

C# .NET

Laborator 8

Exceptii

Exceptii

- Situatii eronate in care CLI nu poate continua executia normala a programului
 - **Executia normala a programului este intrerupta!**
 - Restul functiei nu mai este executat
 - Restul programului nu mai este executat
- Se intampla la runtime
- Inlocuiesc vechile coduri de eroare care trebuiau tratate in fiecare functie in parte
- Exemplu de exceptii
 - Null reference exception
 - Division by 0
 - Stack overflow exception

Exceptii - clasa Exception

- Informatiile despre situatiile exceptionale
 - Reprezentate sub forma unor obiecte de tipul *Exception*
 - Toate tipurile de exceptii sunt subclase ale clasei *Exception*

```
...public class Exception : ISerializable
{
    ...public Exception();
    ...public Exception(string? message);
    ...public Exception(string? message, Exception? innerException);
    ...protected Exception(SerializationInfo info, StreamingContext context);

    ...public virtual string? StackTrace { get; }
    ...public virtual string? Source { get; set; }
    ...public virtual string Message { get; }
    ...public Exception? InnerException { get; }
    ...public int HResult { get; set; }
    ...public virtual IDictionary Data { get; }
    ...public MethodBase? TargetSite { get; }
    ...public virtual string? HelpLink { get; set; }

    ...protected event EventHandler<SafeSerializationEventArgs> SerializeObjectState;

    ...public virtual Exception GetBaseException();
    ...public virtual void GetObjectData(SerializationInfo info, StreamingContext context);
    ...public Type GetType();
    ...public override string ToString();
}
```

Exceptii - clasa Exception

- Message
 - Mesajul exceptiei
 - O descriere human-friendly
- InnerException
 - Obiect de tipul exceptiei
- Stack Trace
 - Numele functiilor parcurse din stiva de program precum si nr. liniilor apelurilor
- ToString()
 - Implementarea default va afisa tipul exceptiei, mesajul, pre cum si continutul InnerException-ului si al Stack trace-ului

```
...public class Exception : ISerializable
{
    ...public Exception();
    ...public Exception(string? message);
    ...public Exception(string? message, Exception? innerException);
    ...protected Exception(SerializationInfo info, StreamingContext context);

    ...public virtual string? StackTrace { get; }
    ...public virtual string? Source { get; set; }
    ...public virtual string Message { get; }
    ...public Exception? InnerException { get; }
    ...public int HResult { get; set; }
    ...public virtual IDictionary Data { get; }
    ...public MethodBase? TargetSite { get; }
    ...public virtual string? HelpLink { get; set; }

    ...protected event EventHandler<SafeSerializationEventArgs> SerializeObjectState;

    ...public virtual Exception GetBaseException();
    ...public virtual void GetObjectData(SerializationInfo info, StreamingContext context);
    ...public Type GetType();
    ...public override string ToString();
}
```

Exceptii - aruncarea unei exceptii

- Situatiile exceptionale sunt marcate prin “aruncarea” unui obiect de tip exception
 - Obiectul de tip exception trebuie sa incapsuleze toate informatiile necesare identificarii si tratarii situatiei exceptionale.
 - Continutul exceptiei va ajuta programatorul in timpul debugging-ului.

```
public void RegisterStudent(Guid id, string name)
{
    if (id == null)
        ► throw new Exception("Guid null. Cannot register student");
    students.Add(new Student(id, name));
}
```

- Instructiunea throw
 - Urmata de un obiect de tipul exceptie
 - Va “comuta” programul in starea de exceptie

Exceptii – tratarea exceptiilor

- try
 - Orice exceptie aruncata in interiorul block-ului *try* va fi prinsa in interiorul block-ului marcat de clauza “*catch*”
- catch
 - Va “prinde” si va permite “tratarea” exceptiilor aruncate in block-ul *try* afferent
 - Daca exceptia nu este rearuncata, executia instructiunilor care urmeaza clauzei catch va fi efectuata liniar, fara a fi afectata de existenta exceptiei. **Programul continua in modul normal**
 - In paranteze (Exception e)
 - va fi specificat tipul exceptiei pe care clauza catch il prinde
 - va fi declarant un identificator al obiectului exception prins
- Finally - optional
 - Toate instructiunile din block-ul finally vor fi rulate indiferent de aparitia, si tratarea sau nu a unei exceptii
- Try-finally
 - Clauza catch este in acest caz optionala, exceptiile nu vor fi prinse dar continutul lui finally va fi rulat chiar si in cazul aparitiei unei exceptii netratate

```
try {  
    Test();  
}  
catch(Exception e)  
{  
    Console.WriteLine(e);  
}  
finally  
{  
    //this will be executed regardless  
}
```

```
try {  
    Test();  
}  
finally  
{  
    //this will be executed regardless  
}
```

Exceptii – mostenire

- Mostenirea exceptiilor
 - Permite definirea unor exceptii “custom”, personalizate
 - Oferă un mecanism mai detaliat de a trata în mod individual exceptiile.
 - Permite realizarea unor ierarhii de exceptii, pe mai multe nivele (destul de rar întâlnit în practică)
 - Putem oferi parametri, campuri, etc pentru exceptii – sunt clase normale
 - Naming convention : sufixul “Exception”
- Toate excepțiile trebuie mostenite din clasa “*Exception*”

```
class InsufficientFundsException : Exception
{
    private const string InsufficientFundsConst = "insufficient funds.";
    private const string InsufficientFundsTemplateConst = "insufficient funds. {0} expected {1} found.";
    0 references
    public InsufficientFundsException():base(InsufficientFundsConst)
    {
    }

    0 references
    public InsufficientFundsException(double expected, double actual)
        :base(string.Format(InsufficientFundsTemplateConst, expected, actual))
    {
    }
}
```


