**C# 9 and 10 - Cheat Sheet**

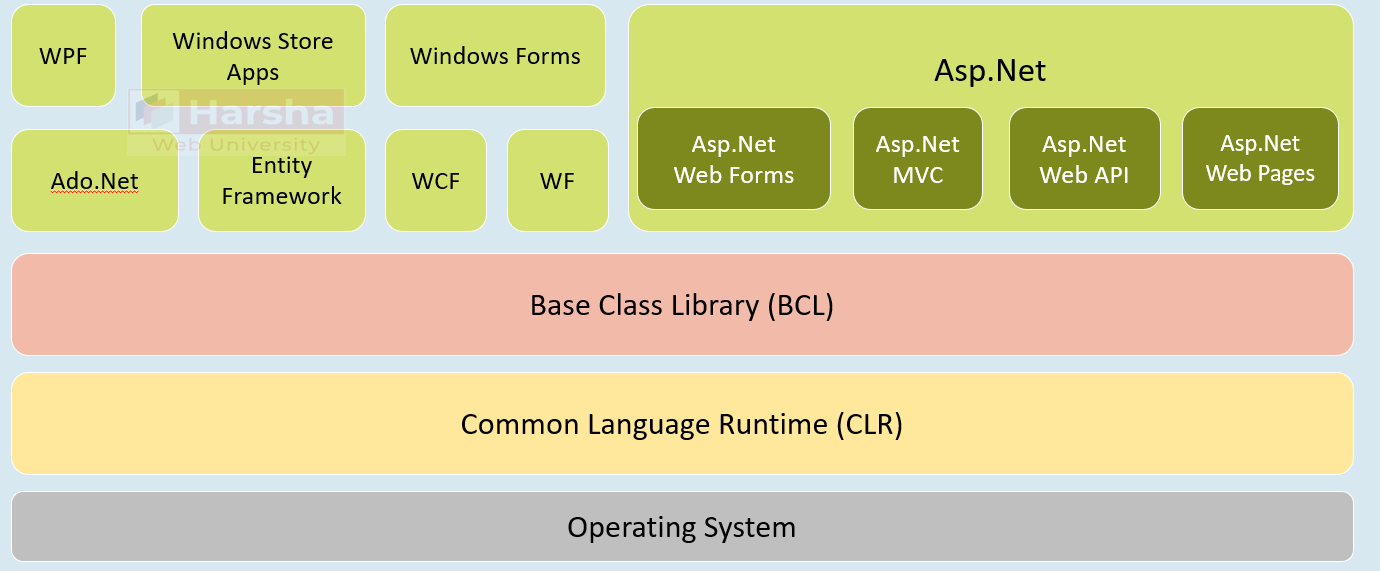
.Net Framework [vs] .Net Core [vs] .Net

.Net Framework:     Since 2002

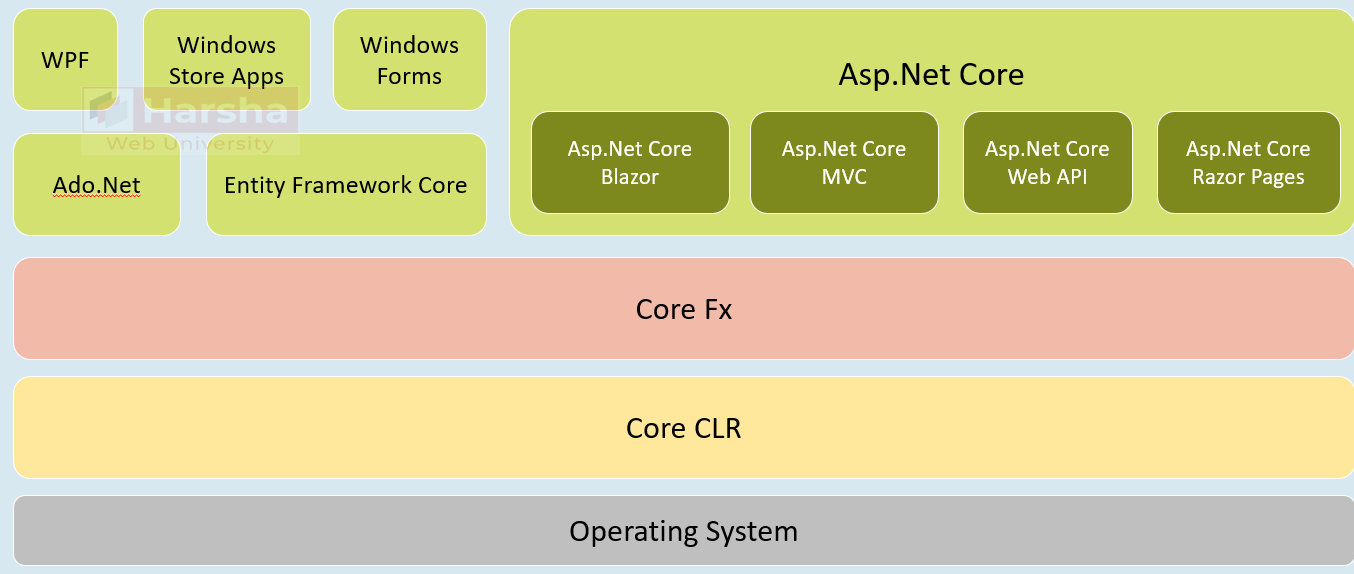
.Net Core:                 Since 2016

.Net:                           Since 2020

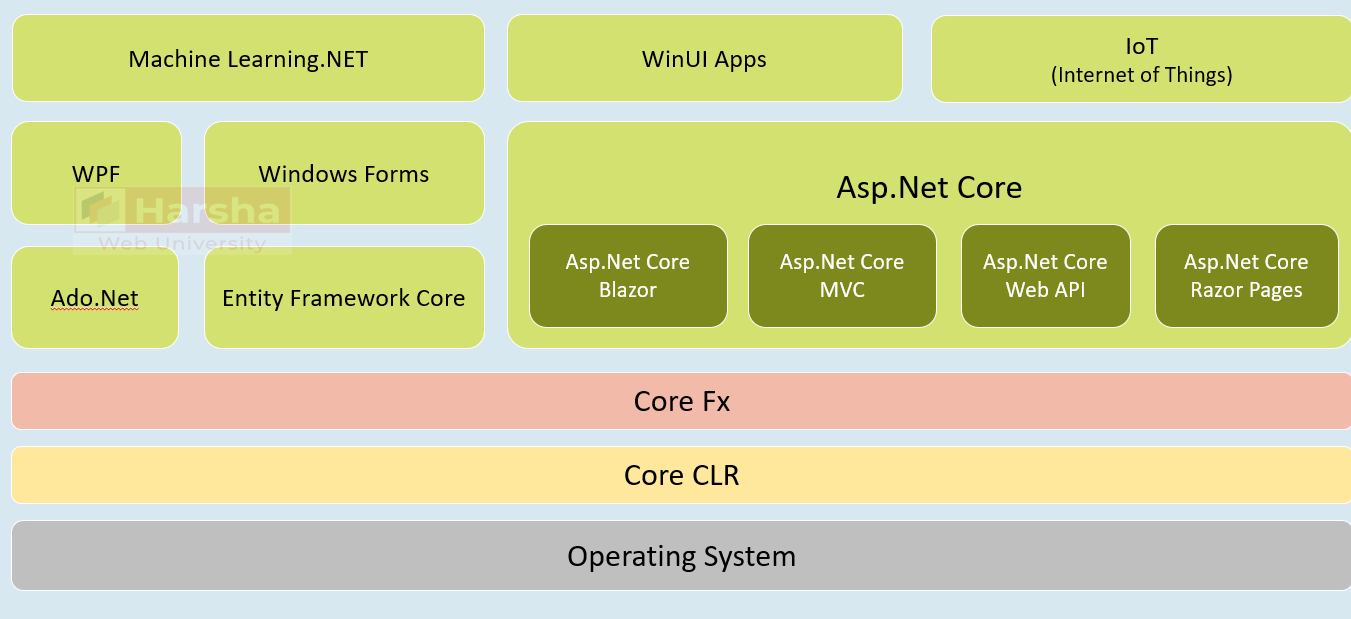
**.Net Framework**Closed-source, Monolithic, Thick, Average-performance



**.Net Core**Open-source, Modular, Cross-platform, Minimalistic, Faster-performance



**.Net**Unified, Open-source, Modular, Cross-platform, Minimalistic, Faster-performance



Top Level Statements

Allows a sequence of statements to occur right before the namespaces / type definitions in a single file in the C# project.

**File1.cs**

1. statements…
2. namespaces / types…

-- would compile as:

1. static class Program
2. {
3. static async Task Main(string[ ] args)
4. {
5. //statements…
6. }
7. }

* **Advantage:**  Make C# learning curve easy for C# learners (newbies).
* The compiler-generated class and Main method are NOT accessible through code of any other areas of the project.
* The compiled Main method would be 'async', by default. So it allows 'await' statements in top-level statements.
* Only one compilation unit (C# file) can have top level statements in a C# project.
* The local variables / local functions declared in the top-level statements are NOT accessible elsewhere (in other types / files).
* Top level statements can access command-line arguments using 'args'. A "string[ ] args" parameter would be generated by the compiler automatically.

