Faculty of Electronics and Information Technology

WARSAW UNIVERSITY OF TECHNOLOGY

Graphical User Interfaces (EGUI)

CSharp

Julian Myrcha

Institute of Computer Science

October 6, 2024

Installation (Ubuntu 20.04)

First we need register microsoft repository to be able getting updates:

```
wget https://packages.microsoft.com/config/ubuntu/22.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

Installation (Ubuntu 20.04)

Then we need to set priority for microsoft packages:

```
sudo nano /etc/apt/preferences.d/99microsoft-dotnet
```

```
Package: *
Pin: origin "packages.microsoft.com"
Pin-Priority: 1001
```

Installation (Ubuntu 20.04)

Then we need to install packages:

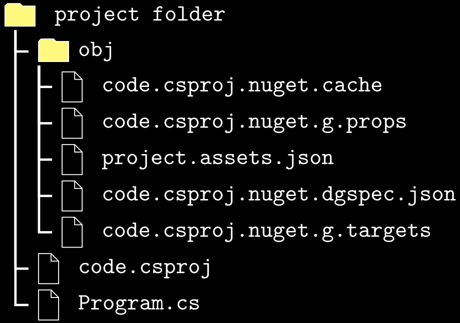
```
sudo apt-get update
sudo apt-get install apt-transport-https
sudo apt-get update
sudo apt-get install dotnet-sdk-6.0
#sudo apt-get install aspnetcore-runtime-6.0
#sudo apt-get install dotnet-runtime-6.0
```

First Console application

Create a console application in current folder:

```
dotnet new console
```

produces following folder structure:



- project folder
  - obj
    - code.csproj.nuget.cache
    - code.csproj.nuget.g.props
    - project.assets.json
    - code.csproj.nuget.dgspec.json
    - code.csproj.nuget.g.targets
  - code.csproj
  - Program.cs

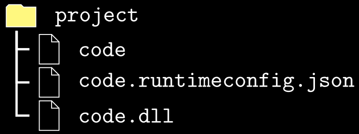
Run First Console application

build and run

```
dotnet build
dotnet run
# bin/Debug/netcoreapp3.0/code
```

to run .net core program one should have following files:

code - executable file
code.dll - library file
code.runtimeconfig.json - configuration file


project
code
code.runtimeconfig.json
code.dll

Hello World

- **Anders Hejlsberg** - creator of Delphi

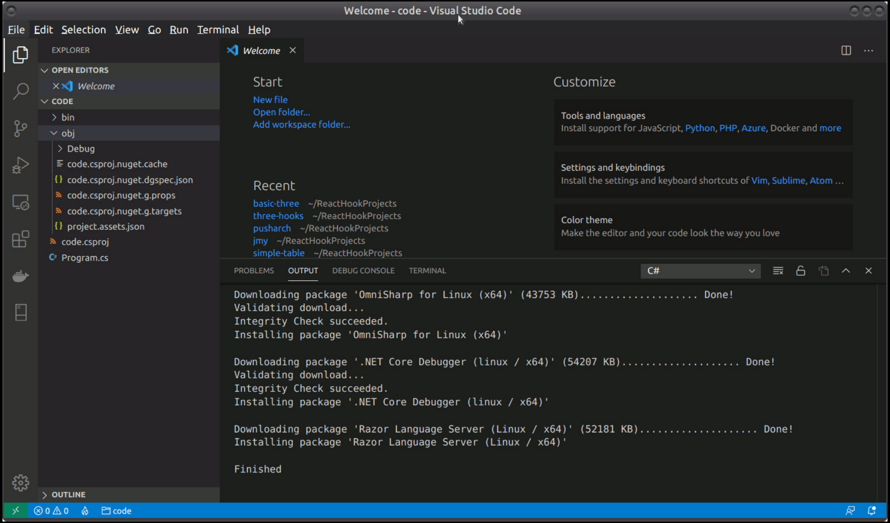
```
using System;    // we can use classes from System namespace

// this was a folder where I have run 'dotnet new console'
namespace code {  /* our symbols will be in their own namespace
                                            to avoid conflicts */

    class Program  {
        static void Main(string[] args)  {
            /* Console is a class in System Namespace */
            // WriteLine is a static method of the Console class
            Console.WriteLine("Hello World!");
            Console.ReadLine();   // read a line of text from the console
        }
    }
}
```

Visual Studio Code

- installation instructions to be found: `https://code.visualstudio.com/docs/setup/linux`

```
sudo apt install ./<file>.deb               # register repository
sudo apt-get install apt-transport-https
sudo apt-get update
sudo apt-get install code # or code-insiders
```

Visual Studio Code


Welcome - code - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
Welcome
CODE
bin
obj
Debug
code.csproj.nuget.cache
code.csproj.nuget.dgspec.json
code.csproj.nuget.g.props
code.csproj.nuget.g.targets
project.assets.json
code.csproj
Program.cs
OUTLINE
Welcome
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
C#
0 0
code

Start

New file
Open folder...
Add workspace folder...

Recent

basic-three ~/ReactHookProjects
three-hooks ~/ReactHookProjects
pusharch ~/ReactHookProjects
jmy ~/ReactHookProjects
simple-table ~/ReactHookProjects

Customize

Tools and languages
Install support for JavaScript, Python, PHP, Azure, Docker and more

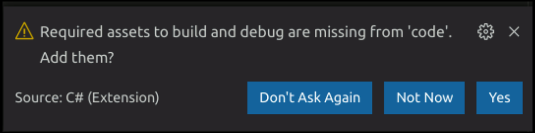
Settings and keybindings
Install the settings and keyboard shortcuts of Vim, Sublime, Atom ...

Color theme
Make the editor and your code look the way you love

```
Downloading package 'OmniSharp for Linux (x64)' (43753 KB).................... Done!
Validating download...
Integrity Check succeeded.
Installing package 'OmniSharp for Linux (x64)'

Downloading package '.NET Core Debugger (linux / x64)' (54207 KB).................... Done!
Validating download...
Integrity Check succeeded.
Installing package '.NET Core Debugger (linux / x64)'

Downloading package 'Razor Language Server (Linux / x64)' (52181 KB).................... Done!
Installing package 'Razor Language Server (Linux / x64)'