Best practices

- Exceptiile nu trebuie ascunse niciodata – bad design
 - Ascunderea exceptiei : exceptii prinse care nu sunt logate sau tratate corespunzator
- Folosirea exceptiilor predefinite – DA
 - `ArgumentException`, `InvalidOperationException`, `NotSupportedException`, `ArgumentOutOfRangeException`
- Catch-uri peste tot – NU
 - Exceptiile vor fi prinse doar acolo unde POT fi tratate.
 - Ex: cnp introdus de la tastatura gresit – exceptia va fi prinsa in UI si va fi afisat un mesaj corespunzator
 - Conexiunea spre un server a esuat
 - Exceptia va fi prinsa in clasa in care conexiunea a esuat pentru a reincerca o noua conexiune
 - In situatia in care trebuie eliberate resurse sau operatiunile precedente trebuie sterse, exceptia va fi prinsa iar apoi rearuncata.

Testarea algoritmilor

Testarea algoritmilor

- Ce este testarea?
- Procesul prin care determinam daca un produs software se ridica la nivelul cerintelor si asteptarilor



Testarea algoritmilor

- Ex: Calculati valoarea functiei pentru un numar x citit de la tastatura

- Cazuri generale
 - Pe toate ramurile posibile
- Capete de interval
- Cazuri invalide

$$\begin{cases} 7x^2, & \text{daca } x \in (0, \frac{1}{2}] \\ 4x - 5, & \text{daca } x \in (\frac{1}{2}, 17) \\ 14x - 7, & \text{daca } x \in [17, \infty) \end{cases}$$

Gandirea iterativă - testare

- Exercițiu

- Scrieți o funcție care va calcula suma numerelor pare de la 1 până la n .

Algoritm


1. Testăm cazul general
2. Testăm situații excepționale

Unit tests

Fundamente

Teste unitare

- Sunt teste **automate** care ne ajuta sa testam automat o “unitate de cod”
- Unitate de cod
 - De obicei o metoda publica a unei clase publice
- Putem scrie teste pentru
 - Metode **publice** apartinand unor clase **publice** din orice assembly referentiat
 - Campuri/proprietati **publice** apartinand unor clase **publice**
- Testele unitare
 - Trebuie sa fie IZOLATE, sa nu depinda de
 - GUI si user interaction
 - alte programe
 - front-end-uri
 - data-base-uri
 - Trebuie sa acopere cat mai multe cazuri
 - Trebuie sa acopere cat mai multe branch-uri din cod – code coverage
- Testele unitare nu vor testa
 - O componenta software intreaga
 - Un workflow intreg cap-coada



Acestea NU sunt teste unitare.
Sunt alte tipuri de teste **automate**

Teste unitare – beneficii

- Sunt automate
 - Odata scrise, ele vor fi rulate printr-o simpla apasare de cod
 - Interactiunea unui tester nu mai este necesara
 - Pot fi integrate in build pipeline, si rulate automat la fiecare commit
 - Consecinta : Scad costul de testare
- “Fixeaza” un anumit comportament
 1. Scriem functia
 2. Scriem testele relevante pentru
 - Cazuri generale
 - Cazuri exceptionale (limite de interval, null pointeri, particularitati)
 3. Cand un alt programator opereaza modificari asupra functiei
 1. Va rula testele unitare
 - daca au trecut, totul e in regula
 - Daca testele nu au trecut – conflict cu ceea ce autorul a vrut de la acea functie. Doua variante
 - a) Comportamentul functiei intr-adevar a trebuit schimbat radical
 - b) Schimbarile au introdus o eroare (cel mai probabil) – corectarea erorii
 2. Va scrie propriile teste unitare care sa acopere noile cazuri
- Putem scrie teste inainte de a scrie codul
 - Test-driven-design
- Ne ajuta sa descoperim bug-uri repede
 - Chiar in timpul developmentului!
 - De cele mai multe ori momentul descoperirii bug-ului este chiar mai important decat bugul in sine

TestMethod

- Componente – triple A pattern / AAA pattern
 - **Arrange**
 - “aranjam” toate cele necesare rularii testului
 - parametrii,
 - Obiecte aditionale
 - initializari
 - **Act**
 - “actionam” prin executia efectiva a metodei care trebuie testata, cu parametrii definiti in “arrange”
 - **Assert**
 - Verificam daca rezultatul primit este in conformitate cu cel asteptat.

