Pentalog

C# FEATURES
OOP Recap

PentaStagiu Remote Brașov

November, 2019 – March, 2020

DATA ANALYSIS
UX/UI
FRONT-END
TECHNOLOGY
NEAR/OFFSHORE
AGILITY
BACK-END
SPRINT
KANBAN
CAMPAIGNS
GROWTH HACKING
SCRUM
BACKLOG
DEVOPS
DESIGN
SEO
CONTINUOUS INTEGRATION
MOBILE
QA
AUTOMATION
RESPONSIVE
UNIT TESTING

Agenda

C# Features

- Type conversions
- Boxing and unboxing
- Var and dynamic
- Indexing operators
- Delegates and events

OOP Recap

- SOLID design principles
- OOP design principles

Type conversions

- In C#, after you declare a variable with a certain type, you cannot change its type or assign values of another type to it

- If you need to copy a value into a variable of a different type, you need to convert it first

There are several types of conversions available in C#:

- Implicit conversions
- Explicit conversions (casts)
- User-defined conversions
- Conversions using helper classes

IMPLICIT CONVERSIONS

- Implicit conversions are conversions that can be made safely without losing data or precision. For example: converting from smaller to larger integral types

```
int i = 123456;
long l = i;
```

- Reference types can be implicitly converted from a derived class to any of its base classes or interfaces

```
Derived d = new Derived();
Base b = d;
```

EXPLICIT CONVERSIONS

- For built-in types, when a conversion cannot be made without a risk of losing information (when want to assign a value of a larger data type to a smaller data type), the compiler requires us to use a **cast** to explicitly specify that you want to make the conversion

```
long longValue = 123;
int intValue = (int)longValue; // compiler error without cast
```

- For reference types, a cast is required when you want to convert from a base class to a derived class

```
Base b = new Derived();
Derived d = (Derived)b; // compiler error without cast
```

- Sometimes, the compiler cannot determine if a cast will be valid at runtime. In this case, it will compile the code, but the runtime will raise an exception (InvalidCastException) if the cast is invalid

```
object o = new object();
Derived d = (Derived)o; // InvalidCastException
```

USER DEFINED CONVERSIONS

- C# allows us to define conversions for our classes, so we can convert to and from other types

- These conversions are declared like operators, and their name is the name of the type we want to convert to. The argument of the operator is the name of the class we want to convert from.

- The operator must have the explicit or implicit keyword
  - If we use an explicit conversion, the compiler requires us to use the cast operator
  - If we use an implicit conversion, we can omit the cast operator

```
public static implicit operator byte(Digit d) => d.digit;
public static explicit operator Digit(byte b) => new Digit(b);
```

USER DEFINED CONVERSIONS

```
public class ComplexNumber
{
  public double Real { get; set; }
  public double Imaginary { get; set; }

  public static implicit operator ComplexNumber(double number)
  {
    ComplexNumber complex = new ComplexNumber();
    complex.Real = number;
    return complex;
  }
}

ComplexNumber cnExplicit = (ComplexNumber)1; // explicit conversion
ComplexNumber cnImplicit = 1; // implicit conversion
Console.WriteLine(cnExplicit);
Console.WriteLine(cnImplicit);
```

The *is* and *as* opeartors

- We can use the **is** operator to check if an object is an instance of a certain type

```
Base b = new Base();
Console.WriteLine(b is Base); // true
Console.WriteLine(b is Derived); // false
```

- We can use the **as** operator to convert a type to another type without raising an exception (the result will be null if the types are not compatible

```
Derived d = new Derived();
Base b = d as Base; // not null reference to derived
MoreDerived md = d as MoreDerived; // null
```

Conversions using helper classes

- The .NET Framework contain some classes that help us convert between incompatible types
- The System.Convert class contains methods that can convert between different framework types:
  https://docs.microsoft.com/en-us/dotnet/api/system.convert
- The System.BitConverter class contains methods that can convert to and from binary representations:
  https://docs.microsoft.com/en-us/dotnet/api/system.bitconverter

Value types vs Reference types, Stack vs Heap

- **Value types** (derived from System.ValueType, e.g. int, bool, char, enum, struct etc and pointers):
  - on **stack:** variable inside a method; method parameter
  - on **heap**: member of a class

In C# all types derived from System.Object (eg: System.ValueType)

- **Reference types** (classes, interfaces, delegates, object, string) are always allocated on the heap

Stack:

- keep track of what's executing in our code(what's been called).
- The stack is self-maintaining (basically takes care of its own memory management) method, a new Frame is added in the stack.
- Object allocated on the stack are available only inside of a stack frame (execution of a method), while object allocated on the heap can be accessed from anywhere.

