## **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.

Kelas adalah jenis tipe referensi yang paling umum dalam C#. Kelas berfungsi sebagai cetak biru untuk membuat objek, yang merupakan contoh dari kelas. Sebuah kelas mengenkapsulasi data (field) dan perilaku (metode) ke dalam satu unit yang kohesif.

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.
* **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.

Kelas yang lebih kompleks dapat disertakan secara opsional:

* Atribut dan pengubah kelas sebelum kata kunci kelas.
* Parameter tipe umum setelah nama kelas.
* Anggota kelas di dalam kurung, seperti bidang, metode, properti, konstruktor, dan lainnya.

Pengubah kelas yang tidak bersarang yang umum termasuk publik, internal, abstrak, tertutup, statis, tidak aman, dan parsial.

## **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.

Field adalah sebuah variabel yang merupakan anggota dari sebuah kelas atau struktur. Field menyimpan data yang terkait dengan sebuah objek.

|  |
| --- |
| 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).

Bidang dapat memiliki berbagai pengubah:

* statis: Field adalah milik kelas itu sendiri, bukan milik masing-masing instance.
* Pengubah akses (publik, internal, privat, terlindungi): Mengontrol visibilitas.
* new: Untuk menyembunyikan anggota yang diwariskan.
* tidak aman: Untuk petunjuk (lanjutan).
* hanya-baca: Hanya-baca: Mencegah modifikasi setelah konstruksi.
* volatile: Untuk multi-threading (lanjutan).

**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.

Pengubah readonly memastikan bahwa sebuah field hanya bisa diberi nilai selama deklarasi atau di dalam konstruktor tipe yang mengapitnya. Setelah ditetapkan, nilainya tidak dapat diubah.

#### **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.

Untuk kenyamanan, Anda dapat mendeklarasikan beberapa field dengan tipe yang sama dalam daftar yang dipisahkan dengan koma. Hal ini memungkinkan mereka untuk berbagi atribut dan pengubah bidang yang sama

|  |
| --- |
| 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.

|  |
| --- |
| public class Test {  public const string Message = "Hello World"; // Constant declaration  } |

#### **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.
* Konstanta dievaluasi pada saat kompilasi. Sebagai contoh, 2 \* System.Math.PI akan digantikan dengan nilai yang dihitung (6.2831853071795862) secara langsung di dalam kode yang dikompilasi. Hal ini masuk akal untuk
* **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.
* Nilai field readonly statis dapat ditentukan pada saat runtime dan berpotensi berbeda setiap kali program dijalankan. Sebagai contoh, readonly statis DateTime StartupTime = DateTime.Now; akan menangkap waktu yang tepat saat program dimulai.
* **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.
* Ketika mengekspos nilai ke rakitan lain, field readonly statis lebih menguntungkan daripada const. Jika rakitan X mengekspos konstanta (misal, konstanta publik desimal ProgramVersion = 2.3;) dan rakitan Y mereferensikannya, nilai 2.3 akan “dimasukkan” ke rakitan Y pada waktu kompilasi. Jika assembly X kemudian mengubah konstanta menjadi 2.4 dan dikompilasi ulang, maka assembly Y akan tetap menggunakan nilai 2.3 yang lama hingga dikompilasi ulang. Field readonly statis menghindari masalah kompatibilitas biner ini karena nilainya dicari pada saat runtime. Nilai apa pun yang mungkin berubah tidak boleh berupa konstanta.
* Constants can also be declared local to a method.

### **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.

Metode mendefinisikan tindakan atau perilaku yang dapat dilakukan oleh sebuah kelas atau objek. Metode dapat menerima data masukan melalui parameter dan mengembalikan data keluaran melalui tipe pengembalian. Tipe pengembalian void mengindikasikan bahwa metode tidak mengembalikan nilai apa pun. Metode juga dapat mengeluarkan data menggunakan parameter ref atau out.

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).

Tanda tangan sebuah metode harus unik dalam jenisnya. Tanda tangan terdiri dari nama metode dan tipe parameternya secara berurutan (nama parameter dan tipe pengembalian bukan bagian dari tanda tangan).

|  |
| --- |
| // 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.

