C#

Slides Roadmap

- Types in C#
- Object Oriented Programming with C#

Types in C#

- C# is a strongly-typed language
- Predefined Data types
  - Char
  - String
  - Int32
  - Int64
  - Boolean
  - Object
  - Etc.

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

Types in C#

- Value Types
  - **Struct**
  - **Enum**

```
public struct Coordinate
{
    public int x, y;

    public Coordinate(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

```
public enum HttpMethod
{
    GET,
    POST,
    DELETE,
    PUT
}
```

Types in C#

- **References Types**
  - Class
  - Delegate
  - Array
  - Interface
- References types are supported by the **garbage collector**
- **Null** is the default value of a reference type

Conventions

- choose easily readable identifier names
- favor readability over brevity
- use PascalCasing for all public member, type, and namespace names consisting of multiple words
- use camelCasing for parameter names.

Namespaces

- namespace keyword
- declare a scope that contains a set of related objects

```
using Microsoft.Owin.Hosting;
using System;
using System.Threading;

namespace MefPlugins
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
```

Accessibility

- All types and type members have an accessibility level
  - **Public** (can be accessed by any other code)
  - **Protected** (can be accessed only by code in the same class or derived class)
  - **Internal** (can be accessed by code in the same assembly)
  - **Private** (can be accessed only by code in the same class)

Strings and Console

- Console
  - Console.WriteLine(…)
  - Console.ReadLine() => return string

- Construct strings

```
string s = String.Format("The temperature is {0}°C.", temp);

Console.WriteLine($"Name = {name}, hours = {hours:hh}")
```

Out keyword

- Out variables
- Ex:

```
var str = "10";
if (int.TryParse(str, out var i))
{
    Console.WriteLine($"int: {i}");
}
```

- Ignore out parameter:

```
var str = "10";
if (int.TryParse(str, out _))
{
    Console.WriteLine($"str is an int");
}
```

Pattern matching

- Pattern matching
  - Constant pattern
  - Type pattern
  - Var pattern
- You can switch on any type
- Patterns can be used in clauses
- Clauses can have additional conditions
- The first one that matches gets picked
- default clause is always evaluated last

Pattern matching

- Is expression

```
object o = 10;
if (o is int i || (o is string s && int.TryParse(s, out i)))
{
}
```

Pattern matching

```
switch (shape)
{
    case Circle c:
        Console.WriteLine($"circle with radius {c.Radius}");
        break;
    case Rectangle r when r.Width == r.Height:
        Console.WriteLine($"{r.Width} x {r.Height} square");
        break;
    case Rectangle r:
        Console.WriteLine($"{r.Width} x {r.Height} rectangle");
        break;
    default:
        Console.WriteLine("<unknown shape>");
        break;
    case null:
        Console.WriteLine("shape is null");
        break;
}
```

Tuple

- Tuple (nuget package System.ValueTuple)
- Tuples are value types with public and mutable fields

```
var movie = GetMovie();
Console.WriteLine($"MOVIE: {movie.Item1} - {movie.Item2}");

(int, string, int) GetMovie()
{
    return (1, "avatar", 2009);
}
```

Tuple

- Better description

```
var movie = GetMovie();
Console.WriteLine($"MOVIE: {movie.id} - {movie.title}");
var movie2 = (1, "avatar", 2009);
if (movie.Equals(movie2))
{
    Console.WriteLine("value type equality!");
}

(int id, string title, int year) GetMovie()
{
    return (1, "avatar", 2009);
}
```

Tuple

- Deconstruction
  - Tuple

```
var (id, title, year) = GetMovie();
Console.WriteLine($"MOVIE: {id} - {title}");
```

  - Other types

```
public class Rectangle : Shape
{
    6 references
    public int Height { get; set; }
    6 references
    public int Width { get; set; }

    0 references
    public void Deconstruct(out int height, out int width) { height = Height; width = Width; }
}
```

Local function

- Local function