File Scoped Namespaces

Allows you to declare a namespace at the top of the file (before/after the 'using' statements) and all types of the same file would be a part of that namespace.

**File1.cs**

1. using statements…
2. namespace namespace\_name;
3. using statements…
4. types…

-- would compile as:

1. using statements…
3. namespace namespace\_name
4. {
5. //types
6. }

* **Advantage:**  Allows developers to quickly create one-or-few types in a namespace without nesting them in the 'namespace declaration'.
* Only one 'file-scoped namespace' statement is allowed for one source file (C# file).
* The 'file-scoped namespace' statement CAN be written before / after the 'using' statements.
* A source file can't contain both 'file-scoped namespace' and 'normal namespace declarations'.

Global 'using' directives

Allows you import a namespace for the entire project, by adding 'global' keyword to the 'using' statement.

**File1.cs**

1. global using namespace\_name;
2. namespaces / types…

-- would compile as:

1. using namespace\_name; (and in other files also)
3. namespace\_names / types…

**Advantage:**Allows developers to reduce attention on lengthy 'using' statements at the top of every file; but concentrate on actual code (types in the file).

It is recommended to write all 'global using' statements in a separate file (one-for each project)

The following namespaces are implicitly imported in every C# project implicitly:

1. System
2. System.Collections.Generic
3. System.IO
4. System.Linq
5. System.Net.Http
6. System.Threading
7. System.Threading.Tasks

Module Initializers

Allows you to run some code with 'global initialization logic' at application startup, when the application loads into memory.

**File1.cs**

1. using System.Runtime.CompilerServices;
3. class class\_name
4. {
5. [ModuleInitializer]
6. internal static void method\_name( )
7. {
8. }
9. }

It would execute at application startup (before the Main method).

**Advantage over static constructors:**The static constructors execute ONLY if the class is used at least once; otherwise will NOT execute.

One project CAN have more than module initializer methods (if so, they are called based on alphabetical order of file names).

The initializer method must be:

1. Either "internal", "protected internal" or "public" only.
2. Static method
3. Parameterless method
4. Return type is 'void'
5. Not be a generic method
6. Can't be a local function

**Use Cases:**

* Loading environment variables
* Initializing connection strings / database server names
* Initializing URL's of API servers
* Loading Azure connection strings
* Initializing file paths

etc.

**The initializer class must be:**

* Either "internal" or "public" only.
* Can be static class [optionally]
* Not be a generic class

Nullable Reference Types

Introduces 'nullable reference types' and 'non-nullable reference types' to allow the compiler to perform

'static flow analysis' for purpose of null-safety.

1. class\_name variable\_name; //'class\_name' is non-nullable reference type
2. class\_name? variable\_name; //'class\_name?' is nullable reference type

**Advantage:**The compiler can perform a static analysis to identify where there is a possibility of 'null' values and can show warnings; so we can avoid NullReference Exceptions at coding-time itself.

By default, all classes and interfaces are 'non-nullable reference types'. To convert them as 'nullable reference type', suffix a question mark (?). Eg: class?

* **Null forgiving operator (!)**
* Meaning: "I'm sure, it's not null".
* Suffix your expression (variable or property) with "!" operator to make that expression as "not null", at compilation time.
* It has no effect at run time.
* It means, the developer says to the C# compiler - that, a variable or property is "not null". But at run time, if it is actually null, it leads to "NullReference Exception" as normal.
* So use this operator only when you are sure that your expression (variable of property) is NOT null.

Target-typed 'new' expressions

Allows the developer "not-to-mention" the class name; but allows to create an object in the 'new' expression.

class\_name variable\_name = new( ); //equivalent to "new class\_name( )"

**Benefit:**We can create object of a class in shortcut way.

**It can't be used in:**

1. //using block:
2. using (var variable = new( ) )
3. {
4. }
6. //foreach:
7. foreach (var variable in new( ) )
8. {
9. }

Pattern Matching - Overview

Enables developers to easily check the data type of a variable and also check its value with some conditions.

