They can specify a base class (which must be the same across all parts) and independently implement interfaces. The compiler makes no guarantees about field initialization order between partial type declarations.

Partial Methods

A **partial type can contain partial methods**. These provide customizable "hooks" from auto-generated code for manual authoring.

```
partial class PaymentForm // In auto-generated file
{
    partial void ValidatePayment(decimal amount); // Definition
}

partial class PaymentForm // In hand-authored file
{
    partial void ValidatePayment(decimal amount) // Implementation
    {
        if (amount > 100) { /* ... validation logic ... */ }
    }
}
```

A partial method has a **definition** (typically generated) and an **implementation** (typically manually authored). If no implementation is provided, the definition and any calls to it are compiled away, preventing code bloat.

Partial methods must be void and are implicitly private. They cannot include out parameters.

Extended Partial Methods (C# 9+)

Extended partial methods (from C# 9) reverse the scenario: they allow a programmer to define hooks that a code generator *must* implement. They are identified by having an **accessibility modifier** (e.g., `public`, `private`).

```
public partial class Test
{
    public partial void M1(); // Extended partial method (must have
implementation)
    private partial void M2(); // Extended partial method (must have
implementation)
}
```

Unlike regular partial methods, extended partial methods *must* have implementations; they do not "melt away" if unimplemented. Because they are guaranteed to exist, they can return any type and include `out` parameters.

The `nameof` Operator

The **`nameof` operator** returns the name of any symbol (type, member, variable, etc.) as a string at compile time.

```
int count = 123;
string name = nameof(count); // name is "count"
```

The advantage of `nameof` over a simple string literal is **static type checking**. Development tools can understand the symbol reference, so if you rename the symbol, all references using `nameof` will also be updated automatically.

To get the name of a type member, include the type.

```
string name = nameof(StringBuilder.Length); // Evaluates to "Length"
```

To get the fully qualified name (e.g., "StringBuilder.Length"), you would concatenate the names.