Finished
```

Visual studio Code

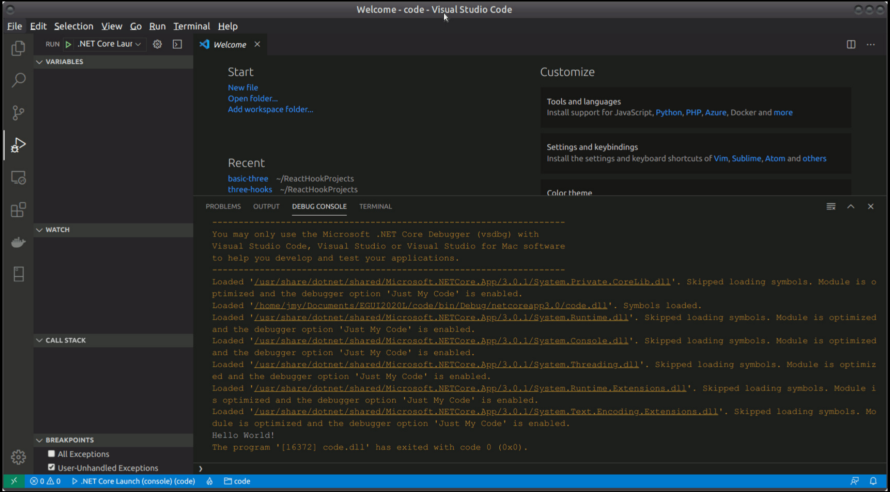
- Visual Studio Code is a lightweight source code editor
- Is available for Windows, macOS and Linux
- It comes with built-in support for JavaScript, TypeScript and Node.js
- Rich ecosystem of extensions for other languages (such as C++, C#, Java, Python, PHP, SQL)
- Runtimes (such as .NET, Unity, Python, Java).



⚠ Required assets to build and debug are missing from 'code'. Add them?

Source: C# (Extension)

Visual Studio Code


Welcome - code - Visual Studio Code
File Edit Selection View Go Run Terminal Help
RUN ▷ .NET Core Laur
Welcome ×
VARIABLES
WATCH
CALL STACK
BREAKPOINTS
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
⊗ 0 ⚠ 0
▷ .NET Core Launch (console) (code)
code

□ All Exceptions
☑ User-Unhandled Exceptions

Start
New file
Open folder...
Add workspace folder...

Recent
basic-three ~/ReactHookProjects
three-hooks ~/ReactHookProjects

Customize

Tools and languages
Install support for JavaScript, Python, PHP, Azure, Docker and more

Settings and keybindings
Install the settings and keyboard shortcuts of Vim, Sublime, Atom and others

Color theme

```
-------------------------------------------------------------------
You may only use the Microsoft .NET Core Debugger (vsdbg) with
Visual Studio Code, Visual Studio or Visual Studio for Mac software
to help you develop and test your applications.
-------------------------------------------------------------------
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/3.0.1/System.Private.CoreLib.dll'. Skipped loading symbols. Module is o
ptimized and the debugger option 'Just My Code' is enabled.
Loaded '/home/jmy/Documents/EGUI2020L/code/bin/Debug/netcoreapp3.0/code.dll'. Symbols loaded.
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/3.0.1/System.Runtime.dll'. Skipped loading symbols. Module is optimized
and the debugger option 'Just My Code' is enabled.
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/3.0.1/System.Console.dll'. Skipped loading symbols. Module is optimized
and the debugger option 'Just My Code' is enabled.
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/3.0.1/System.Threading.dll'. Skipped loading symbols. Module is optimiz
ed and the debugger option 'Just My Code' is enabled.
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/3.0.1/System.Runtime.Extensions.dll'. Skipped loading symbols. Module i
s optimized and the debugger option 'Just My Code' is enabled.
Loaded '/usr/share/dotnet/shared/Microsoft.NETCore.App/3.0.1/System.Text.Encoding.Extensions.dll'. Skipped loading symbols. Mo
dule is optimized and the debugger option 'Just My Code' is enabled.
Hello World!
The program '[16372] code.dll' has exited with code 0 (0x0).
```

formating: omnisharp.json file

- to start auto-formating code one should:
    - In project folder create `omnisharp.json` file

formating: omnisharp.json file

- to start auto-formating code one should:
  - In project folder create `omnisharp.json` file

```
{
    "FormattingOptions": {
        "newLine": "\n",
        "useTabs": false,
        "tabSize": 2,
        "indentationSize": 2,
        "NewLinesForBracesInLambdaExpressionBody": false,
        "NewLinesForBracesInAnonymousMethods": false,
        "NewLinesForBracesInAnonymousTypes": false,
        "NewLinesForBracesInControlBlocks": false,
        "NewLinesForBracesInTypes": false,
        "NewLinesForBracesInMethods": false,
        "NewLinesForBracesInProperties": false,
        "NewLinesForBracesInObjectCollectionArrayInitializers": false,
        "NewLinesForBracesInAccessors": false,
        "NewLineForElse": false,
        "NewLineForCatch": false,
        "NewLineForFinally": false,
        "NewLineForMembersInObjectInit": false,
        "NewLineForMembersInAnonymousTypes": false,
        "NewLineForClausesInQuery": false
    }
}
```

formating: omnisharp.json file

- to start auto-formating code one should:
  - In project folder create `omnisharp.json` file
  - restart VS code

formating: omnisharp.json file

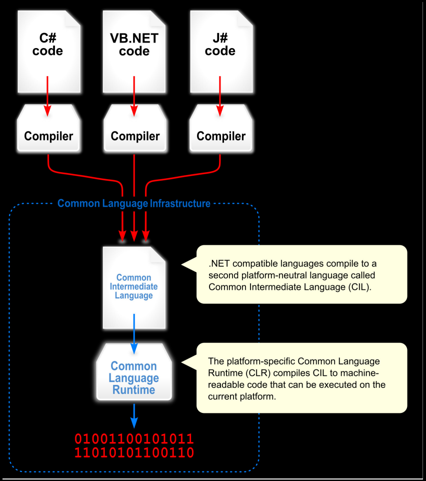
- to start auto-formating code one should:
  - In project folder create `omnisharp.json` file
  - restart VS code
  - format current source file using `<ctrl>+<shift>+i`

CLI = Common Language Infrastructure

CLI -Common Language Infrastructure. Provides a language-neutral platform for application development and execution

CLR -Microsoft's implementation of CLI

CIL -Common Intermediate Language



.NET compatible languages compile to a second platform-neutral language called Common Intermediate Language (CIL).

The platform-specific Common Language Runtime (CLR) compiles CIL to machine-readable code that can be executed on the current platform.

Simple types

name	size	example
bool		true, false
char		'a'
decimal	128	1E–28 to 7.9E+28 (28 significant places), 11.95m
double	64	5E–324 to 1.7E+308., 100.1D lub 100.1
float	32	1.5E–45 to 3.4E+38, 100.1F
sbyte	8	–128 to 127
short	16	–32,768 to 32,767
int	32	–2,147,483,648 to 2,147,483,647
long	64	–9,223,372,036,854,775,808L - 9,223,372,036,854,775,807L
byte	8	0 to 255
ushort	16	0 to 65,535
uint	32	0 to 4,294,967,295
ulong	64	0 to 18,446,744,073,709,551,615

types

name	size	example
string	2 bytes/character	'Hello world'

Numbers

- Integer types
- Floating point types

Type Casting

- **Implicit Casting** (automatically) - converting a smaller type to a larger type size

```
//char -> int -> long -> float -> double
double num = 20.6F;
```

- **Explicit Casting** (manually) - converting a larger type to a smaller size type

```
//double -> float -> long -> int -> char
float num = (float)20.6;
```

Type Conversion Methods

Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long)

strings

strings are objects

```
string txt = "Hello World";
//  implicit casting of number to string because '+' concatenate strings
Console.WriteLine("The length of the txt string is: " + txt.Length);
```

string usefull methods:

ToUpper -return string in uppercase
ToLower -return string in uppercase
IndexOf -
Substring -returns part of the string

String interpolation (C# 6):

```
string firstName = "John";
string lastName = "Doe";
"My full name is: {firstName} {lastName}";
Console.WriteLine(name);
```

arrays (1)

arrays are objects