**"is" expression:**

1. if (variable\_name is class\_name another\_variable)
2. {
3. if (another\_variable.property == value)
4. {
5. statements…
6. }
7. }

**"switch-case" expression:**

1. switch (variable\_name)
2. {
3. case class\_name another\_variable
4. when another\_variable.property == value:
5. statements…
6. break;
7. }

**"switch" expression:**

1. variable\_name switch {
2. class\_name another\_variable when another\_variable.property == value => result\_expression
3. }

Pattern Matching - Type Pattern - with "if"

**Regular Code:**

1. //Check whether the variable is of specified 'class\_name' type.
2. if (variable.GetType( ) == typeof(class\_name) || variable.GetType().IsSubClassOf(typeof(class\_name))
3. {
4. statements…
5. }

**"is" expression:**

1. //Check whether the variable is of specified 'class\_name' type
2. if (variable is class\_name)
3. {
4. statements…
5. }

Pattern Matching - Type Pattern - with "if" - with Variable

**Regular Code:**

1. //Check whether the variable is of specified 'class\_name' type.
2. if (variable.GetType( ) == typeof(class\_name) || variable.GetType( ).IsSubClassOf(typeof(class\_name))
3. {
4. //typecast the value into the specified class
5. class\_name another\_variable = (class\_name)variable\_name;
7. statements…
8. }

**"is" expression:**

1. //Check whether the variable is of specified 'class\_name' type & also typecast the value into specified class.
2. if (variable is class\_name another\_variable)
3. {
4. statements…
5. }

Pattern Matching - Type Pattern - with "switch-case"

**Regular Code:**

1. //Check whether the variable is of specified 'class\_name' type.
2. switch (variable.GetType().Name)
3. {
4. case "class\_name":
5. statements; break;
6. }

**"switch-case" expression:**

1. //Check whether the variable is of specified 'class\_name' type
2. switch (variable)
3. {
4. case class\_name:
5. statements…; break;
6. }

Pattern Matching - Type Pattern - with "switch-case" - with variable

**Regular Code:**

1. //Check whether the variable is of specified 'class\_name' type.
2. switch (variable.GetType( ).Name)
3. {
4. case "class\_name":
5. //typecast the value into the specified class
6. class\_name another\_variable = (class\_name)variable\_name;
8. statements; break;
9. }

**"switch-case" expression:**

1. //Check whether the variable is of specified 'class\_name' type
2. switch (variable)
3. {
4. case class\_name another\_variable:
5. statements…; break;
6. }

Pattern Matching - Type Pattern - with 'when' - with "switch-case"

**Regular Code:**

1. //Check whether the variable is of specified 'class\_name' type.
2. switch (variable.GetType( ).Name)
3. {
4. case "class\_name":
5. //typecast the value into the specified class
6. class\_name another\_variable = (class\_name)variable\_name;
8. if (another\_variable.property == value)
9. {
10. statements;
11. }
12. break;
13. }

**"switch-case" expression:**

1. //Check whether the variable is of specified 'class\_name' type
2. switch (variable)
3. {
4. case class\_name another\_variable
5. when another\_variable.property == value:
6. statements…; break;
7. }

Pattern Matching - Type Pattern - with 'when' - with "switch expression"

**Regular Code:**

1. //Check whether the variable is of specified 'class\_name' type.
2. switch (variable.GetType( ).Name)
3. {
4. case "class\_name":
5. //typecast the value into the specified class
6. class\_name another\_variable = (class\_name)variable\_name;
8. if (another\_variable.property == value)
9. {
10. statements;
11. }
12. break;
13. }

**"switch" expression:**

1. //Check whether the variable is of specified 'class\_name' type
2. variable switch
3. {
4. class\_name another\_variable
5. when another\_variable.property == value
6. => statements…
7. }

Pattern Matching - Relational Pattern

1. //Check whether the variable is of specified 'class\_name' type
2. variable switch
3. {
4. class\_name another\_variable when
5. another\_variable.property is value //another\_variable.property == value
6. another\_variable.property is < value //another\_variable.property < value
7. another\_variable.property is > value //another\_variable.property > value
8. another\_variable.property is <= value //another\_variable.property <= value
9. another\_variable.property is >= value //another\_variable.property >= value
10. => result\_expression…
11. }

Pattern Matching - Logical Pattern

1. //Check whether the variable is of specified 'class\_name' type
2. variable switch
3. {
4. class\_name another\_variable when
5. another\_variable.property is expression1 and expression2 //conjunctive pattern (and) // another\_variable.property == expression1 && another\_variable.property == expression2
7. another\_variable.property is expression1 or expression2 //disjunctive pattern (or) //another\_variable.property == expression1 || another\_variable.property == expression2
9. another\_variable.property is not expression //negated pattern (not) // another\_variable.property != expression
11. => result\_expression…
12. }

