**OOP 4 pillars**

Abstraction - simplify requirements to initial set of classes

Encapsulation - use getters and setters to hide complexity

Inheritance - code reuse through class or constructor inheritance

Polymorphism - method can have the same signature, but different implementations in different classes (eg. inherit method from abstract base class)

**SOLID principles**

**SOLID advantages: easy to modify/extend functionality, easy to maintain, testability, debugging, paralel coding, readability**

1. Single responsibility – every class has to do only one job.

2. Open/Closed – class should be open for extension (inheritance) and closed for modification

3. Liskov Substitution – any child class derived from base class should behave the same without modification

4. Interface segregation – don’t use large interfaces. Use smaller interfaces, so that other classes would not be forced to use what they don’t need.

5. Dependency inversion – high level classes should not depent on low leves classes. Both should depent on abstractions.

**Readonly property** & **static readonly property**

1. Readonly property with value set on initiaization. Value cannot be changed as property has no setter.

2. It is a run-time constant value

3. Readonly properties are defined above constructor(s) in the class

public string ProductId { get; } = Guid.NewGuid().ToString();

OR

public readonly int ProductId = 123;

4. Static Readonly property value cannot be changed after run-time

5. Its value is the same for all instances

Public static readonly int ClientSinceYear = DateTime.Now.Year;

**// Readonly property**

1. Read-only field value can be initialized once, either via constructor or in the declaration.

2. If we mark readonly field as static it will be the same for all instances

public readonly string VendorId;

OR

public string VendorReadonlyId { get; } = Guid.NewGuid().ToString();

// **public, private and protected access modifiers**

private modifier – access class memebers only in the same class

protected modifier – access class members with the same class or its derived classes

// **Readonly property vs Private property**

1. Use **private set** when you want **setter can't be accessed from outside**.

2. Use **readonly** when you want to **set the property only once**. In the constructor or variable initializer.

**// Static class**

1. declared with static keywod

2. Members must be also static

3. Cannot be instantiated with new()

4. Often used a as util class

5. Class is called using class name

**// Static property/method**

1. Can be called only using Class, not the instance

public class Customer {

public Customer() { InstanceCount++; }

public static int InstanceCount { get; set; }

}

Customer.InstanceCount;

**// Abstract class:**

1. is an incomplete class

2. can't be instantiated with new()

3. abstract is used to create a base class. eg. .ToString(), Math.Round() etc.

4. abstract class can have abstract and non-abstract members. Abstract memebers are marked as abstract ant must be overriden by child class that inherits from abstract class

**//Interface – (as suprantu interface kaip custom type)**

0. Using interface-based design concepts provides loose coupling, easier maintainability, makes your code base more scalable and makes code reuse. It indicates what sort of methods, properties, and events are exposed by an object.

1. Is a ‘contract’ that is implemented by a class

2. Interface is defined using interface keyword

2. Interface holds properties (not fields) and method signatures that must be implemented in a class

3. It memebers holds no values or logic, these are provided by classes that implements an interface

4. By using interfaces, you can include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes.

**//Abstract class vs Interface**

1. viena klase gali paveldeti tik is 1 abstract clases, bet gali paveldeti is daugelio interfeisu

**// Inheritance based polymorphism vs Interface based polymorphism**

1. Inheritance based P is implemented by overriding abstract class virtual or abstract methods

2. Interface based P is implemented by passing T<Interface> that exposes the same method that is implemented in different classes

**// Sealed class:**

1. is a concrete class

2. it's members cannot be overriden

3. cannot be extended through inheritance

//**Abstract method vs. Virtual method**

1. abstract method has no logic, it is just a placeholder that must be overriden

2. abstract method can only be used in an abstract class

3. Abstract methods allow only declaration, they have no body

4. Virtual method has a default logic, that can be overriden by child class throught inheritance if needed

5. Virtual method can be used in an abstract or sealed class

**// Overloading vs overriding**

1. Overloading is used to describe methods that have same name but different parameters

eg. Save() and Save(int id)

2. Since overloaded methods have the same name, they have to provide same functionality.

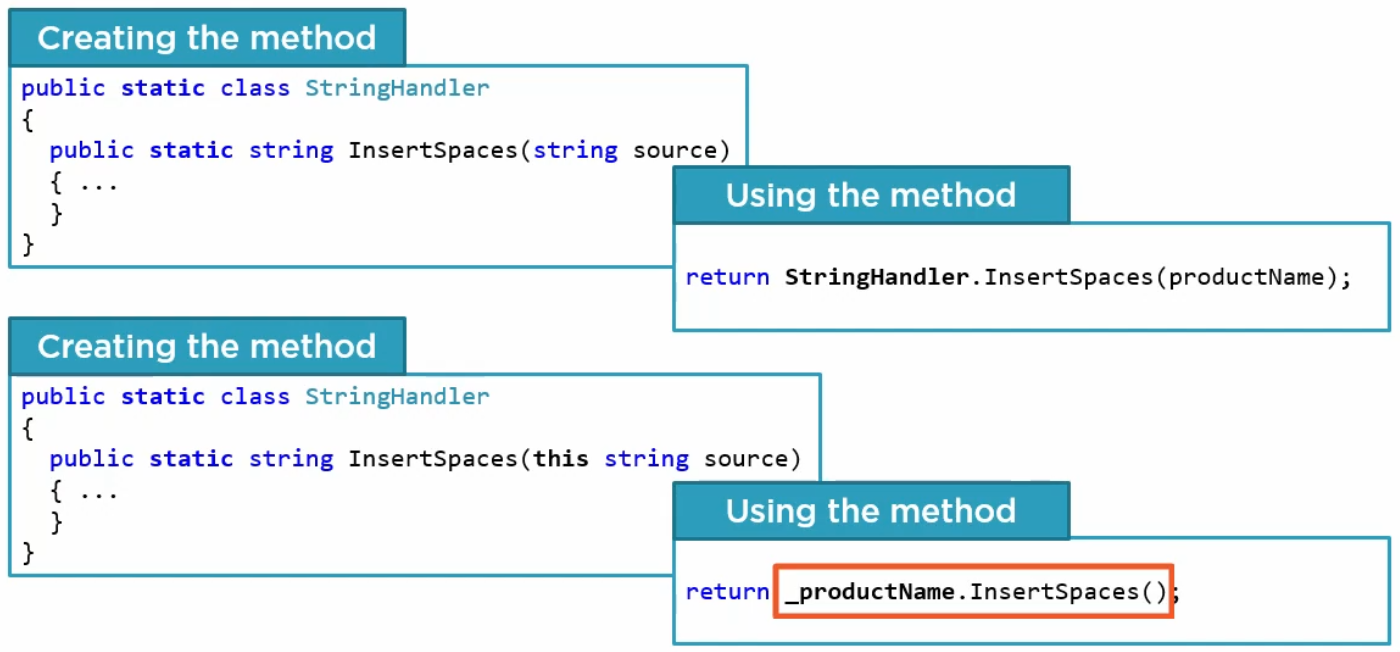
3. Overriding describes usage of base class abstract or virtual method that is overriden by method of child class