Observatie:

- Metoda de test va testa un singur caz pentru o singura metoda (are un singur assert) – best practice
- Numele metodei poate fi modificat.
- In cazul in care assert fail-uie, acest lucru va fi specificat de catre VS sau build pipeline

```
[TestMethod()]  
0 references  
public void AddTest()  
{  
    //Arrange  
    int x = 2;  
    int y = 3;  
    int expectedResult = 5;  
    var calculator = new Calculator();  
  
    //Act  
    var result = calculator.Add(x, y);  
  
    //Assert  
    Assert.AreEqual(expectedResult, result, $"Adunarea a esuat. Rezultatul " +  
        $"trebuia sa fie {expectedResult} si a fost {result}");  
}
```

Adnotare (notita)
Arata ca “AddTest” e o metoda de test

Clasa statica “Assert”

- Pune la dispozitie metode prin care vom compara rezultatul rularii cu cel asteptat
- AreEqual
 - verificare de egalitate intre doua valori cu ajutorul metodei “Equals”
 - Pt valori numerice ofera toleranta pentru valorile double
 - Ofera un string personalizabil cu scopul de a identifica mai usor test case-ul sau de a o tipari in rapoarte
- AreNotEqual
 - Opusul lui AreEqual

```
[TestMethod()]
0 references
public void AddTest()
{
    //Arrange
    int x = 2;
    int y = 3;
    int expectedResult = 5;
    var calculator = new Calculator();

    //Act
    var result = calculator.Add(x, y);

    //Assert
    Assert.AreEqual(expectedResult, result, $"Adunarea a esuat. Rezultatul " +
        $"trebuia sa fie {expectedResult} si a fost {result}");
}
```

Clasa statica “Assert”

- Definitie: enunt afirmativ sau negativ.
- In caz de fail, testul va “*cadea*”
- AreSame
 - succes daca doua referinte point-eaza spre acelasi obiect
- AreNotSame opusul lui AreTheSame
- IsTrue
 - Succes daca o valoare bool cu “true”
- IsNull
 - Succes daca o variabila este null.
- IsNotNull
 - Opusul lui IsNull
- IsInstanceOf
 - Succes daca un parametru este o instanta a altui expected result-ului

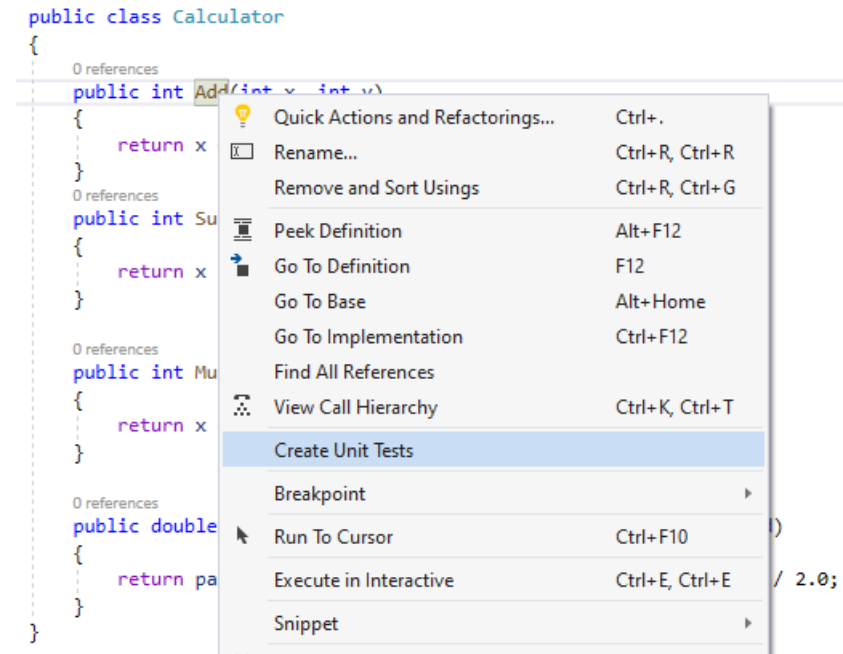
Exemplu

- Clasa publica
- Doar metodele publice ale clasei publice pot fi testate unitar

```
public class Calculator
{
    0 references
    public int Add(int x, int y)
    {
        return x + y;
    }
    0 references
    public int Subtract(int x , int y)
    {
        return x - y;
    }
    0 references
    public int Multiply(int x, int y)
    {
        return x * y;
    }
    0 references
    public double ParalelipipedVolume(IParalelipiped paralelipiped)
    {
        return paralelipiped.X * paralelipiped.Y * paralelipiped.Z / 2.0;
    }
}
```