Pattern Matching - Property Pattern

1. variable switch
2. {
3. { property: value } //variable.property == expression
4. { property: < value } //variable.property < value
5. { property: > value } //variable.property > value
6. { property: <= value } //variable.property <= value
7. { property: >= value } //variable.property >= value
8. => result\_expression…
9. }

Pattern Matching - Tuple Pattern

1. (variable.property1, variable.property2) switch
2. {
3. ( expression1, expression2 ) //variable.property1 == expression1 && variable.property2 == expression2
4. => result\_expression…
6. ( expression1, expression2 ) //variable.property1 == expression1 && variable.property2 == expression2
7. => result\_expression…
8. }

Pattern Matching - Positional Pattern

1. variable switch {
2. ( expression1, expression2 ) //variable.property1 == expression1 && variable.property2 == expression2
3. => result\_expression…
5. ( expression1, expression2 ) //variable.property1 == expression1 && variable.property2 == expression2
6. => result\_expression…
7. }

**Deconstruct method:**

1. public void Deconstruct(out type1 variable1, out type2 variable2)
2. {
3. variable1 = this.property1;
4. variable2 = this.property2;
5. }

Pattern Matching - Nested Property Pattern

1. variable switch
2. {
3. { outer\_property: { nested\_property: value } } //variable.outer\_property.nested\_property == expression
4. { outer\_property: { nested\_property: < value } } //variable.outer\_property.nested\_property < value
5. { outer\_property: { nested\_property: > value } } //variable.outer\_property.nested\_property > value
6. { outer\_property: { nested\_property: <= value } } //variable.outer\_property.nested\_property <= value
7. { outer\_property: { nested\_property: >= value } } //variable.outer\_property.nested\_property >= value
8. => result\_expression…
9. }

Pattern Matching - Extended Property Pattern

1. variable switch
2. {
3. { outer\_property.nested\_property: value } //variable.outer\_property.nested\_property == expression
4. { outer\_property.nested\_property: < value } //variable.outer\_property.nested\_property < value
5. { outer\_property.nested\_property: > value } //variable.outer\_property.nested\_property > value
6. { outer\_property.nested\_property: <= value } //variable.outer\_property.nested\_property <= value
7. { outer\_property.nested\_property: >= value } //variable.outer\_property.nested\_property >= value
8. => result\_expression…
9. }

Need of Immutability

**Goal:**The values of fields and properties should be readonly (immutable). No other classes can change them, after they get initialized.

**Immutable class:**

1. class class\_name
2. {
3. data\_type readonly field\_name; //readonly field
5. data\_type property\_name { get => field\_name } //readonly property
6. }

**Benefits:**

* Avoid unexpected value changes in response data retrieved from API servers.
* Avoid unexpected value changes in the data retrieved from database servers.
* Use objects of immutable classes as 'key' in Dictionary and in Hashtable.
* Avoid unexpected value changes in objects while multiple threads access the same objects simultaneously.

Immutable Classes

A class with readonly fields and readonly properties.

**Immutable class:**

1. class class\_name
2. {
3. data\_type readonly field\_name; //readonly field
5. data\_type property\_name { get => field\_name } //readonly property
7. public class\_name( ) //constructor
8. {
9. field\_name = value; //initialize the field
10. }
11. }

'init' only properties

'init' only properties can be initialized either inline with declaration, in the constructor or in the object initializer.

**Init-only property**

1. data\_type property\_name { get; init; } //'init' instead of 'set'

**Immutable class with 'init' only properties**

1. class class\_name
2. {
3. data\_type readonly field\_name; //readonly field
5. data\_type property\_name
6. {
7. get => field\_name //get accessor
8. init => field\_name = value; //init accessor instead of 'set' accessor
9. }
11. public class\_name( ) //constructor
12. {
13. field\_name = value; //initialize the field
14. }
15. }

**Object of immutable class:**

1. class variable\_name = new class\_name( ) { property\_name = value; } //initialize value of 'init' only property in object initializer

Readonly structs

Enforces you to write only 'readonly fields' and 'readonly properties' to achieve immutability in your struct.