```
private static void TupleDemo()
{
    var (id, title, year) = GetMovie();
    Console.WriteLine($"MOVIE: {id} - {title}");

    (int id, string title, int year) GetMovie()
    {
        return (1, "avatar", 2009);
    }
}
```

Null Conditional Operator

```
var result = new Person[]
{
    new Person("user1"),
    new Person("user2"),
    null
};
foreach (var item in result)
{
    Console.WriteLine(item?.Name ?? "<no name>");
}
```

Nameof keyword

- invalid reference causes a compiler error
- expressions will be correctly updated when you refactor code

```
0 references
public override string ToString() =>  $"{nameof(Name)}: {Name}";
```

Top Level statements

- Remove boilerplate from Program.cs (Main method)
- Only 1 file may use top level statement
- Script like experiennce
- Ideal for Azure functions
- May contain async expression
- Can access to an array of strings named args

Object Oriented Programming with C#

- **Class**: Definition of a group of entities that share the **attributes** (state of object) and **methods**, representing their behavior

- **Object**: Instance of a class
- Declaration and instantiation:

```
Person p1 = new Person();
Person p2 = p1;
```

Object Oriented Programming with C#

- Keyword **this**:
  - Reference to the current object
  - Can only be used in non static methods

```
//attribute
private string name;

//constructor
public Person(string name)
{
    this.name = name;
}
```

Object Oriented Programming with C#

- Example

```
class Person
{
    //attribute
    private string name;

    //constructor
    public Person(string name)
    {
        this.name = name;
    }

    //method
    public void Hello()
    {
        Console.WriteLine("Hello world");
    }

    //method
    public void DisplayName()
    {
        Console.WriteLine(name);
    }
}
```

Object Oriented Programming with C#

- Properties are used to
  - **Get** the value of a field
  - **Set** the value of a field

```
private string name;

public string Name
{
    get { return name; }
    set { name = value; }
}
```

prop snippet

```
public string Name { get; set; }
```

Object Oriented Programming with C#

- **Object initializers**

```
Person p = new Person { FirstName = "Jérémy", LastName = "PEKMEZ" };
```

- **var** infer the type of the variable

```
var i = 10; // implicitly typed
int i2 = 10; //explicitly typed
```

- **Anonymous Types**

```
var obj = new { Message = "Hello world", Count = 10 };
```

Object Oriented Programming with C#

- **Indexers** enable objects to be **indexed** like arrays

```
class ListPerson
{
    private Person[] _collection = new Person[100];

    public Person this[int index]
    {
        get
        {
            return _collection[index] as Person;
        }
        set
        {
            _collection[index] = value;
        }
    }
}
```

Object Oriented Programming with C#

- Class support **inheritance**
- Class that **derive** from another class automatically **contains** all the **public**, **protected** and **internal members**

```
public class MyClass
{
    public MyClass()
    { }
}
public class MyDerivedClass : MyClass
{
    public MyDerivedClass()
        : base()
    { }
}
```

Object Oriented Programming with C#

- **Virtual** and **Override**
  - Only for methods
  - Non static methods
  - Non private methods
- **New**
  - Redefine a non-virtual methods
  - Can be used with virtual methods

Object Oriented Programming with C#

- **Example**

```
public class MyClass
{
    protected int _count;

    public void Decrement()
    {
        _count--;
    }

    public virtual void Increment()
    {
        _count++;
    }
}
```

```
public class MyDerivedClass : MyClass
{
    public override void Increment()
    {
        _count += 2;
    }

    public new void Decrement()
    {
        _count -= 2;
    }
}
```

Object Oriented Programming with C#

- An override method can return a type derived from the return type of the overridden base method (C# 9)

```
class Entity
{
    1 reference
    public int Id { get; set; }

    2 references
    protected Entity(int id)
    {
        Id = id;
    }

    0 references
    public virtual Entity Create(int id) => new Entity(id);
}

2 references
class Product : Entity
{
    1 reference
    public Product(int id)
        : base(id)
    { }