Heap:

- responsible for keeping track of our objects.
- The heap has to warry about Garbage Collector (which deal with how to keep Heap clean)

Boxing and Unboxing

- **Boxing** is converting a value type to an instance of an object class
- **Unboxing** is extracting the value from the object it was boxed in
- **Boxing and unboxing** are used to treat value types and reference types in the same way

- Before C# supported generics, collections could only store objects. Because of this, boxing was used to convert value types to objects in order to add them to collections

- **Boxing** a value (implicit conversion): allocates an object on the heap, and copies the value inside the object
- **Unboxing** (explicit conversion): type **object** to a **value** type or from an interface type to a value type that implements the interface

```
int i = 123;
object o = i; // boxing
int j = (int)o; // unboxing
```

Implicitly typed local variables (var)

- When you declare and instantiate variables on the same line, sometimes code readability when the type of the variable is very long
- C# introduced the **var** keyword that can be used to make the code more readable
- The var type variable can be used to store a simple .NET data type, a complex type, an anonymous type, or a user-defined type.

```
Dictionary<int, string> dict = new Dictionary<int, string>();
var dict = new Dictionary<int, string>();

foreach(KeyValuePair<int, string> kvp in dict) {...}
foreach(var kvp in dict) {...}
```

Implicitly typed local variables (var)

- var can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null, or to a method group, or an anonymous function.
- var cannot be used on fields at class scope.
- Variables declared by using var cannot be used in the initialization expression.
- Multiple implicitly-typed variables cannot be initialized in the same statement. (var x, y, z;)
- If a type named var is in scope, then the var keyword will resolve to that type name and will not be treated as a part of an implicitly typed local variable declaration.

DYNAMIC TYPE

- It can be used to disable compile-time checking, and instead it resolves types at runtime (The compiler does not check the type of the dynamic type variable at compile time, instead of this, the compiler gets the type at the run time.)
- It is used to treat similarly objects that are different. It is also used when interoperability with dynamic languages, and MS Office interop
- When you declare an object, instead of using its type, you can use the “dynamic” keyword

```
class Cat
{
    public void MakeNoise()
    {
        Console.WriteLine("Meow!");
    }
}

dynamic c = new Cat();
c.MakeNoise(); // works
c.GiveFood(); // no compile error, but runtime error
```

INDEXING OPERATORS

- In C# you can overload many operators for a certain class, but you cannot overload the [] operator
- Instead, C# supports defining indexers for a class, so the classes can be indexed just like arrays
- An indexer is like a property (it has a get and set accessor), but it can also receive parameters
- You declare an indexer like a property, except that the property name is the reserved keyword “this”, and then in square brackets you can use parameters

  Return-type this[listOfParameters]

- Indexers can be declared on an interface
- Indexers can be overloaded

INDEXING OPERATORS

```
public Employee this[int index]
{
  get
  {
    return array[index];
  }
  set
  {
    array[index] = value;
  }
}
```

DELEGATES

- Sometimes we need to run a certain action, but we don’t know exactly what that action is
- Instead of coupling the action to the piece of code that needs to execute it, we can create a delegate for that action, so that another class can implement it
- A delegate is a reference type that is used to encapsulate a method with a certain signature (return type and parameters)
- Any method that matches that signature can be connected to a delegate of that type

DELEGATES

- A delegate is a type that represents a reference to methods with a particular signature and return type
- A delegate is declared like a method, but with the reserved word “delegate” before its return type:

```
public delegate double PerformOperation(double a, double b);
```

- To instantiate a delegate, you can assign a method to an object of that delegate’s type:

```
double PerformAddition(double x, double y){ return x+y; }
PerformOperation op = PerformAddition;
```

- To call the method pointed to by the delegate, we use the instantiated delegate as a method, providing its parameters and using its returned value:

```
double result = op(1,2);
```

