C# - Ultimate Guide - Beginner to Advanced | Master class

Section 30 – C# 9 and 10 (.NET 6) - New Features

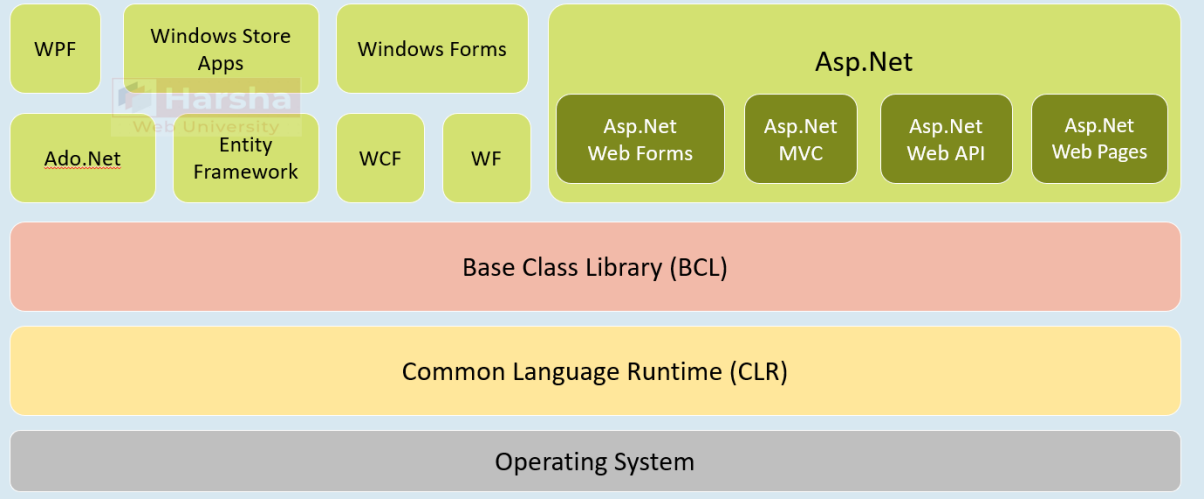
.Net Framework [vs] .Net Core [vs] .Net

.Net Framework: Since 2002

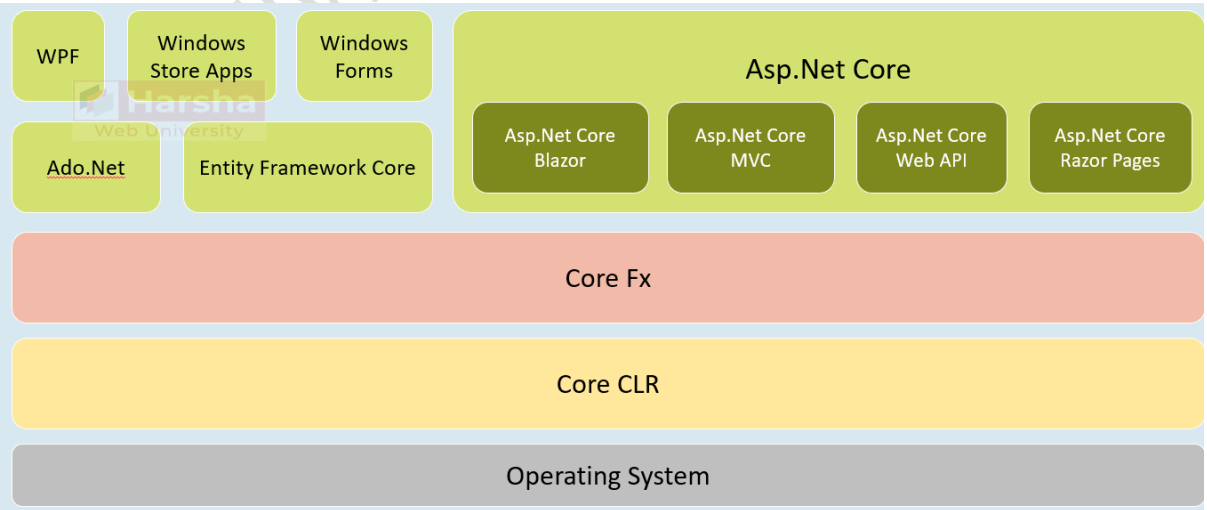
.Net Core: Since 2016

.Net: Since 2020

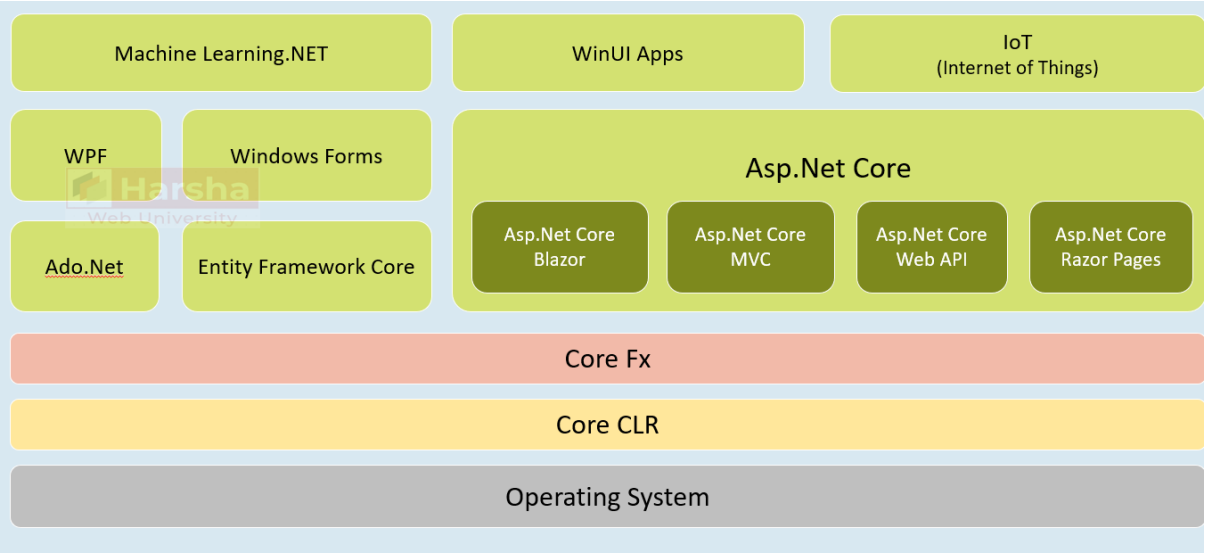
.Net Framework Closed-source, Monolithic, Thick, Average-performance



.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

```
statements…

namespaces / types…

-- would compile as:

static class Program

{

  static async Task Main(string[ ] args)

  {

    //statements…

  }

}
```

- **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

using statements…

namespace namespace_name;

using statements…

types…

-- would compile as:

using statements…

namespace namespace_name

{

  //types

}

- **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

global using namespace_name;

namespaces / types…

-- would compile as:

using namespace_name; (and in other files also)

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

```
using System.Runtime.CompilerServices;


class class_name
{
  [ModuleInitializer]
  internal static void method_name( )
  {
  }
}
```

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.

class_name variable_name; //'class_name' is non-nullable reference type

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:

```
//using block:
using (var variable = new( ) )
{
}

//foreach:
foreach (var variable in new( ) )
{
}
```