    0 references
    public override Product Create(int id) => new Product(id);
}
```

Object Oriented Programming with C#

- **Demo**

Object Oriented Programming with C#

- **Interface**
  - Class and structs can inherit multiple interfaces
  - Means that the type implements all the methods defined in the interface
  - No code
  - Not access modifier

```
public interface IMyClass
{
    void Decrement();
}

public class MyClass : IMyClass
{
    protected int _count;

    public void Decrement()
    {
        _count--;
    }
}
```

Object Oriented Programming with C#

- **Abstract** classes
  - Have to be **subclassed**
  - Cannot be instantiated
  - Can contain abstract methods (subclasses have to provide an implementation)

```
public abstract class Vehicule
{
    public abstract void Accelerate();
}
```

Object Oriented Programming with C#

- **Sealed** classes
  - Cannot be derived

```
public sealed class Renault : Vehicule
{
    public override void Accelerate()
    { }
}
```

- All classes inherits from System.Object
  - ToString()
  - Equals()
  - Etc.

Object Oriented Programming with C#

- Static class and members
  - Access to the members of a static class by using the class name

```
public static class StaticClass
{
    public static void MethodStatic()
    {
        Console.WriteLine("I'm a static method");
    }
}
```

```
StaticClass.MethodStatic();
```

Object Oriented Programming with C#

- Partial class and methods
  - Split the definition of a class, struct, interface or method over two or more files

```
public partial class Employee
{
    public void DoWork()
    { }
}

public partial class Employee
{
    public void GoToLunch()
    { }
}
```

Object Oriented Programming with C#

- **typeof**
  - Get the System.Type of the object
- **is**
  - Check the type of an object
- **as**
  - Check the type of an object and cast it

Object Oriented Programming with C#

- **Operator overloading**

```
public struct Complex
{
    public int real;
    public int imaginary;

    public Complex(int r, int i)
    {
        real = r;
        imaginary = i;
    }

    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
    }
    public static Complex operator ++ (Complex c)
    {
        return new Complex(++c.real, ++c.imaginary);
    }
}
```

Object Oriented Programming with C#

- Delegate
  - Reference method
  - Can be used like any other method
  - Any method that matches the delegates signature can be assigned

```
public delegate int PerformCalculation(int x, int y);
```

Object Oriented Programming with C#

- **Example**

```
public delegate int PerformCalculation(int x, int y);

class Program
{
    public static int Addition(int x, int y)
    {
        return x + y;
    }

    static void Main(string[] args)
    {
        PerformCalculation myDelegate = new PerformCalculation(Addition);
        int result = myDelegate(5, 3);
        Console.WriteLine(result);
    }
}
```

Object Oriented Programming with C#

- **Anonymous methods**
  - block code in delegate parameter

```
PerformCalculation myDelegate = new PerformCalculation(Addition);
PerformCalculation myDelegate2 = delegate(int x, int y)
{
    return x + y;
};
```

Object Oriented Programming with C#

- **Lambda expression**
  - Operator **=>**

```
PerformCalculation myDelegate = new PerformCalculation(Addition);
PerformCalculation myDelegate2 = delegate(int x, int y)
{
    return x + y;
};
PerformCalculation myDelegate3 = (x, y) => x + y;
```

Object Oriented Programming with C#

- **Demo delegate**

Object Oriented Programming with C#

- **Events**
  - Capture an action of the program
  - Based on delegates

```
public delegate void NewEventHandler(object sender, EventArgs e);

public event NewEventHandler MyEventHandler;
```

Object Oriented Programming with C#

- **Subscribe**

```
MyEventHandler += new NewEventHandler(MyClass_MyEventHandler);

void MyClass_MyEventHandler(object sender, EventArgs e)
{
}
```

- **Unsubscribe**

```
MyEventHandler -= new NewEventHandler(MyClass_MyEventHandler);
```

Object Oriented Programming with C#

- **Demo events**

Object Oriented Programming with C#

- **Generics**
  - Used to design classes and methods that defer the specification of one or more types
  - Code reused
  - Type safety