Metode yang terdiri dari satu ekspresi dapat ditulis secara ringkas sebagai metode berbadan ekspresi. Sebuah “tanda panah gemuk” (=>) menggantikan tanda kurung dan kata kunci return.

|  |
| --- |
| 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.

Metode lokal hanya dapat dilihat dan diakses di dalam metode pengapitnya. Hal ini menyederhanakan tipe penampung dan dengan jelas menunjukkan penggunaannya yang terbatas. Manfaat yang signifikan adalah bahwa metode lokal dapat mengakses variabel lokal dan parameter dari metode penampungnya (ini disebut “menangkap variabel luar”). Metode lokal dapat bersarang, dan dapat berupa iterator atau asinkron.

#### **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.

Menambahkan pengubah statis ke metode lokal (dari C# 8) mencegahnya mengakses variabel lokal dan parameter dari metode yang diapit. Hal ini mengurangi penggabungan dan mencegah ketergantungan yang tidak disengaja.

#### **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.

Metode yang dideklarasikan dalam pernyataan tingkat atas (fitur untuk aplikasi konsol yang lebih sederhana) diperlakukan sebagai metode lokal dan dapat mengakses variabel dalam pernyataan tingkat atas kecuali jika ditandai statis.

**Overloading Methods**

A type can **overload methods** (define multiple methods with the same name) as long as their **signatures are different**.  
  
Sebuah tipe dapat membebani metode (mendefinisikan beberapa metode dengan nama yang sama) selama tanda tangannya berbeda.

|  |
| --- |
| 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:

Tipe pengembalian dan pengubah params bukan bagian dari tanda tangan metode. Oleh karena itu, pasangan metode berikut ini tidak dapat hidup berdampingan:

|  |
| --- |
| 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.

Namun, apakah sebuah parameter dioper dengan nilai atau dengan referensi (ref/out) adalah bagian dari signature. Foo(int) bisa berdampingan dengan Foo(ref int) atau Foo(out int). Tetapi Foo(ref int) dan Foo(out int) tidak dapat hidup berdampingan.

### **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.

Konstruktor adalah metode khusus yang menjalankan kode inisialisasi ketika sebuah instance baru dari sebuah kelas atau struct dibuat. Nama konstruktor sama dengan tipe yang melingkupinya, dan tidak memiliki tipe pengembalian eksplisit.

|  |
| --- |
| 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.

Konstruktor instance mendukung pengubah akses (publik, internal, privat, terproteksi) dan pengubah kode yang tidak dikelola (tidak aman, eksternal). Konstruktor pernyataan tunggal juga dapat berupa ekspresi. Jika nama parameter bertentangan dengan nama field, gunakan ini. untuk memisahkan 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.

Sebuah kelas atau struktur dapat membebani konstruktor (memiliki beberapa konstruktor dengan daftar parameter yang berbeda). Untuk menghindari duplikasi kode, satu konstruktor dapat memanggil konstruktor lain menggunakan kata kunci ini, memastikan konstruktor yang dipanggil dieksekusi terlebih dahulu.

|  |
| --- |
| 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.

Ketika memanggil konstruktor lain, ekspresi yang dioper sebagai argumen dapat mengakses anggota statis tetapi tidak dapat mengakses anggota instance, karena objek tersebut belum diinisialisasi secara penuh.

#### **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.

Untuk kelas, kompiler C# secara otomatis menghasilkan konstruktor tanpa parameter publik hanya jika Anda tidak mendefinisikan konstruktor apa pun. Segera setelah Anda mendefinisikan setidaknya satu konstruktor, konstruktor tanpa parameter default ini tidak lagi dihasilkan secara otomatis.

#### **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.

Inisialisasi field (dari deklarasinya) dijalankan sebelum kode konstruktor dieksekusi, dan dijalankan sesuai urutan field dideklarasikan.

#### **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.Konstruktor tidak harus bersifat publik. Penggunaan umum untuk konstruktor nonpublik adalah untuk mengontrol pembuatan instance melalui metode statis, mungkin untuk mengembalikan objek dari kumpulan atau subkelas yang berbeda berdasarkan masukan.