Pattern Matching - Overview

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

"is" expression:

```
if (variable_name is class_name another_variable)
{
  if (another_variable.property == value)
  {
    statements…
  }
}
```

"switch-case" expression:

```
switch (variable_name)
{
  case class_name another_variable
      when another_variable.property == value:
        statements…
  break;
}
```

"switch" expression:

```
variable_name switch {
  class_name another_variable when another_variable.property == value  => result_expression
}
```

Pattern Matching - Type Pattern - with "if"

Regular Code:

```
//Check whether the variable is of specified 'class_name' type.

if (variable.GetType( ) == typeof(class_name) ||
variable.GetType().IsSubClassOf(typeof(class_name))

{

  statements…

}
```

"is" expression:

```
//Check whether the variable is of specified 'class_name' type

if (variable is class_name)

{

  statements…

}
```

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

Regular Code:

```
//Check whether the variable is of specified 'class_name' type.

if (variable.GetType( ) == typeof(class_name) || variable.GetType(
).IsSubClassOf(typeof(class_name))

{

  //typecast the value into the specified class

  class_name another_variable = (class_name)variable_name;


  statements…

}
```

"is" expression:

//Check whether the variable is of specified 'class_name' type & also typecast the value into specified class.

```
if (variable is class_name another_variable)
{
  statements…
}
```