```
public class GenericList<T>
{
    public void Add(T input)
    { }
}
```

Object Oriented Programming with C#

- **Create generic list**

```
GenericList<string> list1 = new GenericList<string>();
GenericList<int> list2 = new GenericList<int>();

list1.Add("toto");
list2.Add("toto"); //error
```

- **Generic method**

```
public void Display<T>(GenericList<T> list, int index)
{
    Console.WriteLine(list[index].ToString());
}
```

Object Oriented Programming with C#

- **Constraints with where keyword**
  - T : struct
    - T is a value type
  - T : class
    - T is a reference type
  - T : new()
    - Public parameterless constructor
  - T : Person
    - T is or inherit from Person
  - T : IPerson
    - T implement interface IPerson

Object Oriented Programming with C#

- **Example**

```
public class GenericList<T> where T : Employee
{
    public void Add(T input)
    { }
}
```

```
public class GenericList<T> where T : Employee, IEnumerable, new()
{
    public void Add(T input)
    { }
}
```

```
public class SuperGeneric<T, U>
    where T : Employee
    where U : new()
{ }
```

Object Oriented Programming with C#

- **Demo generics**

Object Oriented Programming with C#

- **Iterators**
  - used to support **foreach** iteration
  - GetEnumerator return an ordered sequence of same type values
  - **yield return** is used to return each element
  - It is possible to use generics with iterators

Object Oriented Programming with C#

- **Example**

```
public class DaysOfTheWeek : IEnumerable
{
    string[] m_Days = { "Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat" };

    public System.Collections.IEnumerator GetEnumerator()
    {
        for (int i = 0; i < m_Days.Length; i++)
        {
            yield return m_Days[i];
        }
    }
}
```

```
DaysOfTheWeek week = new DaysOfTheWeek();

foreach (string day in week)
{
    System.Console.Write(day + " ");
}
```

Object Oriented Programming with C#

- **Demo iterators**

Object Oriented Programming with C#

- **Nullable types (?)**

```
int? i = null;

if (i.HasValue == true)
{
    System.Console.WriteLine("num = " + i.Value);
}
else
{
    System.Console.WriteLine("num = Null");
}
```

Object Oriented Programming with C#

- **Named parameters**

```
public static int Addition(int x, int y)
{
    return x + y;
}
```

```
Addition(y: 10, x: 5);
```

Object Oriented Programming with C#

- **Optional parameters**

```
public static int Addition(int x, int y = 10)
{
    return x + y;
}

Addition(5);
```

Object Oriented Programming with C#

- **dynamic**
  - at compile time, an dynamic element support any operation
  - if the code is not valid, errors are caught at run time

```
Person p = new Person();
dynamic p2 = p;
p2.FirstName = "Toto";
p2.Error = "error"; //error at run time
```

Init accessor

- set accessor which can only be called during object initialization

```
public class Person
{
    public string? FirstName { get; init; }
    public string? LastName { get; init; }
}
```

```
var person = new Person { FirstName = "Mads", LastName = "Nielsen" }; // OK
person.LastName = "Torgersen"; // ERROR!
```

Record type

- A record is still a class
- Records are much closer to structs, but records are still reference types.
- Value-based equality
- Records can be mutable but they are primarily built for better supporting immutable data models
- Records can inherit from other records

Record type

- non-destructive mutation

```
var person = new Person { FirstName = "Mads", LastName = "Nielsen" };
var otherPerson = person with { LastName = "Torgersen" };
```

Record type

```
public record Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
    public Person(string firstName, string lastName)
      => (FirstName, LastName) = (firstName, lastName);
    public void Deconstruct(out string firstName, out string lastName)
      => (firstName, lastName) = (FirstName, LastName);
}
```

=

```
public record Person(string FirstName, string LastName);
```

Construction / Deconstruction

```
var person = new Person("Mads", "Torgersen"); // positional construction
var (f, l) = person;                          // positional deconstruction
```

Object Oriented Programming with C#

- **Exercices**