```
string[] cities;
string[] cities = new string[2];
string[] cities = new string[2]{"Warsaw", "Krakow"};
string[] cities = {"Warsaw", "Krakow"};
cities = new string[3]{"Warsaw", "Krakow", "Bialystok"};
Console.WriteLine(cities[1]);     // Outputs Krakow
cities[1]="Gdansk";
Console.WriteLine(cities.Length); // Outputs 2
```

Loop Through an Array

```
string[] cities = {"Warsaw", "Gdansk", "Lublin", "Wroclaw"};
for (int i = 0; i < cities.Length; i++)
    Console.WriteLine(cities[i]);
foreach (string city in cities)
    Console.WriteLine(city);
```

arrays (2)

Sort Arrays

```
string[] cities = {"Warsaw", "Gdansk", "Lublin", "Wroclaw"};
Array.Sort(cities);
foreach (string city in cities)
    Console.WriteLine(city);
```

classes (1)

- every class derive (directly or indirectly) from `Object` base class:
- classes are always a reference, should be allocated on the heap by `new`

classes (2)

Object class methods:

method	description
Equals(Object)	Determines whether the specified object is equal to the current object.
Equals(Object, Object)	Determines whether the specified object instances are considered equal.
Finalize()	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection
GetHashCode()	Serves as the default hash function
GetType()	Gets the Type of the current instance
MemberwiseClone()	Creates a shallow copy of the current Object.
ReferenceEquals(Object, Object)	Determines whether the specified Object instances are the same instance.
ToString()	Returns a string that represents the current object.

Access Modifiers

Czy chodziło Ci o: should be specified for all class elements (defaults to rivate)
69/5000
should be specified for all class elements (defaults to `private`)

modifier	description
public	The code is accessible for all classes
private	The code is only accessible within the same class
protected	The code is accessible within the same class, or in a class that is inherited from that class.
internal	The code is only accessible within its own assembly, but not from another assembly.

- Main is a static method of one of the class
- class can be public or (it is default) internal

```
public class Program {
  private string city;
  static void Main(string[] args) {
    program app = new Program("Warsaw");
    Console.WriteLine(app.city);
  }
}
```

classes static

static class - compiler do not allow to create an instance of the class
static field - use class name instead of the object

```
public class Point {
    private static int _counter;
    static int _total;
    public static int _cost;
}
...
Point._cost = 10;
```

Classes - const and readonly (1)

unmodifiable field (readonly) - value established and assigned only during initialisation (also in class constructor)

constant field (const) - value established in compile time

```
class Sample {
    static public int scode1 = 1;
    static public int scode2 ;
    public readonly int rcode1 = 10;
    public readonly int rcode2;
    public const int CCODE1=100;
    public Sample() {
        rcode2 = 2;
        scode2 = 20;
    }
    static Sample() {
        scode2 = 3;
        // rcode2 = 3;    object required
    }
    void fun() {
        scode2 += 3;
        // rcode1 += 1;  // read-only field could not be assigned
        // CCODE1 = 4 ;  // left hand side must be a variable
    }
}
```

Classes - const and readonly (2)

unmodifiable field (readonly) - value established and assigned only during initialisation (also in class constructor)

constant field (const) - value established in compile time

```
class Program {
    static void bar(ref int par) {
        par += 1;
        Console.WriteLine("par: "+par);
    }

    static void Main(string[] args) {
        Sample s = new Sample();
        // Console.WriteLine("scode2: "+Sample.scode2); // class name required
        Console.WriteLine("scode2: "+Sample.scode2);    // scode2: 20
        Console.WriteLine("rcode2: "+s.rcode2);         // rcode2: 2
        //s.rcode1 = 15 ;                               // read-only field dould not be assigned
        //bar(ref s.rcode1);               // a readonly field could not be used with ref or out
        bar(ref Sample.scode2);                         // par: 21
    }
}
```

classes properties (1)

properties inheritance polymorphism abstraction interface enums

```
class Person {
  private string name; // field
  public string Name {  // property
    get { return name; }   // get method
    set { name = value; }  // set method
  }
}
```

Automatic Properties

```
class City {
  public string name  // property
  { get; set; }
}
```

class constructors and properties (2)

- constructors can call other constructors
- if there is no constructor there is one with no arguments
- static constructor must have no parameters

```
class Rectangle {
    public readonly int Width = 6;
    public readonly int Height;
    public Rectangle() {
        Height = 9;
    }
    public Rectangle(int w):this() {
        Width = w;
    }
}
```

class constructors and initializer properties (3)

- we can avoid creating many constructors for different parameter sets
- properties are used
- this is so called 'synctactic sugar' - the code generated do not change

```
class Rectangle {
    public int Width  { get; set; }
    public int Height  { get; set; }
}
Rectangle r = new Rectangle { Width = 10, Height = 15 };
```

equals:

```
Rectangle r = new Rectangle();
r.Width = 10;
r.Height = 15;
```

Object initialization

- Instead of a cluster of constructors, we have a parameterless and new syntax:

```
Student student = new Student { FirstName = "Adam", LastName = "Kot" };
Student student = new Student("132432") { FirstName = "Adam",
                                          LastName = "Kot" };

Student student = new Student("132432");
student.FirstName = "Adam";
student.LastName = "Kot" ;
```

Object initialization

- Instead of a cluster of constructors, we have a parameterless and new syntax:
- Creating internal objects by constructor

```
public class Rectangle {
    Point tl = new Point();
    Point br = new Point();
    public Point TL { get { return tl; } }
    public Point BR { get { return br; } }
    }
Rectangle r = new Rectangle {
    TL = { X = 0, Y = 1 },
    BR = { X = 2, Y = 3 }
};
```

Object initialization

- Instead of a cluster of constructors, we have a parameterless and new syntax:
- Creating internal objects by constructor
- Do not need to be a constant, f.e. Y=a-1

classes inheritance

properties inheritance polymorphism abstraction interface enums

```
class Vehicle {
  public string brand = "Ford";
  public void describe() {
    Console.WriteLine(brand);
  }
}

class Car : Vehicle {
  public string modelName = "Mustang";
}

class Program {
  static void Main(string[] args) {
    Car car = new Vehicle();
    car.describe();
    Console.WriteLine(car.brand + " " + car.modelName);
  }
}
```

classes polymorphism

properties inheritance polymorphism abstraction interface enums

```
class Vehicle {
  public string brand = "Ford";
  public virtual void describe() {
    Console.WriteLine("the vehicle is{0}", brand);
  }
}

class Car : Vehicle {
  public string modelName = "Mustang";
  public override void describe() {
    Console.WriteLine("the car is {0} {1}", brand, modelName);
  }
}

class Program {
  static void Main(string[] args) {
    Car car = new Vehicle();
    car.describe();
  }
}
```

classes abstraction (1)

properties inheritance polymorphism abstraction interface enums

Abstract class - is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class). But we still can use their static (but not abstract) methods!

Abstract method - can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

CSharp - Install
.Net
CLI
simple types
casting
strings
arrays_1
classes
modifiers
static
const
readonly
properties
initializers
inicjalizacja
inheritance
abstraction
interfaces
enums
exceptions
with
method params
default params
var
C# Top Level Statements (9.0)
CLI
Collections
C# Asynchronous programming
Delegation

classes abstraction (2)

properties inheritance polymorphism abstraction interface enums