**Deconstructors**

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

Dekonstruktor (atau metode dekonstruksi) melakukan operasi kebalikan dari konstruktor: metode ini memberikan nilai field kembali ke sekumpulan variabel.

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

Metode dekonstruksi harus diberi nama Deconstruct dan memiliki satu atau lebih parameter keluar.

|  |
| --- |
| 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**.

Ini sama dengan memanggil secara eksplisit rect.Deconstruct(out width, out height);. Panggilan dekonstruksi mendukung pengetikan implisit (var (lebar, tinggi) = rect;). Anda bisa menggunakan simbol buang (\_) untuk variabel yang tidak Anda pedulikan. Jika variabel sudah didefinisikan, Anda bisa menghilangkan tipe mereka ((width, height) = rect;), yang disebut sebagai penugasan dekonstruksi.

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.  
Anda dapat membebani metode Dekonstruksi untuk menawarkan opsi dekonstruksi yang berbeda. Dekonstruktor juga bisa menjadi metode ekstensi, berguna untuk mendekonstruksi tipe yang tidak Anda buat. Dari C# 10, Anda dapat mencampur variabel yang ada dan variabel baru dalam dekonstruksi.

### **Object Initializers**

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

menyediakan sintaks yang mudah digunakan untuk mengatur bidang atau properti yang dapat diakses dari sebuah objek secara langsung setelah pembuatannya.

|  |
| --- |
| 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.

Kompiler menerjemahkan inisialisasi objek ke dalam kode yang pertama-tama memanggil konstruktor, kemudian memberikan anggota yang ditentukan. Variabel sementara digunakan secara internal untuk memastikan bahwa jika terjadi pengecualian selama inisialisasi, Anda tidak akan mendapatkan objek yang diinisialisasi sebagian.

#### **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.

Meskipun konstruktor dengan parameter opsional juga dapat menginisialisasi objek, inisialisasi objek (terutama dengan pengubah init C# 9 untuk properti) menawarkan keuntungan, terutama terkait keabadian dan kompatibilitas ke belakang dalam pustaka publik. Menambahkan parameter opsional ke konstruktor publik akan merusak kompatibilitas biner untuk konsumen yang sudah ada. Inisialisasi objek, dikombinasikan dengan properti init-only, memungkinkan mutasi yang tidak merusak dan versi yang lebih baik.

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

Untuk inisialisasi yang kompleks, menggunakan inisialisasi objek untuk mengisi detail dapat menyederhanakan konstruktor, terutama dalam skenario subkelas.

### **The this Reference**

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

Referensi ini merujuk pada instance saat ini dari kelas atau struktur. Ini biasanya digunakan untuk:

* 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.
* Mengatur bidang dari objek saat ini atau mengoper objek saat ini ke metode lain.
* Menguraikan variabel atau parameter lokal dari field dengan nama yang sama.

|  |
| --- |
| 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.

Referensi ini hanya berlaku di dalam anggota non-statis dari kelas atau struktur.

### **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.

Properti menggabungkan aspek-aspek dari field dan metode. Dari luar, properti terlihat seperti field, namun secara internal, properti berisi logika, mirip dengan metode. Hal ini memungkinkan enkapsulasi, menyediakan akses terkontrol ke 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.

Sebuah properti dideklarasikan dengan pengakses get dan/atau set.

|  |
| --- |
| 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.

Pengakses get harus mengembalikan nilai dari tipe properti. Pengakses set memiliki parameter implisit bernama value, yang menyimpan data yang ditetapkan, biasanya ditetapkan ke bidang dukungan pribadi. Properti memberikan kontrol penuh kepada implementator tentang bagaimana nilai diambil dan ditetapkan, memungkinkan perubahan representasi internal tanpa mempengaruhi antarmuka publik. Sebagai contoh, pengakses set dapat menyertakan logika validasi.

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

Meskipun field publik terkadang digunakan untuk penyederhanaan dalam contoh, properti publik umumnya lebih disukai daripada field publik dalam aplikasi nyata untuk mempromosikan enkapsulasi.