DELEGATES

- The instantiated delegate is an object, and it can be passed as a parameter to a method. This is useful when you want to abstract away part of an operation.
- A delegate can also be instantiated with a method belonging to an instance of an object
- (e.g. “Print” method from the next example)

```
class ConsolePrinter
{
    private ConsoleColor color;
    public ConsolePrinter(ConsoleColor color)
    {
        this.color = color;
    }
    public void Print(string message)
    {
        Console.ForegroundColor = color;
        Console.WriteLine(message);
        Console.ResetColor();
    }
}

class ThirdExample
{
    public void Run()
    {
        ConsolePrinter successPrinter = new ConsolePrinter(ConsoleColor.Green);
        ConsolePrinter errorPrinter = new ConsolePrinter(ConsoleColor.Red);

        FileCopier copier = new FileCopier();
        FileCopier.Printer printerSuccess = successPrinter.Print;
        FileCopier.Printer printerError = errorPrinter.Print;

        copier.StartCopying(printerSuccess, printerError);
    }
}
```

MULTICAST DELEGATES

- Sometimes a delegate needs to call multiple methods. This is called multicasting in the .NET Framework.
- A delegate contains a list of methods with the same signature that can be called at once (invocation list). To add or remove methods to/from a delegate's list, we can use the + and – operator

```
double PerformAddition(double x, double y){ return x+y; }
double PerformSubtraction(double x, double y){ return x-y; }

PerformOperation opAdd = PerformAddition;
PerformOperation opSub = PerformSubtraction;
PerformOperation opCombined = opAdd + opSub;

double result = opCombined(1,2)
```

- It's not a good idea to use multicast delegates with non-void return types. The return value is the value returned by the last method in the method group. The execution order is unspecified.

EVENTS

- Implementation of the publisher-subscriber pattern
  - Publishers are objects that send messages (Contains the definition of the event and the delegate. The event-delegate association is also defined in this object)
  - Subscribers are objects that listen to messages and react accordingly (The delegate in the publisher class invokes the method (event handler) of the subscriber class)

- Are heavily used in Windows Forms and WPF apps

- For example: a button publishes an event when the user clicks on it. The way the event is handled is very specific to the a use case, and the button should not care about how the event is handled

- The button publishes an event, and another class can subscribe to the event and handle the implementation

EVENTS

- To declare an event, you use the “event” reserved keyword, then a type of delegate that describes the signature of the event, and then the name of the event

```
delegate void Printer(string message);
event Printer OnThresholdReached;
```

- To raise the event, you can call it like a method, passing the required parameters

- When an event has no subscribers, it is null. Calling a null event raises an exception. Before you call any event, you must check if it’s null or not.

```
if (ThresholdReached != null)
{
  ThresholdReached("A threshold has been reached");
}
```

EVENTS

- To subscribe to an event, we can use the += operator and a method that matches the delegate type
- To unsubscribe from an event, we use the -= operator and the method that we want to unsubscribe

```
private void Adder_SumModified(object sender, int e)
{
  Console.WriteLine($"The sum is now {e}");
}

private void Adder_ThresholdReached(object sender, EventArgs e)
{
  Console.WriteLine("Threshold reached");
}

NumberAdderThresholdEventHandler adder = new NumberAdderThresholdEventHandler(10);
adder.ThresholdReached += Adder_ThresholdReached;
adder.SumModified += Adder_SumModified;
```

EVENTS

- To make it easier to write events, the .NET framework provides some ready-made delegates that we can used, so we don’t have to write our own

- EventHandler
  - Void return type
  - Two parameters: the sender, and an EventArgs object

- EventHandler<T>
  - Void return type
  - Two parameters: the sender, and a T object

```
public event EventHandler ThresholdReached;
public event EventHandler<int> SumModified;
```

EVENTS

- In order to handle events in C#, we use delegates
- The class that publishes the event usually declares a delegate, and the classes that subscribe to the event use that delegate (or EventHandler)
- Methods that handle events are called event handlers, and by convention return void and take two parameters: the sender and the event arguments
- Several methods can subscribe to the same event
- The same method can handle several events (this is where the sender parameter is useful)

SOLID

- **Single responsibility principle**
  A class should have a single responsibility

- **Open/Closed principle**
  Open for extension, closed for modification

- **Liskov substitution principle**
  Derived classes must be substitutable for their base classes

- **Interface segregation principle**
  Many client-specific interfaces are better than one general-purpose interface

- **Dependency inversion principle**
  Depend on abstractions, not on concretions

Single responsibility principle

- A class should have a single responsibility
- A class should have only one reason to change
- If a class does more than one thing, different teams may need to make modifications to the same class at the same time, leading to conflicts
- Example:
  Don’t mix presentation and business logic
  Don’t mix serialization and business logic
  Use overloading to handle different types of objects

Open/Closed principle

- Software entities should be open for extension, but closed for modification (you should be able to extend a class's behavior, without modifying it)

- Organize our code in a way that if a new functionality is needed, we don't change any existing code, but write new code that is used by the existing code

- Use inheritance to decouple different types of objects that you need to work with

Liskov substitution principle

- If any module is using a Base class then the reference to that Base class can be replaced with a Derived class without affecting the functionality of the module.