```
abstract class Pet {                                        // Abstract class
  public abstract void animalSound();                       // Abstract method
  public void sleep() { Console.WriteLine("Zzz");}          // Regular method
  static public void wakeUp() { Console.WriteLine("Alarm");}// Static method
}
class Dog : Pet {                                           // Derived class
  public override void animalSound() {
    Console.WriteLine("The dog says: bark");
  }
}
class Program {
  static void Main(string[] args) {
    Dog dog = new Dog();                               // Create a Dog object
    dog.animalSound();                                 // Call the abstract method
    dog.sleep();                                       // Call the regular method
    Pet.wakeUp();                                      // Call the static method
  }
}
```

classes interface (1)

properties inheritance polymorphism abstraction interface enums

```
interface IPet {                                     // interface
  public void animalSound();                         // normal interface method
  public void sleep() { Console.WriteLine("Zzz");}   // default implementation (C# 8.0)
  public void wakeUp() { Console.WriteLine("Alarm");}// default implementation (C# 8.0)
}
class Dog : IPet {                                   // Derived class
  public void animalSound() {
    Console.WriteLine("The dog says: bark");
  }
}
class Program {
  static void Main(string[] args) {
    IPet pet = new Dog();                              // Create a Dog object
    pet.animalSound();                                 // Call the abstract method
    pet.sleep();                                       // Call the regular method
    pet.wakeUp();                                      // Call the default method (C# 8.0)
  }
}
```

classes interface (2)

properties inheritance polymorphism abstraction interface enums

Explicit - Available only by interface (but we could use casting)
Implicit - Available both way without casting

```
public interface IA {
    string Name { get;set; }
    void First(string prefix);
    void Second(string prefix);
}
public interface IB {
    string Name { get; set; }
    void First(string prefix);
    void Second(string prefix);
}
public static class IAExtension {
    public static IA Third(this IA ia, string postfix) {
        Console.WriteLine("Extension");
        return ia;
    }
}
```

classes interface (3)

properties inheritance polymorphism abstraction interface enums

```
public class Test : IA, IB {
    public string Name {
        get; set;
    }
    public void First(string prefix) {
        Console.WriteLine("{0} {1}",Name, prefix);
    }
    public void Second(string prefix) {
        Console.WriteLine("second:{0} {1}", Name, prefix);
    }
    void IB.Second(string prefix) {
        Console.WriteLine("Second(IB):{0} {1}", Name, prefix);
    }
}
```

classes interface (4)

```
public void f21() {
    Test t = new Test { Name = "t" };
    t.First("ala");
    IA ia = t;
    t = ia as Test;
    ia.First("from ia");
    t.First("t");
    t.Second("A");
    (t as IB).Second("B");
    ia.Third("EX").First("A");
    Console.ReadKey();
}
```

classes multiple enums (1)

properties inheritance polymorphism abstraction interface enums An enum represents a group of constants (unchangeable/read-only variables).

```
enum Level {
  Low,
  Medium,
  High
}
  ...
Level myVar = Level.Medium;
Console.WriteLine(myVar);       // outputs Medium
```

classes multiple enums (2)

Enum Values

```
enum Months {
  January= 1, // 1
  February,   // 2
  March,      // 3
  April=40,   // 40
  May         // 41
}
static void Main(string[] args) {
  int val = (int) Months.April;
  Console.WriteLine(val);       // outputs 40
}
```

C# Exceptions - try..catch..finally

- exception it is an object with attributes which can obtain a value

```
try {
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]); // how to create exception object
  if(1+1 != 2)                      // how to create exception object
    throw new ArithmeticException("adding numbers is not working");
}
catch (IndexOutOfRangeException e) {
  Console.WriteLine(e.message); // exception is handled



}
finally {
  Console.WriteLine("The 'try catch' is finished");    // free resources
}
```

C# Exceptions - try..catch..finally

- if nobody catches exception it is being catch by the system - and program may be terminated

```
try {
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]); // how to create exception object
  if(1+1 != 2)                      // how to create exception object
    throw new ArithmeticException("adding numbers is not working");
}
catch (IndexOutOfRangeException e) {
  Console.WriteLine(e.message); // exception is handled



}
finally {
  Console.WriteLine("The 'try catch' is finished");    // free resources
}
```

C# Exceptions - try..catch..finally

- `finally` block is always executed (regarding existence/not existence of the exception)

```
try {
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]); // how to create exception object
  if(1+1 != 2)                      // how to create exception object
    throw new ArithmeticException("adding numbers is not working");
}
catch (IndexOutOfRangeException e) {
  Console.WriteLine(e.message); // exception is handled



}
finally {
  Console.WriteLine("The 'try catch' is finished");    // free resources
}
```

C# Exceptions - try..catch..finally

- order of catch statements is important - the first matching type is used

```
try {
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]); // how to create exception object
  if(1+1 != 2)                      // how to create exception object
    throw new ArithmeticException("adding numbers is not working");
}
catch (IndexOutOfRangeException e) {
  Console.WriteLine(e.message); // exception is handled
} catch (ArithmeticException e) {   // this catch Arithmetic Exception
  Console.WriteLine(e.Message);     // data in exception object
  throw;                            // throws exception to next level
}
finally {
  Console.WriteLine("The 'try catch' is finished");    // free resources
}
```

C# Exceptions - try..catch..finally

- if there is no matching statement exception propagates up

```
try {
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]); // how to create exception object
  if(1+1 != 2)                      // how to create exception object
    throw new ArithmeticException("adding numbers is not working");
}
catch (IndexOutOfRangeException e) {
  Console.WriteLine(e.message); // exception is handled
} catch (ArithmeticException e) {   // this catch Arithmetic Exception
  Console.WriteLine(e.Message);     // data in exception object
  throw;                            // throws exception to next level
}
finally {
  Console.WriteLine("The 'try catch' is finished");   // free resources
}
```

C# Exceptions - try..catch..finally

- `Exception e` matches all exception types and should be last catch statement

```
try {
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]); // how to create exception object
  if(1+1 != 2)                      // how to create exception object
    throw new ArithmeticException("adding numbers is not working");
}
catch (IndexOutOfRangeException e) {
  Console.WriteLine(e.message); // exception is handled
} catch (Exception e) {                                 // this catch all exceptions
  Console.WriteLine(e.Message);     // data in exception object
  throw;                            // throws exception to next level
}
finally {
  Console.WriteLine("The 'try catch' is finished");    // free resources
}
```

C# Exceptions - with statement

- exception handling is used in `using`

```
using (MyResource myRes = new MyResource()) {
    myRes.DoSomething();
}
```

C# Exceptions - with statement

- exception handling is used in `using`
- is the same as `try-finally`:

```
{
    MyResource myRes= new MyResource();
    try {
        myRes.DoSomething();
    }
    finally {
        if (myRes!= null)
            ((IDisposable)myRes).Dispose();
    }
}
```

C# Exceptions - with statement

- exception handling is used in `using`
- is the same as `try-finally`:
- protected resource must implement `IDisposable` interface:

```
public interface IDisposable {
    public void Dispose ();
}


class MyResource:IDisposable {
    void DoSomething() {
        ...
    }
}
```

Method parameters

- overloaded methods simplifies interface

```
int myMethod(int x);                      // first
float myMethod(float x);                  // overloaded
double myMethod(double x, double y);      // overloaded
```

Method parameters

- overloaded methods simplifies interface
- by default, parameters are passed by value

```
class Sample {
  public String caption;
}
class Program {
  static void fun(Sample sample, int value) {
    sample.caption = "balbinka";
    value = 100;
  }
  static void Main(string[] args) {
    int v = 10;
    Sample s = new Sample();
    fun(s,v);
    Console.WriteLine(s.caption);    // balbinka
    Console.WriteLine(v);            // 10
  }
}
```

Method parameters

- overloaded methods simplifies interface
- by default, parameters are passed by value
- we can use in, out i ref

in -the parameter passed cannot be modified by the method

```
class Sample {
  public String caption;
}
class Program {
  static void fun(Sample sample, in int value) {
    sample.caption = "balbinka";
    // value = 100; // do not compile
  }
  static void Main(string[] args) {
    int v = 10;
    Sample s = new Sample();
    fun(s,v);
    Console.WriteLine(s.caption);    // balbinka
    Console.WriteLine(v);            // 10
  }
}
```

Method parameters

- overloaded methods simplifies interface
- by default, parameters are passed by value
- we can use in, out i ref

  in -the parameter passed cannot be modified by the method
  out -the parameter passed must be modified by the method

```
class Sample {
  public String caption;
}
class Program {
  static void fun(Sample sample, out int value) {
    sample.caption = "balbinka";
    value = 100;
  }
  static void Main(string[] args) {
    int v = 10;
    Sample s = new Sample();
    fun(s,v);
    Console.WriteLine(s.caption);   // balbinka
    Console.WriteLine(v);           // 100
  }
}
```

Method parameters

- overloaded methods simplifies interface
- by default, parameters are passed by value
- we can use `in`, `out` i `ref`

  `in` -the parameter passed cannot be modified by the method
  `out` -the parameter passed must be modified by the method
  `ref` -

```
class Sample {
  public String caption;
}
class Program {
  static void fun(Sample sample, in int value) {
    sample.caption = "balbinka";
    value = 100;     // if missing then treated as error
  }
  static void Main(string[] args) {
    int v = 10;
    Sample s = new Sample();
    fun(s,v);
    Console.WriteLine(s.caption);    // balbinka
    Console.WriteLine(v);            // 100
  }
}
```

MDefault parameters

- helps reduce method overloading
- we can use named parameters

```
class Logger {
    static public void DoMsg(string title="Title", string msg="Kom") {
        Console.WriteLine("{0} - {1}", title, msg);
    }
}
...
Logger.DoMsg();                 // outputs Title - Kom
Logger.DoMsg("Uwaga");          // outputs Uwaga - Kom
Logger.DoMsg("Baczna","Uwaga"); // outputs Baczna - Uwaga
// using named parameters
Logger.DoMsg(msg: "Uwaga");     // outputs Title - Uwaga
```

Declaration type resolved by compiler

- var - this is not an Variant
- This could not be a field in class or structure
- Require an assignment in declaration position (because a compiler must establish precise data type)

```
var a = 2;       // int -> eqivalence of int a = 2
object b = 2;    // boxing
int c = a;       // no casting
int d = (int) b; // casting
```

C# Top Level Statements (9.0)

- Starting in C# 9, there is no need to explicitly include a Main method in a console application project

```
dotnet new console
```

Program.cs

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

C# Top Level Statements (9.0)

- Starting in C# 9, there is no need to explicitly include a Main method in a console application project
  - The compiler generates a method to serve as the program entry point for a project with top-level statements.

Top-level code contains	Implicit Main
await and return	static async T
await	static async T
return	static int Mai
No await or return	static void Ma

C# Top Level Statements (9.0)

- Starting in C# 9, there is no need to explicitly include a Main method in a console application project
  - The compiler generates a method to serve as the program entry point for a project with top-level statements.
  - Only one top-level file

Top-level code contains	Implicit Main
await and return	static async T
await	static async T
return	static int Mai
No await or return	static void Ma

C# Top Level Statements (9.0)

- Starting in C# 9, there is no need to explicitly include a Main method in a console application project
  - The compiler generates a method to serve as the program entry point for a project with top-level statements.
  - Only one top-level file
- using directives

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("Hello");
builder.AppendLine("World!");

Console.WriteLine(builder.ToString());
```

C# Top Level Statements (9.0)

- Starting in C# 9, there is no need to explicitly include a Main method in a console application project
    - The compiler generates a method to serve as the program entry point for a project with top-level statements.
    - Only one top-level file
- using directives
- A file with top-level statements can also contain namespaces and type definitions, but they must come after the top-level statements

```
MyClass.TestMethod();
MyNamespace.MyClass.MyMethod();

public class MyClass
{
    public static void TestMethod()
    {
        Console.WriteLine("Hello World!");
    }

}

namespace MyNamespace
{
    class MyClass
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hello World from MyNamespace.MyClass.MyMethod!");
        }
    }
}
```

Graphical
User
Interfaces
(EGUI)
Julian Myrcha

CSharp -
Install

.Net
CLI
simple types
casting
strings
arrays_1
classes
modifiers
static
const
readonly
properties
initializers
inicjalizacja
inheritance
abstraction
interfaces
enums
exceptions
with
method params
default params
var
C# Top Level
Statements (9.0)
CLI

Collections

C#
Asynchronous
programming

Delegation

json

```
public class Person {
public int Id { get; set; }
public string FirstName { get; set; }
public string LastName { get; set; }
public string City { get; set; }
}
***
```

json

data.json:

```
[
  {
    "Id": 1,
    "FirstName": "James",
    "LastName": "May",
    "City": "Birmingham"
  },
  {
    "Id": 2,
    "FirstName": "Richard",
    "LastName": "Hammond",
    "City": "Manchester"
  }
]
```

data2.json:

```
[
  {
    "Id": 2,
    "FirstName": "Richard",
    "LastName": "Hammond",
    "City": "Bristol"
  },
  {
    "Id": 3,
    "FirstName": "Jeremy",
    "LastName": "Clarkson",
    "City": "London"
  }
]
```

```
public class Person {
public int Id { get; set; }
public string FirstName { get; set; }
public string LastName { get; set; }
public string City { get; set; }
}
```

***

C# collections

ArrayList -represents ordered collection of an object that can be indexed individually

- you can add and remove items from a list at a specified position using an index
- dynamic memory allocation
- adding
- searching and sorting items in the list.

Hashtable -It uses a key to access the elements in the collection.

SortedList -It uses a key as well as an index to access the items in a list.

- A sorted list is a combination of an array and a hash table.
- It contains a list of items that can be accessed using a key or an index.
- If you access items using an index, it is an ArrayList
- if you access items using a key t is a Hashtable.

Stack -a last-in, first out collection of object.

Queue -It represents a first-in, first out collection of object.

BitArray -It represents an array of the binary representation using the values 1 and 0.