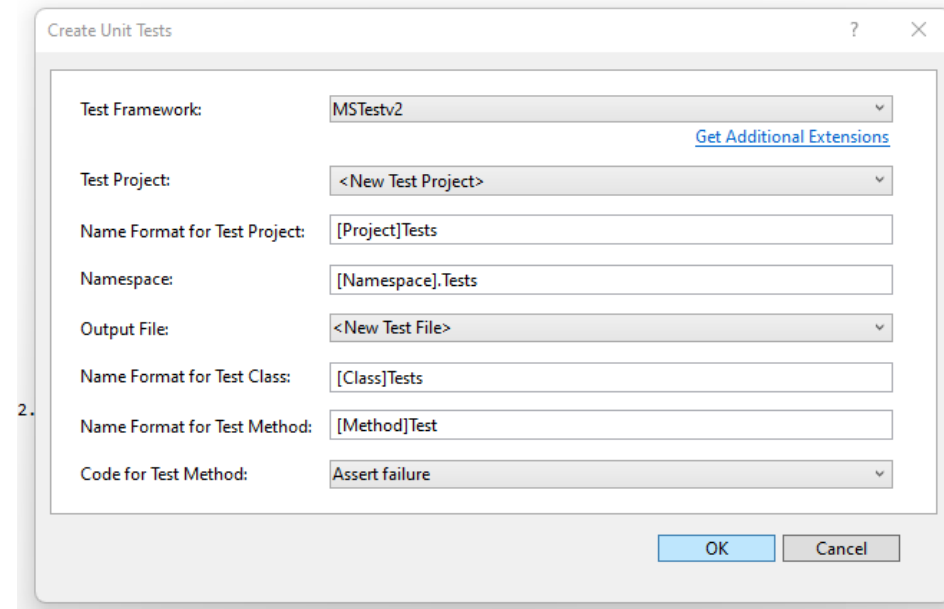
Exemplu

1. Right click pe metoda pentru care vom scrie un test
2. Click pe “create unit tests”



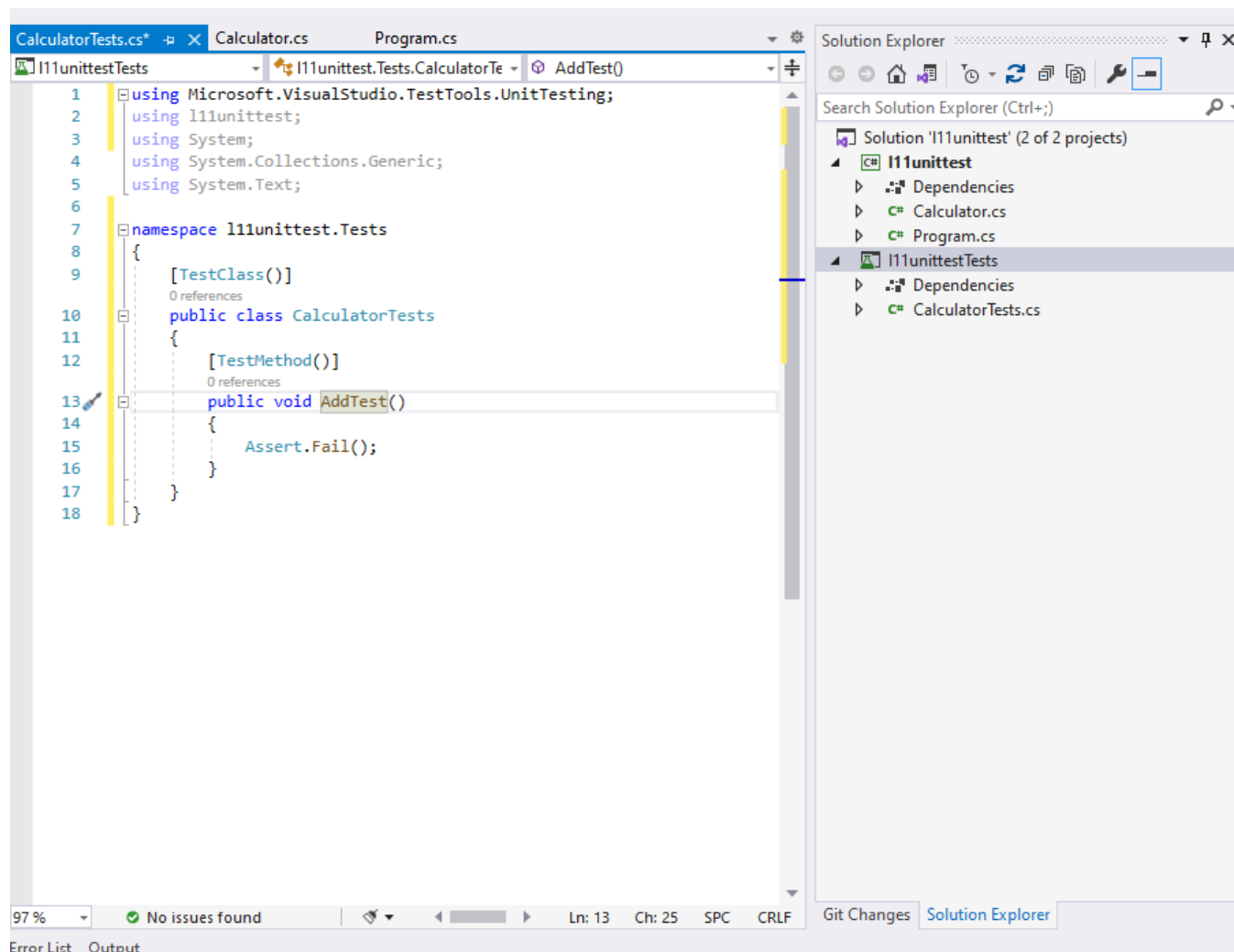
Exemplu

- Optiuni de configurare a testului unitar, a clasei de test precum si a proiectului de teste din care testul unitar va face parte
- Apasam “OK”



Exemplu

- Proiectul a fost creat
 - Pentru fiecare Assembly va fi creat un proiect nou
 - Structura namespace-urilor din testProject va mirror-ui pe a celor din proiectul testat
- Clasa de test a fost adaugata
 - Clasa este in relatia 1-1 cu clasa ale carei teste le va contine
 - Toate metodele clasei testate vor fi testate in aceiasi clasa. Contine adnotarea *TestClass()*
- Metoda de test
 - Adnodarea *TestMethod()*



The screenshot displays the Visual Studio IDE with a solution named 'I11unittest' containing two projects: 'I11unittest' and 'I11unittestTests'. The 'I11unittestTests' project is selected, and the file 'CalculatorTests.cs' is open in the editor. The code in 'CalculatorTests.cs' is as follows:

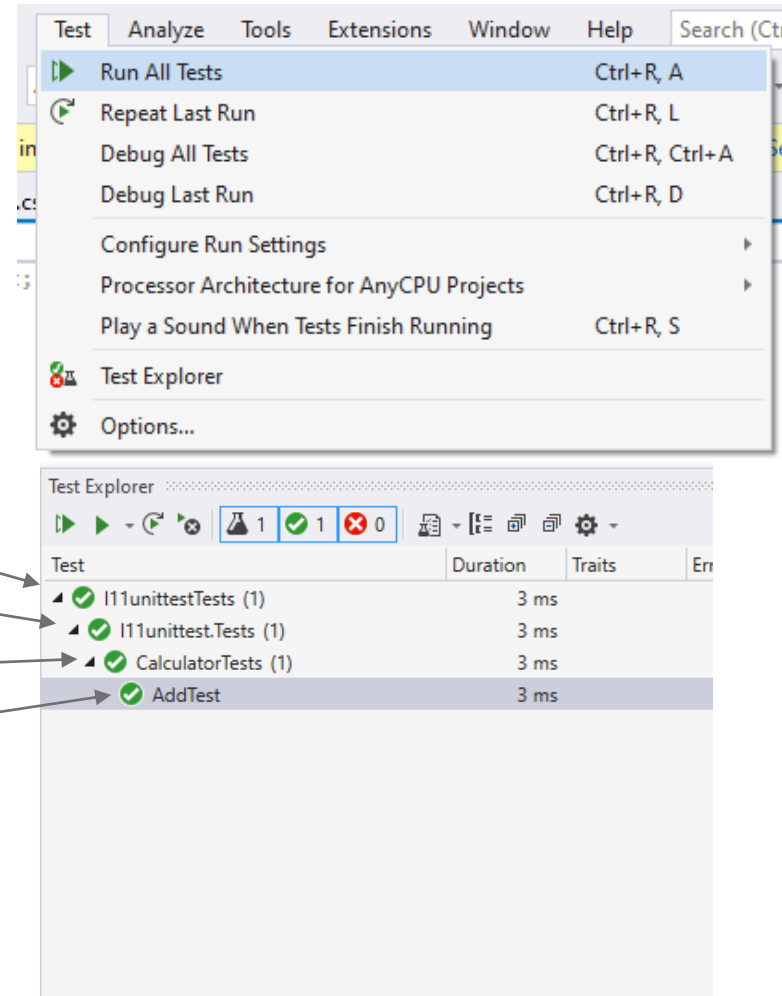
```
1 using Microsoft.VisualStudio.TestTools.UnitTesting;
2 using I11unittest;
3 using System;
4 using System.Collections.Generic;
5 using System.Text;
6
7 namespace I11unittest.Tests
8 {
9     [TestClass()]
10     public class CalculatorTests
11     {
12         [TestMethod()]
13         public void AddTest()
14         {
15             Assert.Fail();
16         }
17     }
18 }
```

The Solution Explorer on the right shows the project structure:

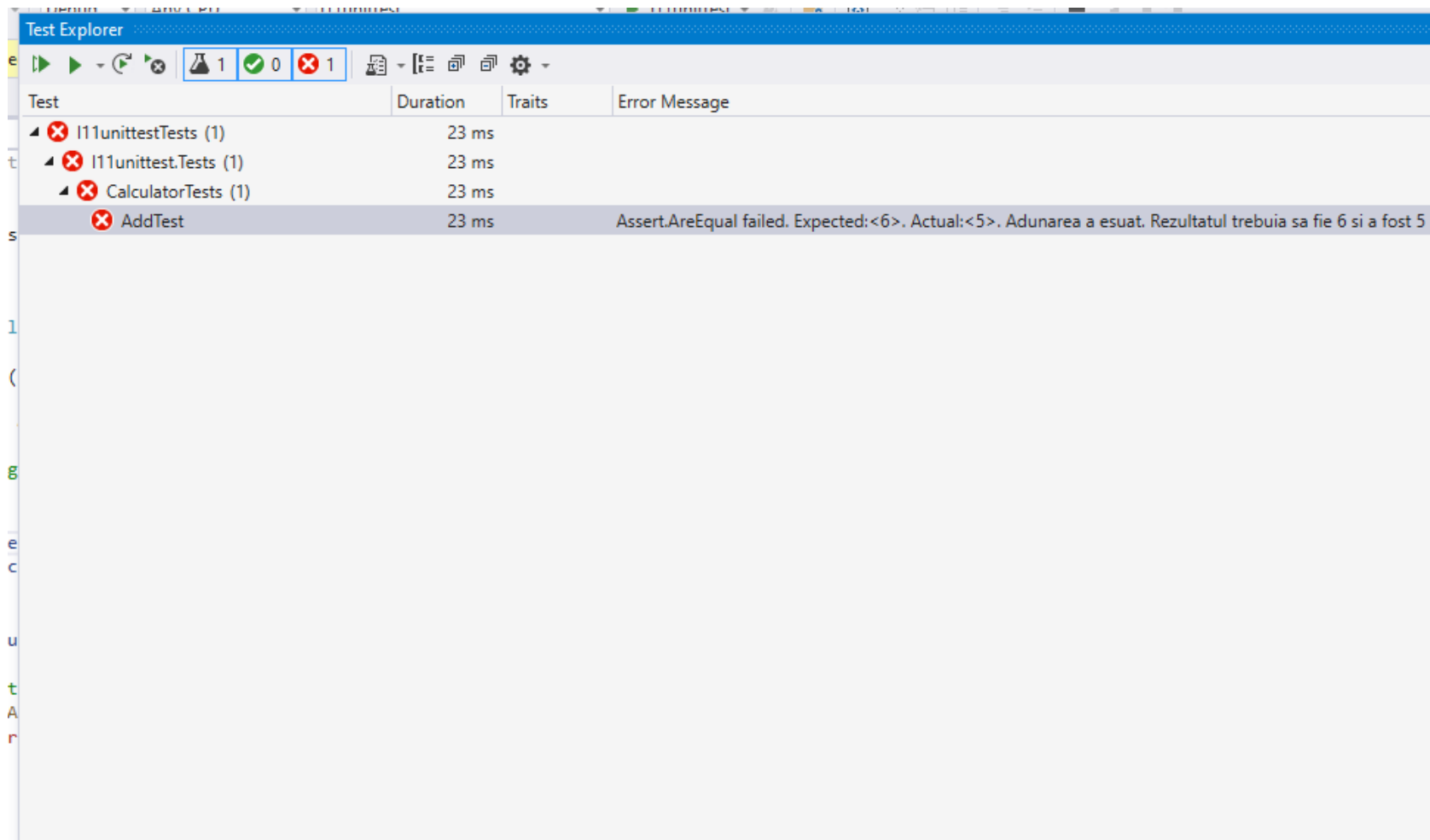
- Solution 'I11unittest' (2 of 2 projects)
 - I11unittest
 - Dependencies
 - Calculator.cs
 - Program.cs
 - I11unittestTests
 - Dependencies
 - CalculatorTests.cs