Properties allow various modifiers, similar to methods.

Properti memungkinkan berbagai pengubah, mirip dengan metode.

#### **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).

Sebuah properti bersifat hanya-baca jika hanya memiliki pengakses get. Properti bersifat hanya-tulis jika hanya memiliki pengakses set (jarang digunakan).

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

Sebuah properti tidak selalu membutuhkan bidang dukungan khusus; bidang ini dapat dihitung dari data lain.

|  |
| --- |
| 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.

Properti read-only bisa dideklarasikan secara terselubung sebagai properti bertubuh ekspresi menggunakan tanda panah gemuk =>. Pengakses set juga dapat berupa ekspresi.

|  |
| --- |
| 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.

Implementasi properti yang paling umum hanya membaca dan menulis ke sebuah bidang pendukung pribadi. Properti otomatis (diperkenalkan pada C# 3.0) menginstruksikan kompiler untuk menghasilkan field pendukung dan pengaksesnya secara otomatis, sehingga mengurangi kode boilerplate.

|  |
| --- |
| 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.

Kompiler membuat sebuah field pendukung yang bersifat privat dan tidak diberi nama. Pengakses set dapat ditandai privat atau diproteksi untuk membuat properti hanya-baca untuk tipe eksternal.

#### **Property Initializers**

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

Anda dapat menambahkan inisialisasi properti ke properti otomatis, mirip dengan bidang.

|  |
| --- |
| 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**.

Properti otomatis dengan hanya pengakses get juga bisa memiliki inisialisasi, menjadikannya hanya-baca. Properti ini juga dapat ditetapkan di dalam konstruktor, berguna untuk membuat tipe yang tidak dapat diubah.

**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).  
Pengakses dapat memiliki tingkat akses yang berbeda. Biasanya, properti itu sendiri memiliki akses yang lebih permisif (misalnya, publik), dan pengakses yang ditetapkan kurang dapat diakses (misalnya, pribadi atau 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.

Dari C# 9, Anda bisa mendeklarasikan pengakses properti dengan init, bukan set. Properti init-only berperilaku seperti properti read-only, tetapi hanya dapat diatur melalui inisialisasi objek selama konstruksi. Setelah inisialisasi, properti tersebut tidak dapat diubah.

|  |
| --- |
| 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.

Penyetel init-only dapat memodifikasi field readonly di kelas mereka sendiri. Properti ini menguntungkan untuk pustaka publik karena menambahkan properti init-only baru tidak merusak kompatibilitas biner, tidak seperti menambahkan parameter opsional ke konstruktor. Mereka juga memfasilitasi mutasi yang tidak merusak dengan catatan.

#### **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.

Pengakses properti C# dikompilasi secara internal ke dalam metode get\_XXX dan set\_XXX. Pengakses init diproses dengan cara yang sama dengan pengakses set tetapi dengan flag tambahan dalam metadata mereka. Pengakses properti nonvirtual sederhana sering kali di-‘inline’ oleh kompiler Just-In-Time (JIT), sehingga menghilangkan perbedaan kinerja antara properti dan field. Inlining menggantikan pemanggilan metode dengan badan metode.

**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.

Pengindeks menyediakan sintaks alami untuk mengakses elemen dalam kelas atau struktur yang secara internal mengenkapsulasi daftar atau kamus nilai. Mereka mirip dengan properti tetapi diakses menggunakan argumen indeks di dalam tanda kurung siku, bukan nama properti.

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

Kelas string, misalnya, memiliki pengindeks untuk mengakses karakternya:

|  |
| --- |
| 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 ?.

Pengindeks dapat memiliki tipe apa saja untuk argumen indeks dan pengubah yang sama dengan properti. Mereka juga dapat dipanggil dengan null-kondisional menggunakan ?.

#### **Implementing an Indexer**

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

Untuk membuat pengindeks, Anda mendefinisikan properti bernama ini dan menentukan argumen dalam tanda kurung siku.

|  |
| --- |
| 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.