**//Extension method**

1. Used to add methods to .NET types

2. Extension methods appears in intelisense

3. Must be a static method in a static class



**List vs Dictionary**

**Link:** [**https://zetcode.com/lang/csharp/collections/**](https://zetcode.com/lang/csharp/collections/)

1. A list is a group of items that can be accessed by index

List<string> myList = new List<string>() {

   "Maths",

   "English",

"   Science"

};

OR

var myList = new List<string>() {… values here}

2. Dictionary is a set of key-value pairs.

Dictionary<string, int> d = new Dictionary<string, int>();

d.Add("squash", 1);

d.Add("football", 2);

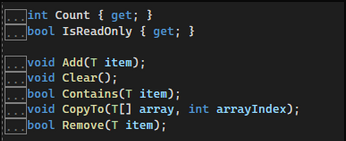
d.Add("rugby", 3);

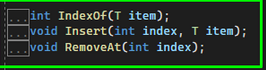
3. Looping is easier and faster in a list and access element using index easily with a List.

\*/

// IEnumerable => Icollection +> Ilist

GetEnumerator() – get next element





**ArrayList vs Array**

1. Unlike arrays, an ArrayList can hold data of multiple data types. Elements in the ArrayList are accessed via an integer index.

var data = new ArrayList();

data.Add("Visual Basic");

data.Add(344);

2. An array is the data structure that stores a fixed number of literal values (elements) of the same [data type](https://www.tutorialsteacher.com/csharp/csharp-data-types).

3. If you are adding array elements at the time of declaration, then size is optional. The compiler will infer its size based on the number of elements

int[] evenNums = new int[5]{ 2, 4, 6, 8, 10 };

var evenNums = new int[2]{2, 4};

or

int[] = { 2, 4, 6, 8, 10 };

//must specify the size

int[] evenNums = new int[];

//number of elements must be equal to the specified size

int[] evenNums = new int[5] { 2, 4 };

//cannot use var with array initializer

var evenNums = { 2, 4, 6, 8, 10};

**List of string arrays**

var listOfArrays = new List<string[]> { new string[] { "abba", "baba" }, new string[] { "babas" }, new string[] { "pienas", "ainpes", "naspie" } };

/\* **Related objects**

Three vays to instanciate related objects:

1. If related object is used only in one method/property - just create the instance inside that method and use it

2. If related object is allways used in the class - create related object as a property (don't forget to instanciate related object instance in the constructor)

3. If related object is sometimes needed in the class - use lazy-loading technique to instanciate related object only when needed.

So instead of instantiating related object in the default constructor on every instance creation. Create new instance when the getter is called

\*/

**//Enums**

1. is used to assign constant names to a group of numeric integer values.

2. An enum is defined using the enum keyword, directly inside a namespace, class, or structure.

enum WeekDays

{

Monday = 1, // 1

Tuesday, // 2

Wednesday, // 3

Thursday, // 4

Friday, // 5

Saturday, // 6

Sunday // 7

}

// access value var wd = (WeekDays) 5; // Friday

// access index int day = (int) WeekDays.Friday; // 5

ABSTRACT CLASS

namespace Povil

{

public abstract class Vehicle

{

public Vehicle(string color, int ratuSk)

{

\_color = color;

RatuSkaicius = ratuSk;

}

protected string \_color; // prop can be used only with derived classes

public int RatuSkaicius { get; set; }

public string Color

{

get { return \_color; }

set { \_color = value; }

}

public abstract void Repaint(string newColor);

}

CHILD CLASS

public class Motociklas : Vehicle

{

protected readonly string BikeId;

public Motociklas(string color, int ratuSk, BikeTypes motocikloTipas) : base(color, ratuSk)

{

BikeId = Guid.NewGuid().ToString();

MotocikloTipas = motocikloTipas;

}

public BikeTypes MotocikloTipas { get; set; }

public override void Repaint(string newColor)

{

\_color = "black";

}

}

}

TEST

[Fact]

public void ReturnNthFibElement()

{

//arrange

var sut = new FibonacciService();

var result = sut.GetNthFibElement(6);

//expected

var expected = 8;

//assert

Assert.Equal(expected, result);

}