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

Regular Code:

//Check whether the variable is of specified 'class_name' type.

```
switch (variable.GetType().Name)
{
  case "class_name":
    statements; break;
}
```

"switch-case" expression:

//Check whether the variable is of specified 'class_name' type

```
switch (variable)
{
  case class_name:
  statements…; break;
}
```

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

Regular Code:

```
//Check whether the variable is of specified 'class_name' type.
switch (variable.GetType( ).Name)
{
  case "class_name":
     //typecast the value into the specified class
     class_name another_variable = (class_name)variable_name;

     statements; break;
}
```

"switch-case" expression:

```
//Check whether the variable is of specified 'class_name' type
switch (variable)
{
  case class_name another_variable:
    statements…; break;
}
```

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

Regular Code:

```
//Check whether the variable is of specified 'class_name' type.
switch (variable.GetType( ).Name)
{
   case "class_name":
        //typecast the value into the specified class
        class_name another_variable = (class_name)variable_name;

        if (another_variable.property == value)
        {
              statements;
        }
        break;
}
```

"switch-case" expression:

```
//Check whether the variable is of specified 'class_name' type
switch (variable)
{
    case class_name another_variable
        when another_variable.property == value:
          statements…; break;
}
```

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

Regular Code:

```
//Check whether the variable is of specified 'class_name' type.
switch (variable.GetType( ).Name)
{
  case "class_name":
    //typecast the value into the specified class
    class_name another_variable = (class_name)variable_name;

    if (another_variable.property == value)
    {
        statements;
    }
    break;
}
```

"switch" expression:

```
//Check whether the variable is of specified 'class_name' type
variable switch
{
   class_name another_variable
       when another_variable.property == value
       => statements…
}
```

Pattern Matching - Relational Pattern

```
//Check whether the variable is of specified 'class_name' type
variable switch
{
  class_name another_variable when
    another_variable.property is value //another_variable.property == value
    another_variable.property is < value //another_variable.property < value
    another_variable.property is > value //another_variable.property > value
    another_variable.property is <= value //another_variable.property <= value
    another_variable.property is >= value //another_variable.property >= value
    => result_expression…
}
```

Pattern Matching - Logical Pattern

```
//Check whether the variable is of specified 'class_name' type
variable switch
{
  class_name another_variable when
    another_variable.property is expression1 and expression2 //conjunctive pattern (and)  //
another_variable.property == expression1 && another_variable.property == expression2

    another_variable.property is expression1 or expression2 //disjunctive pattern (or)
//another_variable.property == expression1 || another_variable.property == expression2

    another_variable.property is not expression //negated pattern (not)  //
another_variable.property != expression

    => result_expression…
}
```

Pattern Matching - Property Pattern

variable switch

{

  { property: value } //variable.property == expression

  { property: < value } //variable.property < value

  { property: > value } //variable.property > value

  { property: <= value } //variable.property <= value

  { property: >= value } //variable.property >= value

  => result_expression…

}

Pattern Matching - Tuple Pattern

(variable.property1, variable.property2) switch

{

  (expression1, expression2) //variable.property1 == expression1 && variable.property2 == expression2

  => result_expression…

  (expression1, expression2) //variable.property1 == expression1 && variable.property2 == expression2

  => result_expression…

}

Pattern Matching - Positional Pattern

variable switch {

(expression1, expression2) //variable.property1 == expression1 && variable.property2 == expression2

=> result_expression…

(expression1, expression2) //variable.property1 == expression1 && variable.property2 == expression2

=> result_expression…

}

Deconstruct method:

public void Deconstruct(out type1 variable1, out type2 variable2)

{

variable1 = this.property1;

variable2 = this.property2;

}

Pattern Matching - Nested Property Pattern

variable switch

{

{ outer_property: { nested_property: value } } //variable.outer_property.nested_property == expression

{ outer_property: { nested_property: < value } } //variable.outer_property.nested_property < value

{ outer_property: { nested_property: > value } } //variable.outer_property.nested_property > value

{ outer_property: { nested_property: <= value } } //variable.outer_property.nested_property <= value

{ outer_property: { nested_property: >= value } } //variable.outer_property.nested_property >= value