Sebuah tipe dapat mendeklarasikan beberapa pengindeks dengan tipe parameter yang berbeda atau beberapa parameter. Jika pengakses yang ditetapkan dihilangkan, pengindeks hanya-baca. Sintaks bertubuh ekspresi dapat digunakan untuk pengindeks hanya-baca.

#### **CLR Indexer Implementation**

Indexers compile internally to get\_Item and set\_Item methods.

Pengindeks melakukan kompilasi secara internal ke metode get\_Item dan set\_Item.

#### **Using Indices and Ranges with Indexers**

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

Dari C# 8, Anda dapat mendukung indeks (Index) dan rentang (Range) dalam pengindeks Anda.

|  |
| --- |
| 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**.

Dari C# 12, Anda dapat menyertakan daftar parameter secara langsung setelah deklarasi kelas (atau struct). Hal ini akan menciptakan sebuah konstruktor primer.

|  |
| --- |
| 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.

Konstruktor primer berguna untuk skenario dan pembuatan prototipe sederhana. Mereka menghilangkan kebutuhan akan deklarasi bidang eksplisit dan konstruktor terpisah untuk inisialisasi dasar.

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

Setiap konstruktor tambahan yang Anda tulis secara eksplisit harus memanggil konstruktor utama dengan menggunakan ini. Hal ini memastikan bahwa parameter konstruktor utama selalu terisi.

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.

Meskipun mirip dengan record (yang juga mendukung konstruktor primer), record secara default menghasilkan properti init-only publik untuk setiap parameter konstruktor primer. Pertimbangkan records jika perilaku ini diinginkan.

**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.
* Anda tidak dapat menambahkan kode inisialisasi tambahan secara langsung ke konstruktor primer.
* Mengekspos parameter konstruktor primer sebagai properti publik baca/tulis dan menambahkan logika validasi bisa jadi rumit.
* Konstruktor primer menggantikan konstruktor tanpa parameter default yang seharusnya dibuat oleh C#.

#### **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.

Tidak seperti parameter konstruktor biasa yang keluar dari ruang lingkup setelah konstruktor selesai, parameter konstruktor primer tetap berada dalam ruang lingkup di seluruh kelas selama masa pakai objek. Parameter tersebut merupakan konstruksi C# khusus, bukan field, meskipun kompiler bisa saja membuat field tersembunyi untuk menyimpan nilainya.

#### **Primary Constructors and Field/Property Initializers**

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

Aksesibilitas parameter konstruktor utama diperluas ke inisialisasi field dan properti.

|  |
| --- |
| 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.

#### Field atau properti dapat menggunakan kembali nama parameter konstruktor utama. Dalam kasus ini, field/properti akan didahulukan, menyembunyikan parameter konstruktor utama, kecuali di sisi kanan inisialisasi.

#### **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.

Anda dapat melakukan komputasi dan validasi pada inisialisasi field menggunakan parameter konstruktor utama. Hal ini memungkinkan validasi pada saat konstruksi, seperti melempar ArgumentNullException jika parameter bernilai null. Kode dalam inisialisasi field atau properti dieksekusi saat objek dikonstruksi.

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.

Namun, menambahkan validasi yang rumit pada properti baca/tulis yang diekspos dari parameter konstruktor utama dapat menjadi tidak praktis, sehingga berpotensi kembali ke definisi konstruktor eksplisit dan field pendukung.

### **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.

dieksekusi sekali per tipe, bukan sekali per instance. Sebuah tipe hanya bisa memiliki satu konstruktor statis, dan konstruktor tersebut harus tidak memiliki parameter dan memiliki nama yang sama dengan tipe tersebut.

|  |
| --- |
| 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.

Runtime secara otomatis memanggil konstruktor statis sebelum tipe tersebut pertama kali digunakan, baik dengan menginstansiasi tipe tersebut atau dengan mengakses anggota statis di dalamnya. Hanya pengubah yang tidak aman dan eksternal yang diperbolehkan. Jika konstruktor statis melempar eksepsi yang tidak ditangani, tipe tersebut tidak dapat digunakan selama masa pakai aplikasi.