**Readonly struct**

1. readonly struct struct\_name
2. {
3. //readonly fields
4. //readonly properties
5. }

**Readonly struct**

1. readonly struct struct\_name
2. {
3. data\_type readonly field\_name; //readonly field
5. data\_type property\_name
6. {
7. get => field\_name //get accessor
8. init => field\_name = value; //init accessor instead of 'set' accessor
9. }
11. public class\_name( ) //constructor
12. {
13. field\_name = value; //initialize the field
14. }
15. }

Parameterless Struct Constructors

**Struct with parameter-less constructor**

1. readonly struct struct\_name
2. {
3. data\_type readonly field\_name; //readonly field
5. data\_type property\_name
6. {
7. get => field\_name //get accessor
8. init => field\_name = value; //init accessor instead of 'set' accessor
9. }
11. public class\_name( ) //constructor
12. {
13. field\_name = value; //you must initialize all the fields
14. }
15. }

Records

**Goal:**Concise syntax to create a reference-type with immutable properties.

**Record**

record record\_name(data\_type Property1, data\_type Property2, …);

-- would be compiled as:

**Compiled code of Record**

1. class record\_name
2. {
3. public data\_type Property1 { get; init; }
4. public data\_type Property2 { get; init; }
6. public record\_name(data\_type Parameter1, data\_type Parameter2)
7. {
8. this.Property1 = Parameter1;
9. this.Property2 = Parameter2;
10. }
11. }

**Features:**

* Records are 'immutable' by default.
* All the record members become as 'init-only' properties.
* Records can also be partially / fully mutable - by adding mutable properties.
* Supports value-based equality.
* Supports inheritance.
* Supports non-destructive mutation using 'with' expression.

**Records with mutable properties:**

1. record record\_name(data\_type Property\_name, …)
2. {
3. data\_type Property\_name { get; set; }
4. }

Records - Equality

Supports value-based equality.

Records provide a compiler-generated Equals() method and overloads == and != operators that compares two instances of records that compare the values of fields (but doesn't compare references) .

**Record - 'Equals' method**

1. record\_name variable1 = new record\_name(value1, value2);
2. record\_name variable2 = new record\_name(value1, value2);
3. variable1 == variable2; //true
4. variable1.Equals(variable2); //true

Records - "with" expression

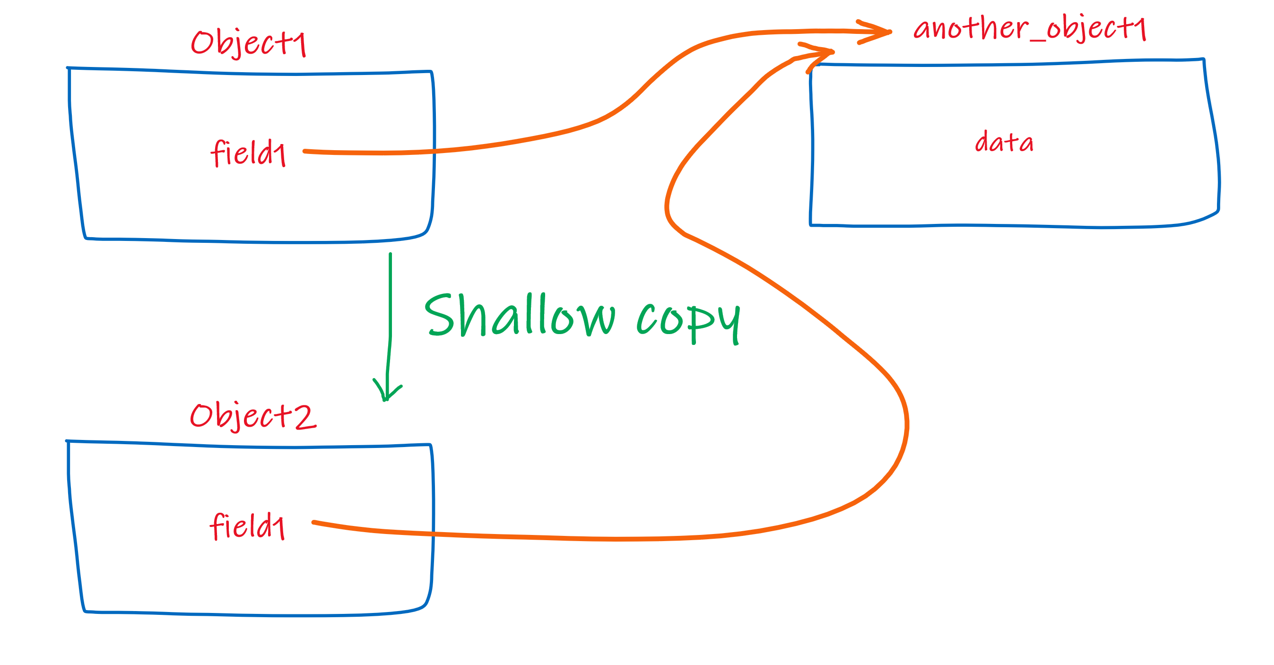
'with' expression acts as object initializer for 'records'.