Collections

To be a collection a class must implement IEnumerable<T> (lub IEnumerable)

```
public class ColorCollection : IEnumerable<String> {
    public IEnumerator<string> GetEnumerator() {
        yield return "red";
        yield return "green";
        yield return "blue";
    }
    // IEnumerable<T> derives from Enumerable.
    System.Collections.IEnumerator
    System.Collections.IEnumerable.GetEnumerator() {
        // Calls IEnumerator<string> GetEnumerator()
        return GetEnumerator();
    }
}
void f2h() {
    ColorCollection rgb = new ColorCollection();
    foreach (string s in rgb)
        Console.WriteLine("Value: {0}", s);
    Console.ReadKey();
}
```

Kolekcje (2) - Yield

- Keyword `yield` returns value and stops computations.
- Next attempt starts there and returns next value.

We can bleak iteration:

```
public IEnumerator<string> GetEnumerator() {
    yield return "red";
    if(DateTime.Now.Year==2010)
        yield break;
    yield return "green";
    yield return "blue";
}
```

Collections (3) - interfaces IDispose,IEnumerable, IEnumerable<T>

```
public interface IDisposable {
    void Dispose();
}

public interface IEnumerator {
    object Current { get; }
    bool MoveNext();
    void Reset();
}

public interface IEnumerator<out T> : IDisposable, IEnumerator {
    T Current { get; }
}
```

Collections (3) - interfaces IEnumerable, IEnumerable<T>, ICollection<T>

implemented interfaces

```
public interface IEnumerable {
    IEnumerator GetEnumerator();
}
public interface IEnumerable<out T> : IEnumerable {
    IEnumerator<T> GetEnumerator();
}
public interface ICollection<T> : IEnumerable<T>, IEnumerable {
    int Count { get; }
    bool IsReadOnly { get; }
    void Add(T item);
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    bool Remove(T item);
}
```

Collections (4) - Initialisation of collection

Initialisation require Add method `Add`

```
public class ColorCollection : IEnumerable<String> {
  string[] colors = new string[0];
  public IEnumerator<string> GetEnumerator() {
      foreach (string c in colors)
          yield return c;
  }
  // wersja bez generic potrzebna, bo IEnumerable<T>
  // dziedziczy z Enumerable.
  System.Collections.IEnumerator
  System.Collections.IEnumerable.GetEnumerator() {
      // Wola IEnumerator<string> GetEnumerator()
      return GetEnumerator();
  }
  public void Add(string p) {
      List<string> l = colors.ToList();
      l.Add(p);
      colors = l.ToArray<string>();
  }
}
```

Collections (5) - Foreach

foreach is handled by compiler:

```
static void f2h() {
  ColorCollection rgb = new ColorCollection { "red", "green", "blue" };
  foreach (string s in rgb)
    Console.WriteLine("Value: {0}", s);
}
// is equivalent
static void f2h()
{
  ColorCollection rgb = new ColorCollection { "red", "green", "blue" };
  using(IEnumerator<string> e = rgb.GetEnumerator())
    while(e.MoveNext())
      Console.WriteLine("Value: {0}", (string)e.Current);
}
```

Sample collection

collections derive from ICollection<T>

```
public class ColorCollection : ICollection<String> {
  string[] colors = new string[0];
  public string this[int idx] {
      get { return colors[idx]; }
  }
  public void Add(string p) {
        List<string> l = colors.ToList();
        l.Add(p);
        colors = l.ToArray<string>();
  }
  public void Clear()              { throw new NotImplementedException(); }
  public bool Contains(string item) { throw new NotImplementedException(); }
  public void CopyTo(string[] array, int arrayIndex) {
                                      throw new NotImplementedException(); }
```

Sample collection

collections derive from ICollection<T>

```
    public int Count {
        get { return colors.Count(); }
    }
    public bool IsReadOnly {    get {  throw new NotImplementedException(); } }
    public bool Remove(string item) {  throw new NotImplementedException(); }
}
...
void f2i() {
    ColorCollection rgb = new ColorCollection { "red","green","blue"};
    for(int i = 0; i < rgb.Count; i++)
        Console.WriteLine("Value: {0}", rgb[i]);
    Console.ReadKey();
}
```

Threads

Thread - Thread represents an actual OS-level thread, with its own stack and kernel resources

Threads

Thread - Thread represents an actual OS-level thread, with its own stack and kernel resources

- The problem with Thread is that OS threads are costly.
- Each thread you have consumes a non-trivial amount of memory for its stack, and adds additional CPU overhead as the processor context-switch between threads.
- Instead, it is better to have a small pool of threads execute your code as work becomes available.
- `Thread.Join()` gives a possibility to synchronize to the result

Threads

Thread - Thread represents an actual OS-level thread, with its own stack and kernel resources

ThreadPool - is a wrapper around a pool of threads maintained by the CLR.

Threads

Thread - Thread represents an actual OS-level thread, with its own stack and kernel resources

ThreadPool - is a wrapper around a pool of threads maintained by the CLR.

- you can submit work to execute at some point
- you can control the size of the pool
- you can't set anything else
- ThreadPool is best used for short operations where the caller does not need the result.

Threads

Thread - Thread represents an actual OS-level thread, with its own stack and kernel resources

ThreadPool - is a wrapper around a pool of threads maintained by the CLR.

Task - class from the Task Parallel Library offers the best of both worlds

Threads

Thread - Thread represents an actual OS-level thread, with its own stack and kernel resources

ThreadPool - is a wrapper around a pool of threads maintained by the CLR.

Task - class from the Task Parallel Library offers the best of both worlds

- Like the ThreadPool, a task does not create its own OS thread.
- Tasks are executed by a TaskScheduler; the default scheduler simply runs on the ThreadPool.

Threads

Thread - Thread represents an actual OS-level thread, with its own stack and kernel resources

ThreadPool - is a wrapper around a pool of threads maintained by the CLR.

Task - class from the Task Parallel Library offers the best of both worlds

We are using Tasks or higher level functionality

Tasks

- Tasks do not have name

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace ntr {
  class Program {
    public static void Main() {
      Thread.CurrentThread.Name = "Main";
      // Create a task and supply a user delegate by using a lambda expression
      Task task = new Task(() => Console.WriteLine("Hello from task"));
      // Start the task
      task.Start();
      // Output a message from the calling thread
      Console.WriteLine("Hello from thread <{0}>",
                        Thread.CurrentThread.Name);
      task.Wait();
    }
  }
}
```

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace ntr {
  class Program {
    public static void Main() {
      Thread.CurrentThread.Name = "Main";
      // Create a task and supply a user delegate by using a lambda expression
      Task<String> task = new Task<String>(() => "Hello from task");
      // Start the task
      task.Start();
      // Output a message from the calling thread
      Console.WriteLine("Hello from thread <{0}>",
```

Tasks

- Tasks do not have name
- We can wait for result

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace ntr {
  class Program {
    public static void Main() {
      Thread.CurrentThread.Name = "Main";
      // Create a task and supply a user delegate by using a lambda expression
      Task task = new Task(() => Console.WriteLine("Hello from task"));
      // Start the task
      task.Start();
      // Output a message from the calling thread
      Console.WriteLine("Hello from thread <{0}>",
                        Thread.CurrentThread.Name);
      task.Wait();
    }
  }
}
```

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace ntr {
  class Program {
    public static void Main() {
      Thread.CurrentThread.Name = "Main";
      // Create a task and supply a user delegate by using a lambda expression
      Task<String> task = new Task<String>(() => "Hello from task");
      // Start the task
      task.Start();
      // Output a message from the calling thread
      Console.WriteLine("Hello from thread <{0}>",
```