=> result_expression…

}

Pattern Matching - Extended Property Pattern

```
variable switch
{
  { outer_property.nested_property: value } //variable.outer_property.nested_property == expression
  { outer_property.nested_property: < value } //variable.outer_property.nested_property < value
  { outer_property.nested_property: > value } //variable.outer_property.nested_property > value
  { outer_property.nested_property: <= value } //variable.outer_property.nested_property <= value
  { outer_property.nested_property: >= value } //variable.outer_property.nested_property >= value
  => result_expression…
}
```

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:

```
class class_name
{
  data_type readonly field_name; //readonly field

  data_type property_name { get => field_name } //readonly property
}
```

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:

```
class class_name
{
  data_type readonly field_name; //readonly field

  data_type property_name { get => field_name } //readonly property

  public class_name( ) //constructor
  {
    field_name = value; //initialize the field
  }
}
```

'init' only properties

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

Init-only property

```
data_type property_name { get; init; } //'init' instead of 'set'
```

Immutable class with 'init' only properties

```
class class_name
{
  data_type readonly field_name; //readonly field

  data_type property_name
  {
    get => field_name //get accessor
    init => field_name = value; //init accessor instead of 'set' accessor
  }

  public class_name( ) //constructor
  {
    field_name = value; //initialize the field
  }
}
```

Object of immutable class:

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

```
readonly struct struct_name
{
  //readonly fields
  //readonly properties
}
```

Readonly struct

```
readonly struct struct_name
{
  data_type readonly field_name; //readonly field

  data_type property_name
  {
    get => field_name //get accessor
    init => field_name = value; //init accessor instead of 'set' accessor
  }

  public class_name( ) //constructor
  {
    field_name = value; //initialize the field
  }
}
```


Parameterless Struct Constructors

Struct with parameter-less constructor

```
readonly struct struct_name
{
  data_type readonly field_name; //readonly field
```

```
  data_type property_name
  {
    get => field_name //get accessor
    init => field_name = value; //init accessor instead of 'set' accessor
  }

  public class_name( ) //constructor
  {
    field_name = value; //you must initialize all the fields
  }
}
```

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

```
class record_name
{
  public data_type Property1 { get; init; }
  public data_type Property2 { get; init; }

  public record_name(data_type Parameter1, data_type Parameter2)
  {
    this.Property1 = Parameter1;
    this.Property2 = Parameter2;
  }
}
```

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:

```
record record_name(data_type Property_name, …)
{
  data_type Property_name { get; set; }
}
```

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

```
record_name variable1 = new record_name(value1, value2);
record_name variable2 = new record_name(value1, value2);
variable1 == variable2; //true
variable1.Equals(variable2); //true
```

Records - "with" expression

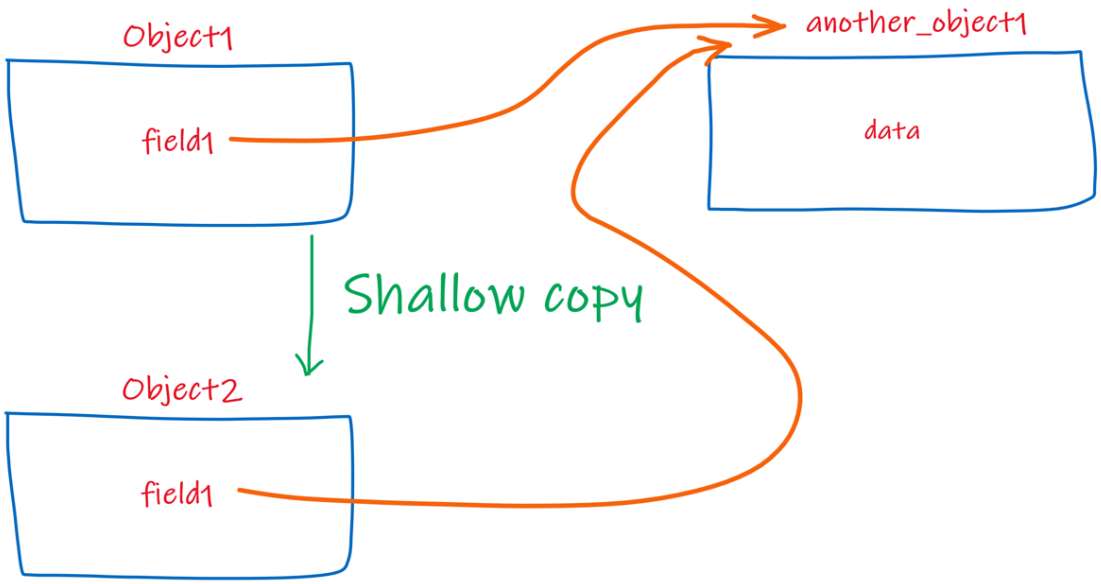
'with' expression acts as object initializer for 'records'.