It creates a shallow copy of an existing record object and also overwrites the values of specified properties.

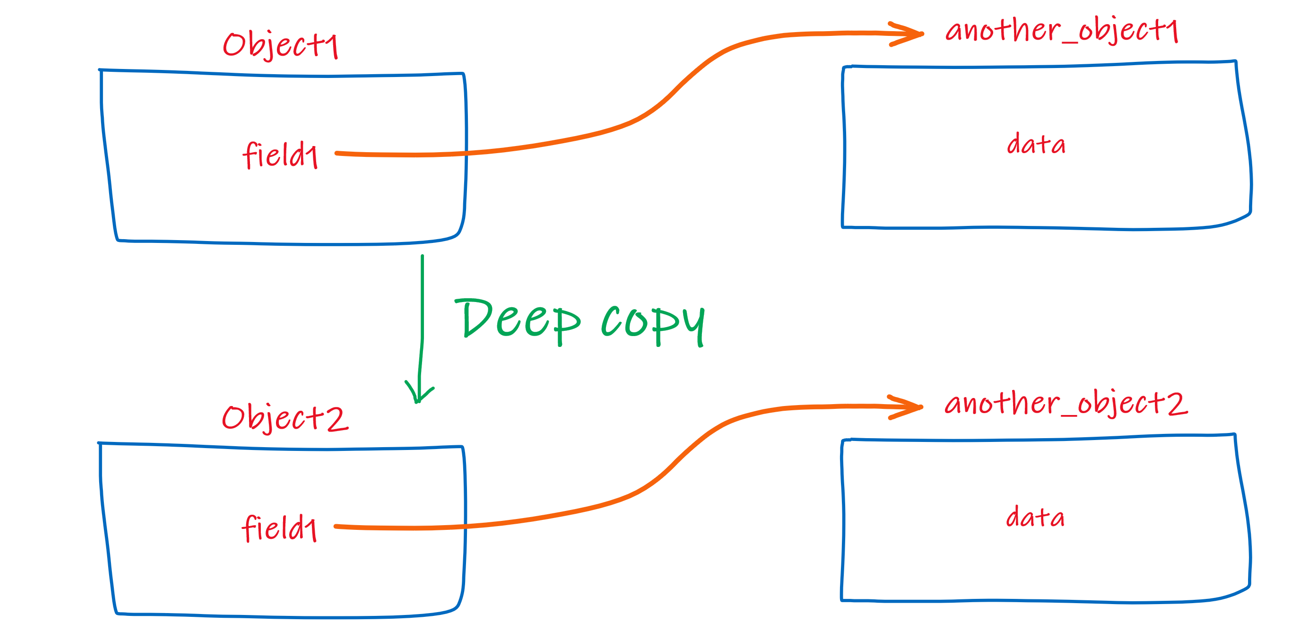
**Record - 'with' expression**

1. record\_name variable1 = new record\_name(value1, value2);
2. record\_name variable2 = variable1 with { Property = value, … } //with expression

**Shallow copy:**



**Deep copy:**



Records - "Deconstruct"

A compiler-generated 'Deconstruct' method is provided for all records that returns all property values as a tuple.

It is useful while reading few specific set of properties from a record object.

**Record - deconstruct'**

1. record\_name reference\_variable = new record\_name(value1, value2);
2. var (variable1, variable2, …) = reference\_variable;

Records - ToString()

A compiler-generated 'ToString()' is provided for all records that returns a string with all properties and values.

1. public record record\_name(Properties\_list)
2. {
3. public override string ToString( ) //compiler-generated
4. {
5. //returns a string: Record\_Name { Property1 = value1, Property2 = value2, … }
6. }
7. }

You can override that compiler-generated 'ToString()' with 'override' keyword.

**Record - ToString() - User defined**

1. public record record\_name(Properties\_list)
2. {
3. public override string ToString( ) //user-defined
4. {
5. //return any string
6. }
7. }

Records - Constructor

A compiler-generated 'parameterized constructor' is provided for all records that initializes all property values.

You must invoke the compiler-generated constructor of the record with 'this' keyword, in case if you create your own constructor.

**Record - User-Defined Constructor**

1. public record\_name(parameters): this(parameters) //invokes compiler-generated constructor
2. {
3. Property = value;
4. }

Records - Inheritance

A record can inherit from another record.