- How to tell if this principle is broken:
  A method in the subclass is not implemented
  A method in a subclass overrides the method in the base class and gives it a new meaning

Interface segregation principle

- Clients should not be forced to depend on methods they do not use
- This helps to keep systems decoupled and easier to modify
- Many client-specific interfaces are better than one general-purpose interface

Dependency inversion principle

- Entities must depend on abstractions, not on concretions
- High-level modules should not depend on low-level modules. Both should depend on abstractions
- Use interfaces to decouple components
- The control is inverted - it calls me rather me calling the framework. This phenomenon is Inversion of Control (also known as the Hollywood Principle - "Don't call us, we'll call you").

Object oriented programming principles

- **Encapsulation**
  Bind data and behavior in a single unit and restrict access to some components

- **Abstraction**
  Expose the relevant parts, and hide the unnecessary details

- **Inheritance**
  Inherit members from a parent class

- **Polymorphism**
  Different objects perform the same action in different ways

OOP - Encapsulation

- Encapsulation is implemented by using **access specifiers**. An **access specifier** defines the scope and visibility of a class member.

- C# supports the following access specifiers:
  Public (accessed from outside the class)
  Private (accessed only from the class)
  Protected (a child class can access protected members from the base class)
  Internal (accessed from the current assembly – within the application/namespace)
  Protected internal (allows a class to hide its member variables and member functions from other class objects and functions, except a child class within the same application)

OOP - Inheritance

- Inheritance is a concept in which you define parent classes and child classes.
- The child classes inherit methods and properties of the parent class, but at the same time, they can also modify the behavior of the methods if required.
- The child class can also define methods of its own if required.

OOP - Polymorphism

- Polymorphism is a OOPs concept where one name can have many forms.

Types:

- **Compile time polymorphism**
  defining a multiple methods with same name but with different parameters. By using compile time polymorphism, we can perform a different tasks with same method name by passing different parameters.

- In C#, the compile time polymorphism can be achieved by using **method overloading** and it is also called as **early binding** or **static binding**.

```
public void AddNumbers(int a, int b)
{
    Console.WriteLine("a + b = {0}", a + b);
}
public void AddNumbers(int a, int b, int c)
{
    Console.WriteLine("a + b + c = {0}", a + b + c);
}
```