The status bar at the bottom indicates '97 %', 'No issues found', and the cursor is at line 13, column 25.

- Test – Run All Tests
 - Va rula toate testele unitare din solutia curenta
 - Testele “success” le va marca prin intermediul unei buline verzi
 - Testele “fail” vor fi marcate ca atare
- Numele proiectului de test
- Namespace-ul proiectului de test
- Numele clasei de test
- Numele metodei de test



Exemplu - eroare



The screenshot shows the Test Explorer window with a toolbar at the top containing icons for running tests, a summary of results (1 failed, 0 passed, 1 error), and a settings icon. Below the toolbar is a table with columns for Test, Duration, Traits, and Error Message. The table lists a hierarchy of tests: I11unittestTests (1), I11unittest.Tests (1), CalculatorTests (1), and a specific test named AddTest which has failed. The error message for AddTest states: 'Assert.AreEqual failed. Expected:<6>. Actual:<5>. Adunarea a esuat. Rezultatul trebuia sa fie 6 si a fost 5'.

Test	Duration	Traits	Error Message
✖ I11unittestTests (1)	23 ms		
✖ I11unittest.Tests (1)	23 ms		
✖ CalculatorTests (1)	23 ms		
✖ AddTest	23 ms		Assert.AreEqual failed. Expected:<6>. Actual:<5>. Adunarea a esuat. Rezultatul trebuia sa fie 6 si a fost 5

Teste unitare-exceptii

- ExpectedException
 - Adnotare
 - Specifica tipul exceptiei care ne asteptam sa fie aruncata pentru testul current
- Assert
 - Assert.Fail() - daca exceptia nu a fost aruncata atunci testul unitar va fi automat trecut in modul Fail
 - Functia testata functioneaza corect cand pentru acest caz este aruncata o exceptie!

```
[TestMethod()]  
[ExpectedException(typeof(DivideByZeroException))]  
✓ | 0 references  
public void DivideTest()  
{  
    // Arrange  
    int x = 10;  
    int y = 0;  
    var calculator = new Calculator();  
  
    //Act  
    calculator.Divide(x, y);  
  
    //Assert  
    Assert.Fail();  
}
```

Test-driven development : snippets

- Proces de dezvoltare software in care codul este scris **dupa** ce
 - cerintele (requirements) sunt analizate
 - testele sunt scrise
- Exercițiu
 - Scrieti o functie care va calcula factorialul unui numar.
 - Scrieti testele,
 - Scrieti codul
 - Rulati codul
 - Rulati testele