1. public record Parent\_record\_name(Properties\_list);
2. public record Child\_record\_name(Properties\_list) : Parent\_record\_name;

* A record CAN inherit from another record.
* A record CAN'T inherit from another class.
* A class CAN'T inherit from another record.
* A record CAN implement (inherit) one or more interfaces.
* A record CAN be 'abstract' and 'sealed'.

Records - Sealed ToString()

The user-defined 'override ToString()' can be 'sealed', in order to prevent further overriding.

1. public record record\_name(Properties\_list)
2. {
3. public override sealed string ToString( ) //user-defined
4. {
5. //return any string
6. }
7. }

Record Structs

**Record [or] Record class**

record record\_name(Properties\_list);

* A record is a class internally (after compilation).
* All positional parameters of a record are init-only properties by default.

**Readonly record struct**

readonly record struct record\_name(Properties\_list);

* A readonly record struct is a 'struct' internally (after compilation).
* All positional parameters of a readonly record struct are init-only properties by default.

**Record struct**

record struct record\_name(Properties\_list);

* A record struct is a 'struct' internally (after compilation).
* All positional parameters of a record struct are read-write properties by default.

Command Line Arguments

**Goal:**Supply inputs from the command line / terminal to an application.

app.exe value1 value2

-- will be converted as array:

**Code that receives arguments from command line / terminal**

1. class class\_name
2. {
3. static void Main(string[ ] args)
4. {
5. //Use args
6. }
7. }

**Features:**

* CLR converts all the command line arguments (space-separated values) as string array (string[ ]) only (in the same order).
* The Main method can't have additional arguments - other than
* string[ ] args.
* The parameter name 'args' isn't fixed. You can give it any other name.
* Top level statements have an implicit parameter called 'args' of string[ ] type, which contains the command line argument s received.

Improvements in Partial Methods in C# 9

* A partial method CAN have any return type (not-only 'void') in C# 9.
* A partial method CAN have any access modifier in C# 9.
* A partial method CAN have 'out' parameters in C# 9.
* Partial methods must have a definition in any one of the parts of the same partial class.

**Partial Method in C# 9**

public partial return\_type Method\_name(Parameters\_list);

Static Anonymous Functions

* A static anonymous function is an anonymous method or lambda expression, prefixed with 'static' keyword.
* It CAN'T access the state (local variables, parameters, 'this' keyword and 'base' keyword) of the enclosing method; and also CAN'T access instance members of enclosing type.
* It CAN access static members and constants of enclosing type.

**Static anonymous function (anonymous method)**

1. static delegate (Parameters\_list)
2. {
3. //can't access locals, parameters, instance members
4. //can access static members and constants
5. }

**Static anonymous function (lambda expression)**

1. static (Parameters\_list) =>
2. {
3. //can't access locals, parameters, instance members
4. //can access static members and constants
5. }

Return Type of Lambda Functions

A lambda expression (or lambda function) can have a return type before the list of parenthesized parameters.

Useful when you return a value of any one of two or more types.

**Lambda Function Return Type in C# 10**

return\_type (Parameters\_list) => return\_value;

Constant Interpolated Strings

Constant strings may be initialized using 'string interpolation' i.e. with ${ }, if all the placeholders are constant strings.

Eg: Useful when you are creating global API URLs.

**Constant Interpolated Strings**

const data\_type variable\_name = $"{constant\_string}";

Interface Default Methods

Default methods are methods in interfaces with concrete implementation.

These methods are accessible through a reference variable of the interface type.

**Interface Default Methods**

1. interface interface\_name
2. {
3. access\_modifier return\_type method\_name(parameters)
4. {
5. //method body
6. }
7. }

Access Modifiers on Interface Methods

Interface methods can have any access modifiers including private, protected, internal, protected internal, private protected and public

Non-public interface methods can be either implemented as public or explicitly (to preserve the same access modifier).

**Interface methods with access modifier**

1. interface interface\_name
2. {
3. access\_modifier return\_type method\_name(parameters);
4. }

Interface Private Methods

Private interface methods must have method body.

**Private interface method**

1. interface interface\_name
2. {
3. private return\_type method\_name(parameters)
4. {
5. //method body
6. }
7. }

Interface Static Methods

Static methods are allowed with concrete implement in interface.

Interface static methods can be called through the interface name.

**Interface Static Methods**

1. interface interface\_name
2. {
3. access\_modifier static return\_type method\_name(parameters)
4. {
5. //method body
6. }
7. }
9. interface\_name.method\_name(arguments); //calling the interface static method