OOP - Polymorphism

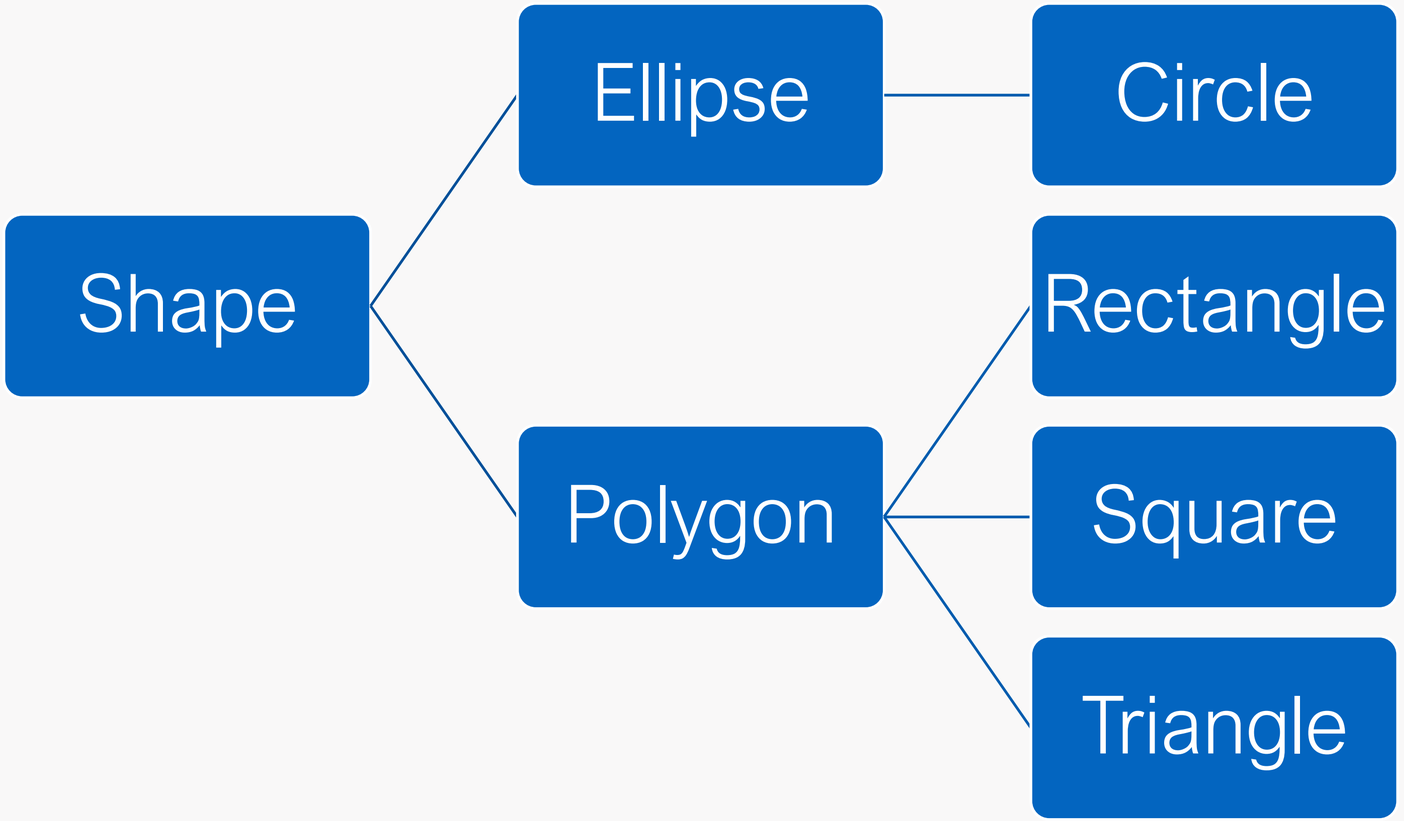
Types:

- **Run time polymorphism** means overriding a base class method in derived class by creating a similar function and this can be achieved by using override & virtual keywords along with inheritance principle

- In C#, the run time polymorphism can be achieved by using **method overriding** and it is also called as **late binding** or **dynamic binding**.

```
// Base Class
public class Users
{
    public virtual void GetInfo()
    {
        Console.WriteLine("Base Class");
    }
}
// Derived Class
public class Details : Users
{
    public override void GetInfo()
    {
        Console.WriteLine("Derived Class");
    }
}
```