Tasks

- Tasks do not have name
- We can wait for result
- We can wait for result

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace ntr {
  class Program {
    public static void Main() {
      Thread.CurrentThread.Name = "Main";
      // Create a task and supply a user delegate by using a lambda expression
      Task task = new Task(() => Console.WriteLine("Hello from task"));
      // Start the task
      task.Start();
      // Output a message from the calling thread
      Console.WriteLine("Hello from thread <{0}>",
                        Thread.CurrentThread.Name);
      task.Wait();
    }
  }
}
```

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace ntr {
  class Program {
    public static void Main() {
      Thread.CurrentThread.Name = "Main";
      // Create a task and supply a user delegate by using a lambda expression
      Task<String> task = new Task<String>(() => "Hello from task");
      // Start the task
      task.Start();
      // Output a message from the calling thread
      Console.WriteLine("Hello from thread <{0}>",
```

async/await asynchronous programming (1)

- Threads may be difficult to design and test

async/await asynchronous programming (2)

- Threads may be difficult to design and test
- Library functions often may operate in parallel if we have proper design

```
static void Main(string[] args)  {
    callMethod();
}

public static async void callMethod()  {
    Task<int> task = Method1();  // auto task = Method1();
    Method2();
    int count = await task;
    Console.WriteLine("Total count is " + count);
}
```

async/await asynchronous programming (3)

- Threads may be difficult to design and test
- Library functions often may operate in parallel if we have proper design

```
public static async Task<int> Method1() {
    Console.WriteLine("enter Method 1 -------------");
    int count = 0;
    await Task.Run(() => {
        for (int i = 0; i < 300; i++)  {
            Console.WriteLine(" Method 1");
            count += 1;
        }
    });
    Console.WriteLine("leave Method 1 -------------");
    return count;
}

public static void Method2() {
    Console.WriteLine("enter Method 2 -------------");
    for (int i = 0; i < 25; i++)
        Console.WriteLine(" Method 2");
    Console.WriteLine("leave Method 2 -------------");
}
```

async/await asynchronous programming (4)

- Threads may be difficult to design and test
- Library functions often may operate in parallel if we have proper design

```
enter Method 1 -------------
 Method 1
 Method 1
 ...
 Method 1
enter Method 2 -------------
 Method 2
 Method 2
 ...
 Method 2
leave Method 2 -------------
 Method 1
 Method 1
 ...
 Method 1
 Method 1
 Method 1
leave Method 1 -------------
Total count is 300
```

Delegates - observator pattern (1)

Delegate

Delegations enable the implementation of the observer pattern. The basic syntax is

```
public delegate string LogIt(string info);
class ConsoleLogger {
    public string WriteToLog(string msg) {
        Console.WriteLine(ID + ":" + msg);
        return msg;
    }
    public string ID { get; set; }
}
class Program {
    static void f1() {
        ConsoleLogger logger = new ConsoleLogger { ID = "A" };
        LogIt logIt = new LogIt(logger.WriteToLog);
        logIt("FIRST message");
        ConsoleLogger logger2 = new ConsoleLogger { ID = "B" };
        logIt += new LogIt(logger2.WriteToLog);
        logIt("SECOND message");
    }
```

Delegates observator pattern (2)

Delegate

The strength of delegation is the ability to pass as a parameter:

```
public delegate string LogIt(string info);

void someFunc(LogIt param) {
    param("FIRST");
}

void f2a() {
    ConsoleLogger logger = new ConsoleLogger { ID = "A" };
    LogIt logIt = new LogIt(logger.WriteToLog);
    ConsoleLogger logger2 = new ConsoleLogger { ID = "B" };
    logIt += logger2.WriteToLog;
    someFunc(logIt);
}
```

delagates - anonymous methods (3)

- The implementing method can be in the same class
- When using anonymous methods, you don't even need to have an explicit implementation method

```
public delegate string LogIt(string info);

void f2b() {
    LogIt logIt = new LogIt(WriteToLog);
    logIt += delegate(string info){
        Console.WriteLine("C"+":"+info);
        return info;
    };
    someFunc(logIt);
    Console.ReadKey();
}
```

Delegations - operations on delegations

listener registration:

```
logIt=new LogIt( logger2.WriteToLog );
lub:
logIt+=new LogIt( logger2.WriteToLog );
```

unregister the listener:

```
logIt-=new LogIt( logger2.WriteToLog );
lub:
logIt=null;
```

calling list of listeners:

```
logIt( "FIRST message" );
lub:
foreach( LogIT log in logIt.GetInvocationList() ) {
    string result = log( "FIRST message" );
    Debug.WriteLine( "Returned result: {0}", result);
```

Delegations - standard templates

We have already declared delegations using templates

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
up to sixteen
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1,in T2,out TResult>(T1 arg1,T2 arg2);
 ... up to sixteen
```

Delegations - use of templates

We avoid creating our own delegations ...

```
void someFunc2(Func<String, String> log) {
    log("FIRST");
}
void f2c() {
    ConsoleLogger logger = new ConsoleLogger { ID = "A" };
    Func<String, String> logIt = logger.WriteToLog;
    logIt += delegate(string info) {
        Console.WriteLine("C" + ":" + info);
        return info;
    };
    someFunc2(logIt);
}
```

Events

- Incorrect initialization code can cut other listeners' registrations
- new keyword `event` restricting availability (no overwriting or calling outside of class)

```
class Listener {
    public Func<String, Object> logItDelegate;
    public event Func<String, Object> logItEvent;
    public void Go(String msg) {
        logItEvent("Go:" + msg);
    }
}
void f2f() {
    ConsoleLogger logger = new ConsoleLogger { ID = "A" };
    Listener ear = new Listener();
    ear.logItDelegate = logger.WriteToLog2;
    ear.logItDelegate("Delegate");
    ear.logItEvent += logger.WriteToLog2;
.allowed only } += i -=
    // ear.logItEvent("Event"); // poza klasa
    ear.Go("Event");
}
```

event-conventions

The handler returns void and has 2 parameters: sender and System.EventArgs

```
//public delegate void EventHandler<T>(object sender, T e)
                                           where T : EventArgs;
//public delegate void EventHandler(object sender, EventArgs e);
public class LogEventArgs : EventArgs {
    public String Msg { get; set; }
}
class Listener {
  public Action<Object, EventArgs> logItDelegate;
  public event Action<Object, EventArgs> logItEvent;
  public event EventHandler logItEvent2;
  public event EventHandler<LogEventArgs> logItEvent3;
  public void Go(String msg) {
    logItDelegate(this, new LogEventArgs { Msg = msg });
    logItEvent(this, new LogEventArgs { Msg = msg });
    if (logItEvent2 != null)
        logItEvent2(this, new LogEventArgs { Msg = msg });
    logItEvent3(this, new LogEventArgs { Msg = msg });
  }
}
```

event-conventions

The handler returns void and has 2 parameters: sender and System.EventArgs