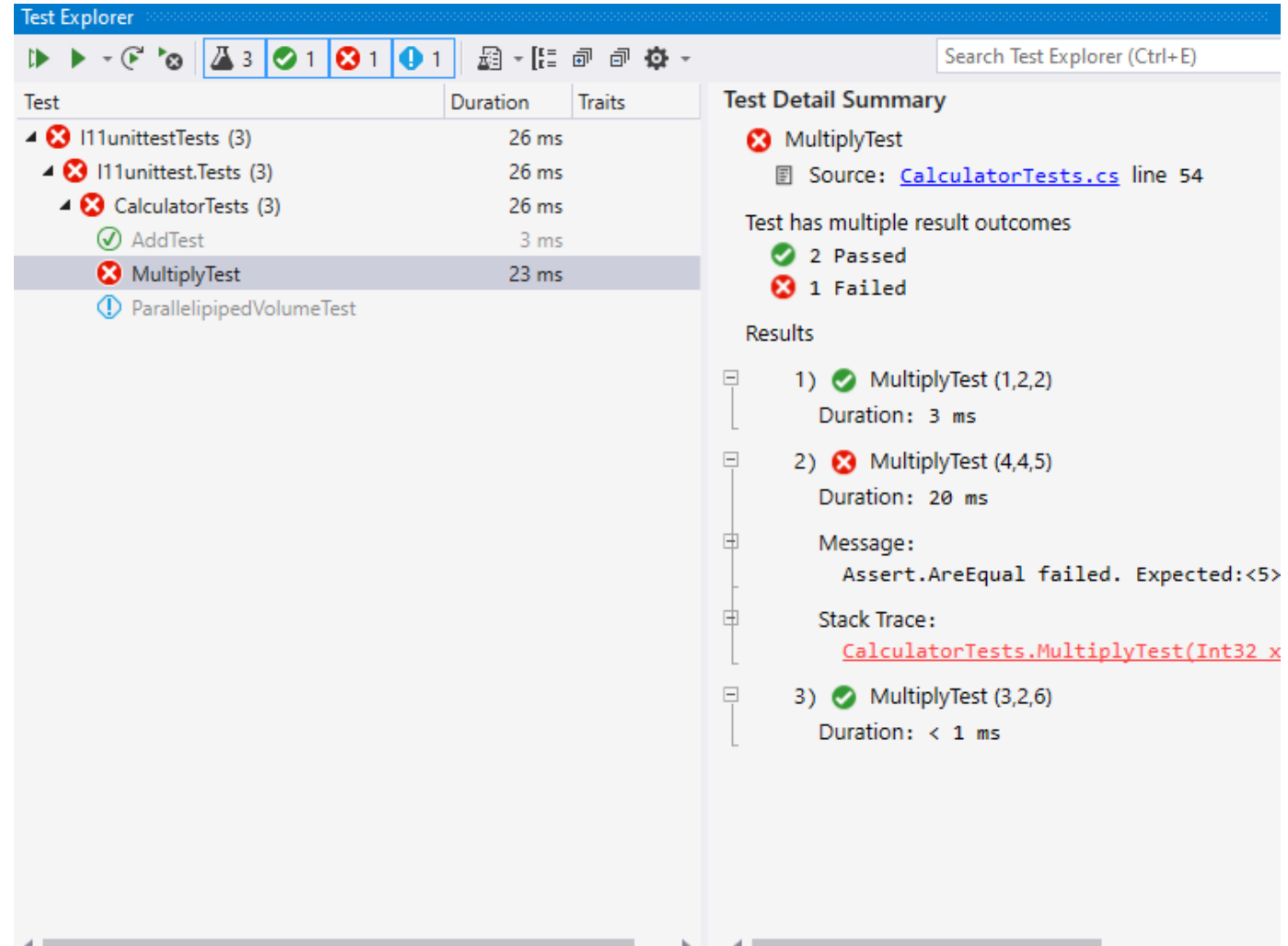
Metode de test parametrizate

- Metodele de test sunt parametrizabile
 - Sunt metode de clase
 - Pot folosi membri ai clasei initializati in constructor, etc...
- Folosim adnotarea “*DataRow*”
 - valorile parametrilor vor fi definite in adnotare
 - parametri vor fi folositi la executia testului

```
[TestMethod()]  
[DataRow(1, 2, 2)]  
[DataRow(4, 4, 5)]  
[DataRow(3, 2, 6)]  
0 | 0 references  
public void MultiplyTest(int x, int y, int expectedResult)  
{  
    var calculator = new Calculator();  
    var result = calculator.Multiply(x, y);  
    Assert.AreEqual(expectedResult, result);  
}
```

Metode de test parametrizate

- Metoda de test va fi rulata de mai multe ori astfel incat toate testcase-urile sa fie acoperite
- Rezultatele rularilor individuale ale aceleiasi metode vor fi prezentate separate
- Nu sunt necesare testmethod-uri individuale pentru fiecare testcase in parte



Lambda expressions, predicates

Lambda

- Expresii lambda
 - [Microsoft documentation](#)
 - [CSharpCorner](#)
- Inlocuiesc functiile anonime
- Se bazeaza pe operatorul lambda:
- => *“trece in”, “devine”*
 - *lambda_input => lambda_body*
 - Parametru => expresie
 - *x => x+1*
 - *listaDeParametri=> expresie*
 - *(x,y) => x+y*
 - *name=>console.WriteLine(name)*

Lambda

- Expresia lambda poate fi persistata sub forma unui delegate
- Delegate – tipul corespunzator

```
delegate int AddInt(int x, int y);  
0 references  
class Program  
{  
    0 references  
    static void Main(string[] args)  
    {  
        AddInt add = (int x, int y) => { return x + y; };  
  
        Console.WriteLine(add(2, 3));  
        //5  
    }  
}
```


Lambda

- Parametri

- Int, int

- Lambda body

- Returneaza int

- Return type-ul lambda-ului este dedus **implicit** pe baza tipului returnat!

```
delegate int AddInt(int x, int y);
```

```
0 references
```

```
class Program
```

```
{
```

```
0 references
```

```
static void Main(string[] args)
```

```
{
```

```
AddInt add = (int x, int y) => { return x + y; };
```

```
Console.WriteLine(add(2, 3));
```

```
//5
```

```
}
```

```
}
```

Lambda

- Tipurile parametrilor – deduse *implicit* pe baza tipului parametrilor delegatului

```
delegate int AddInt(int x, int y);  
0 references  
class Program  
{  
    0 references  
    static void Main(string[] args)  
    {  
        AddInt add = (int x, int y) => { return x + y; };  
  
        Console.WriteLine(add(2, 3));  
        //5  
  
        AddInt add1 = (x, y) => { return x + y; };  
  
        Console.WriteLine(add1(2, 3));  
        //5  
    }  
}
```