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

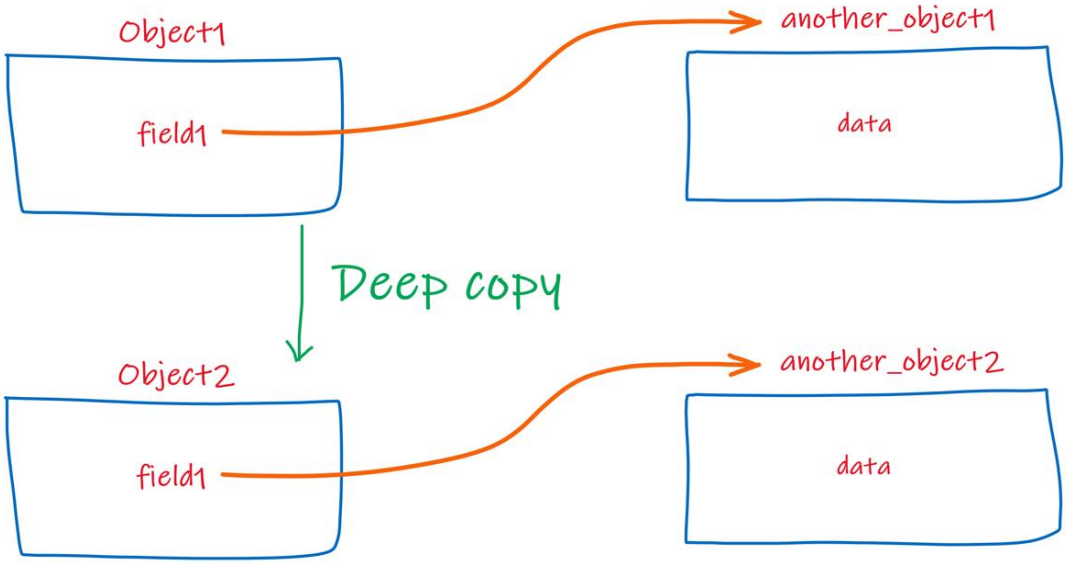
Record - 'with' expression

```
record_name variable1 = new record_name(value1, value2);
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'

record_name reference_variable = new record_name(value1, value2);

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.

```
public record record_name(Properties_list)
{
  public override string ToString( ) //compiler-generated
  {
    //returns a string: Record_Name { Property1 = value1, Property2 = value2, … }
  }
}
```

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

Record - ToString() - User defined

```
public record record_name(Properties_list)
{
  public override string ToString( ) //user-defined
  {
    //return any string
  }
}
```

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

```
public record_name(parameters): this(parameters) //invokes compiler-generated constructor
{
  Property = value;
}
```

Records - Inheritance

A record can inherit from another record.

```
public record Parent_record_name(Properties_list);
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.

```
public record record_name(Properties_list)
{
  public override sealed string ToString( ) //user-defined
  {
    //return any string
  }
}
```

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

```
class class_name
{
  static void Main(string[ ] args)
  {
    //Use args
  }
}
```

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)

```
static delegate (Parameters_list)
{
  //can't access locals, parameters, instance members
  //can access static members and constants
}
```

Static anonymous function (lambda expression)

```
static (Parameters_list) =>
{
  //can't access locals, parameters, instance members
  //can access static members and constants
}
```

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

```
interface interface_name
{
  access_modifier return_type method_name(parameters)
  {
    //method body
  }
}
```

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

```
interface interface_name
{
  access_modifier return_type method_name(parameters);
}
```

Interface Private Methods

Private interface methods must have method body.

Private interface method

```
interface interface_name
{
  private return_type method_name(parameters)
  {
    //method body
  }
}
```

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

```
interface interface_name
{
  access_modifier static return_type method_name(parameters)
  {
    //method body
  }
}
```

```
interface_name.method_name(arguments); //calling the interface static method
```