```
class Program {
  public void WriteToLog3(object sender, EventArgs e) {
    Console.WriteLine("Event:" + ":" + (e as LogEventArgs).Msg);
  }
  void f2g1() {
    Listener ear = new Listener();
    ear.logItDelegate += WriteToLog3;
    ear.logItEvent  += WriteToLog3;
    ear.logItEvent2 += WriteToLog3;
    ear.logItEvent3 += WriteToLog3;
    ear.Go("Event");
  }
  static void Main(string[] args) {
    (new Program()).f2g1();
    Console.ReadKey();
  }
}
```

Delegate - metody anonimowe (poprzedni wykład)

- Metoda implementująca może być w tej samej klasie.
- Używając metod anonimowych nie musimy nawet mieć jawnie wydzielonej metody implementującej:

```
public delegate string LogIt(string info);
public string WriteToLog(string msg) {
    Console.WriteLine("A"+":"+msg);
    return msg;
}
void someFunc(LogIt log) {
    log("FIRST");
}
void f2b() {
    LogIt logIt = new LogIt(WriteToLog);
    logIt += delegate(string info){
        Console.WriteLine("C"+":"+info);
        return info;
    };
    someFunc(logIt);
    Console.ReadKey();
}
```

Lambda expression (1)

- Zamiast ”delegate (...)” mamy po prostu ”(...) =>”

```
public delegate string LogIt(string info);
public string WriteToLog(string msg) {
    Console.WriteLine("A"+":"+msg);
    return msg;
}
void someFunc(LogIt log) {
    log("FIRST");
}
void f2b() {
    LogIt logIt = new LogIt(WriteToLog);
    logIt += (string info) => {
        Console.WriteLine("C"+":"+info);
        return info;
    };
    someFunc(logIt);
    Console.ReadKey();
}
```

- Czytamy ”parametry ... przekształcają sie w ”

Lambda expression (2)

- delegate

```
logIt += delegate(string info) {
    Console.WriteLine("C"+":"+info);
    return info;
};
```

- lambda expression

```
logIt += (string info) => {
    Console.WriteLine("C"+":"+info);
    return info;
};
```

Lambda expression (3)

- Kompilator potrafi domyśleć się typu - tak jakby było to var

```
logIt += (info) => {
    Console.WriteLine("C"+":"+info);
    return info;
};
```

- jak jest jeden parametr i nie ma typu, to można darować sobie nawiasy

```
logIt +=  info => {
    Console.WriteLine("C"+":"+info);
    return info;
};
```

Lambda expression (4)

- dwie wersje:

```
delegate T Sum<T>(T a, T b);
public void f2m() {
    Sum<int> statement = (a, b) => { return a + b; };
    Sum<int> expression = (a, b) =>  a + b;
    Console.WriteLine(statement(4,5));
    Console.WriteLine(expression(4, 5));
    Console.ReadKey();
}
```

- możliwości:

```
( int a, int b ) => { return a + b; } // typowane, statement
( int a, int b ) => a + b;            // typowane, wyrazenie
( a, b ) => { return a + b; }         // domyslne, statement
( a, b ) => a + b                     // domyslne, wyrazenie
( x ) => sum += x                     // Pojedynczy parametr w nawiasach
x => sum += x                         // i bez nawiasow
() => sum + 1           // ale jak nie ma parametru to nawiasy musza byc
```

zmienne zewnętrzne

```
public void f2r() {
    int counter = 0;                // zmienna zewnetrzna
    LogIt logIt = info => {
        Console.WriteLine("C" + ":" + info);
        counter++;
        return info;
    };
    Console.WriteLine(counter);
    logIt("A");
    Console.WriteLine(counter);
}
```

- Kompilator musi się trochę nagłówkować, aby zmienna istniała tak długo, jak długo jest delegacja:
  - Tworzy ukrytą klasę, zawierającą wszystkie zmienne zewnętrzne oraz metodę delegacji
  - Tworzy obiekt tej klasy
  - Zamienia odwołania do zmiennych zewnętrznych na odwołania do pól obiektu
  - śmieciarka czeka, aż ktoś odepnie się od delegacji

Generics

```
public class Test<T>  {
  public delegate T Func(T a, T b);
    public static T Aggregate(List<T> l, Func f) {
      T result = default(T);
      foreach (T value in l)
        result = f(result, value);
      return result;
    }
}
```

Differences from C ++

- Generics compilation as such (and in C ++ there was a compilation of every text instance of the type)

```
public class Test<T> {
        ...
    public static T Sum(T a, T b) {
        return a + b;                // it won't compile - it doesn't know what it is +
    }
}
```

Generics

Methods operating on objects must be from the outside

```
class Program {
    static int intSum(int a, int b) {
        return a + b;
    }
    static string strSum(string a, string b) {
        return a + b;
    }
    static void Main(string[] args) {
        List<int> intData = new List<int>(){10,20,30};
        Console.WriteLine(Test<int>.Aggregate(intData, intSum)); // 60
        List<string> strData = new List<string>(){"10","20","30" };
        Console.WriteLine(Test<string>.Aggregate(strData, strSum));
                                                                //102030
        Console.ReadKey();
    }
}
```

Generics-construcors

the constructor does not require a type:

```
public struct Pair<T>: IPair<T> {
  public Pair(T first, T second) {
    this.first = first;
    this.second = second;
  }
  public Pair(T first) {
    this.first = first;
    this.second = default(T);
  }
}
public class BinaryTree<T>  where T: System.IComparable<T> {
...
```

- `default<T>` is the default value for a given type

Generics-constraints

- Sometimes generic only makes sense for special types

Restriction due to the existence of a base class for the parameter

```
public class EntityDictionary<TKey, TValue>
: System.Collections.Generic.Dictionary<TKey, TValue>
where TValue : EntityBase {
...
}
```

Generics-constraints

Restriction by type being a value or reference - e.g. Nullable<T> require a value

```
public struct Nullable<T> :
IFormattable, IComparable,
IComparable<Nullable<T>>, INullable
where T : struct {
// ...
}
```

Limitation on the existence of a parameterless constructor

```
public class EntityDictionary<TKey, TValue> :Dictionary<TKey, TValue>
                          where TKey: IComparable<TKey>, IFormattable
                          where TValue : EntityBase<TKey>, new() {
                          ...
```

Generic methods

```
public static class MathEx {
  public static T Max<T>(T first, params T[] values)
                            where T : IComparable<T> {
    T maximum = first;
    foreach (T item in values) {
    if (item.CompareTo(maximum) > 0) {
      maximum = item;
    }
  }
  return maximum;
}
```

Generics and execution

```
class Program {
    static void Main(string[] args) {
        List<int> Values = new List<int>();
        Values.Add(100); Values.Add(200);
        int sum = Test<int>.Aggregate(
           Values, delegate( int a, int b ) { return a + b; }
         );
        Console.WriteLine(sum);                // 300
        List<string> Strings = new List<string>();
        Strings.Add("100"); Strings.Add("200");
        string ssum = Test<string>.Aggregate(
           Strings, delegate(string a, string b) { return a+b;}
         );
        Console.WriteLine(ssum);        // 100200
        Console.ReadLine();
    }
}
```