OOP Example

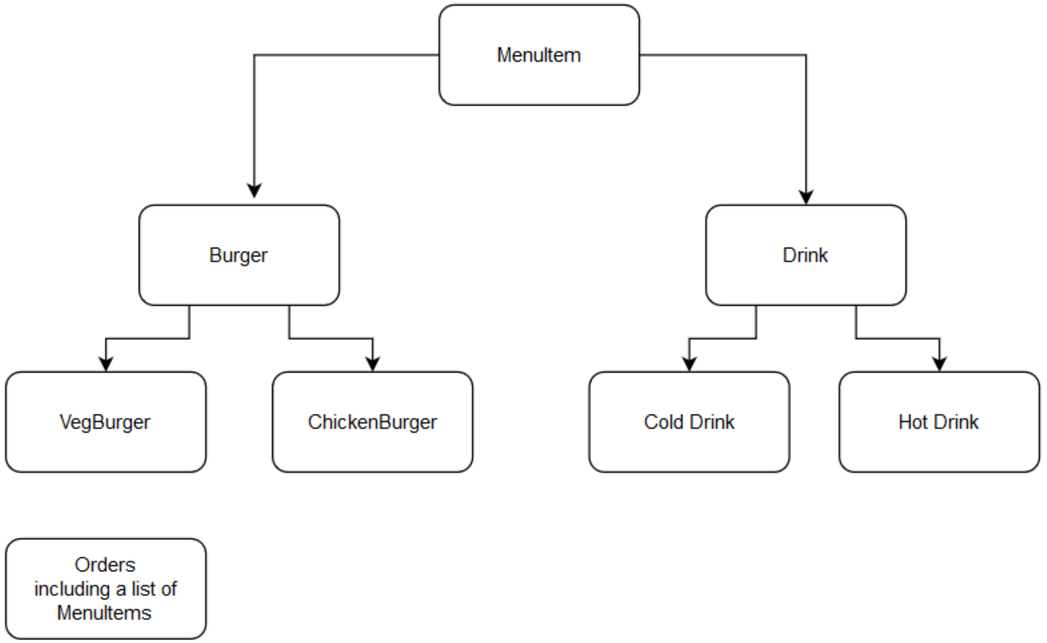


Homework

Create a project and apply solid/oop principle (overload, override, private, public, virtual/abstract etc).

Example: A restaurant where you can have a list of Orders

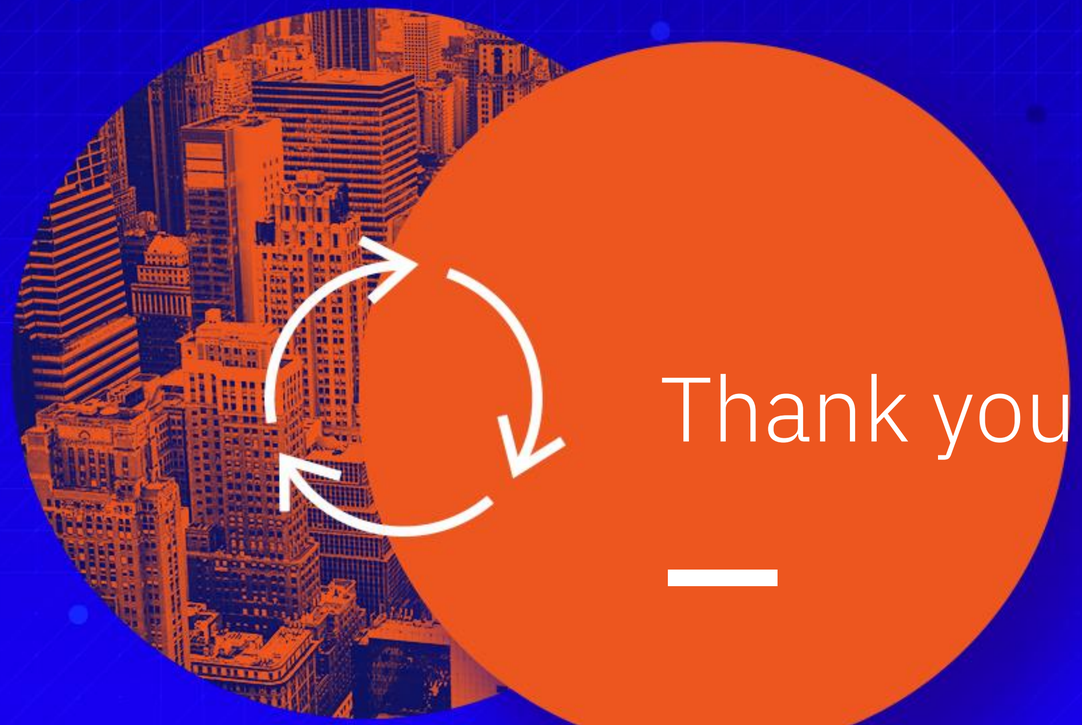
- We can have a MenuItem with some properties
- We can have other concrete classes that will extend MenuItem
- For Restaurant we can create a interface/abstract class that will contain a list of methods to add/edit/remove from a list (the MenuItems)



Reference

- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/using-type-dynamic
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/
- https://en.wikipedia.org/wiki/SOLID
- https://www.intermittentbug.com/article/articlepage/the-4-primary-object-oriented-principles-explained-in-simple-terms/2299

Pentalog



Thank you!

DATA ANALYSIS

UX/UI

FRONT-END

TECHNOLOGY

NEAR/OFFSHORE

AGILITY

BACK-END

SPRINT

KANBAN

CAMPAIGNS

GROWTH HACKING

SCRUM

BACKLOG

DEVOPS

DESIGN

SEO

CONTINUOUS INTEGRATION

MOBILE

QA

AUTOMATION

RESPONSIVE

UNIT TESTING