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

juga dapat didefinisikan, yang dieksekusi sekali per rakitan ketika rakitan pertama kali dimuat.

**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.

Inisialisasi bidang statis dijalankan tepat sebelum konstruktor statis dipanggil. Jika tidak ada konstruktor statis, inisialisasi ini akan dijalankan tepat sebelum tipe digunakan, atau lebih awal sesuai dengan kebijaksanaan runtime. Inisialisasi bidang statis berjalan sesuai urutan yang dideklarasikan.

**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.

Kelas yang ditandai statis tidak dapat diinstansiasi (Anda tidak dapat membuat objek darinya) atau disubkelaskan. Kelas ini juga harus hanya terdiri dari anggota statis. System.Console dan System.Math adalah contoh klasik kelas statis.

### **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 ~.

(juga disebut dekonstruktor dalam beberapa konteks, tetapi berbeda dengan metode dekonstruksi C#) adalah metode khusus kelas yang dieksekusi sebelum pengumpul sampah mengambil kembali memori untuk objek yang tidak direferensikan. Sintaksnya adalah nama kelas yang diawali dengan ~.

|  |
| --- |
| 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.

Sintaks C# ini adalah singkatan untuk mengganti metode Object.Finalize. Finalis dibahas secara rinci dengan pengumpulan sampah. Penyelesai pernyataan tunggal dapat menggunakan sintaks ekspresi.

**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.

memungkinkan satu kelas, struktur, atau definisi antarmuka dibagi ke beberapa file sumber. Hal ini biasa terjadi pada kode yang dibuat secara otomatis (misalnya, dari desainer UI) di mana Anda ingin menambahkan penambahan secara manual tanpa memodifikasi file yang dibuat.

|  |
| --- |
| // 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.

Setiap bagian dari definisi harus memiliki kata kunci parsial. Part tidak boleh memiliki anggota yang saling bertentangan (misalnya, konstruktor duplikat dengan parameter yang sama). Semua bagian harus tersedia pada saat kompilasi dan berada di assembly yang sama. Mereka dapat menentukan kelas dasar (yang harus sama di semua bagian) dan mengimplementasikan antarmuka secara independen. Kompiler tidak memberikan jaminan tentang urutan inisialisasi field di antara deklarasi tipe parsial.

#### **Partial Methods**

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

Tipe parsial dapat berisi metode parsial. Ini menyediakan “pengait” yang dapat disesuaikan dari kode yang dibuat secara otomatis untuk penulisan manual.

|  |
| --- |
| 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.

Metode parsial memiliki definisi (biasanya dibuat) dan implementasi (biasanya ditulis secara manual). Jika tidak ada implementasi yang disediakan, definisi dan semua pemanggilannya akan dikompilasi, sehingga mencegah kode yang membengkak.

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

Metode parsial harus batal dan secara implisit bersifat privat. Metode-metode tersebut tidak dapat menyertakan parameter.

#### **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).

Metode parsial yang diperluas (dari C# 9) membalikkan skenario: metode ini memungkinkan programmer untuk mendefinisikan hook yang harus diimplementasikan oleh pembuat kode. Metode-metode ini diidentifikasi dengan memiliki pengubah aksesibilitas (misalnya, publik, privat).

|  |
| --- |
| 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.

Tidak seperti metode parsial biasa, metode parsial yang diperluas harus memiliki implementasi; metode ini tidak “meleleh” jika tidak diimplementasikan. Karena dijamin ada, metode ini dapat mengembalikan tipe apa pun dan menyertakan parameter.

### **The nameof Operator**

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

Operator nameof mengembalikan nama simbol apa pun (tipe, anggota, variabel, dll.) sebagai sebuah string pada waktu kompilasi.

|  |
| --- |
| 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.

Keuntungan dari nameof dibandingkan dengan literal string sederhana adalah pemeriksaan tipe statis. Alat pengembangan dapat memahami referensi simbol, sehingga jika Anda mengganti nama simbol, semua referensi yang menggunakan nameof juga akan diperbarui secara otomatis.

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.