Lambda

- Lambda body
 - Pentru ca lambda-ul contine o singura instructiune, nu mai este necesar block-ul *{return x+y;}*
 - Block-ul este inlocuit de expresia *x+y*
 - Return –ul dispare
 - Tipul returnat de lambda
 - Este dat de rezultatul instructiunii (int)
 - Trebuie sa fie compatibil cu delegatul

```
delegate int AddInt(int x, int y);  
0 references  
class Program  
{  
    0 references  
    static void Main(string[] args)  
    {  
        AddInt add = (int x, int y) => { return x + y; };  
  
        Console.WriteLine(add(2, 3));  
        //5  
  
        AddInt add1 = (x, y) => { return x + y; };  
  
        Console.WriteLine(add1(2, 3));  
        //5  
  
        AddInt add2 = (x, y) => x + y;  
  
        Console.WriteLine(add2(2, 3));  
        //5  
    }  
}
```

Lambda

- Delegate-ul poate fi înlocuit cu func!
 - Input, input, output

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Func<int,int,int> add = (x, y) => x + y;

        Console.WriteLine(add(2, 3));
        //5
    }
}
```

Lambda

- Cand body-ul lambda-ului returneaza void, functia lambda poate fi persistata intr-o variabila de tipul **Action**

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Action<int, int> addAndShow = (x, y) => Console.WriteLine(x + y);
        addAndShow(2, 3);
        //5
    }
}
```

Lambda

- Operatorul lambda poate fi folosit la definirea metodelor

- Return type int
- Return type void

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

0 references

```
public int AddAsLambda(int x, int y) => x + y;
```

0 references

```
public void AddAndShow(int x, int y) => Console.WriteLine(x + y);
```

0 references

```
public double Pi { get => 3.1416; }
```

- Operatorul lambda poate fi folosit la definirea getter-ilor

- Good code ☺

List.ForEach

List.ForEach

- ForEach sub forma unei functii
- Primeste ca parametru un action
- Action-ul reprezinta corpul foreach-ului classic
 - Nu exista conceptul de break

```
var students = new List<Student>()
{
    new Student { LastName = "Popa", Age = 16, FirstName = "Florin" },
    new Student { LastName = "Pop", Age = 22, FirstName = "Maria" },
    new Student { LastName = "Popescu", Age = 22, FirstName = "Ion" },
    new Student { LastName = "Chitac", Age = 23, FirstName = "Chitac" },
    new Student { LastName = "Marin", Age = 35, FirstName = "Marin" },
    new Student { LastName = "Popa", Age = 16, FirstName = "Dumitru" },
    new Student { LastName = "Popa", Age = 16, FirstName = "Agamemnon" }
};
```

```
students.ForEach()
```

```
void List<Student>.ForEach(Action<Student> action)
```

Performs the specified action on each element of the List<T>.

action: The Action<in T> delegate to perform on each element of the List<T>.

```
students.ForEach(s => Console.WriteLine(s));
//va afisa toti studentii
```


List.ForEach

- Poate contine un block de instructiuni
 - Ca orice Action
- Parametru : lambda function
- Continue- poate fi simulat prin return;
 - Foreach va trece la urmatoarea entitate din lista

```
var students = new List<Student>()
{
    new Student { LastName = "Popa", Age = 16, FirstName = "Florin" },
    new Student { LastName = "Pop", Age = 22, FirstName = "Maria" },
    new Student { LastName = "Popescu", Age = 22, FirstName = "Ion" },
    new Student { LastName = "Chitac", Age = 23, FirstName = "Chitac" },
    new Student { LastName = "Marin", Age = 35, FirstName = "Marin" },
    new Student { LastName = "Popa", Age = 16, FirstName = "Dumitru" },
    new Student { LastName = "Popa", Age = 16, FirstName = "Agamemnon" }
};

students.ForEach(s => {
    if (s.Age % 2 == 0)
    {
        return;
    }
    Console.WriteLine(s);
});
//va afisa toti studentii cu varsta impara
```

Collection.FindAll

- [Documentatie](#)
- E specific tuturor colectiilor
- Rezultatul: o colectie continand toate elementele pentru care predicatul va produce un rezultat *true*

```
var students = new List<Student>()
{
    new Student { LastName = "John Doe", Age = 16, FirstName = "USA" },
    new Student { LastName = "Jane Doe", Age = 22, FirstName = "USA" },
    new Student { LastName = "Joe Doe", Age = 22, FirstName = "Germany" },
    new Student { LastName = "Jenna Doe", Age = 23, FirstName = "Germany" },
    new Student { LastName = "James Doe", Age = 35, FirstName = "USA" },
};

List<Student> all22s = students.FindAll(s => s.Age == 22);
```

Alte metode List/Collection

- Documentatie
 - [Microsoft docs](#)
 - Right-click -> go to definition pe clasa List
- List.FindAll
 - Extrage intr-o noua lista toate elementele care respecta o conditie
 - Conditia data ca parametru sub forma unei functii, uzual expresie lambda
- List.FindIndex
 - Determina indexul primului element din lista care respecta o conditie
 - Conditia data ca parametru sub forma unei functii, uzual expresie lambda
- List.FindLastIndex
 - Determina indexul ultimului element care respecta o conditie
- List.Exists
 - Returneaza *true* daca lista contine un element care respecta o conditie

Va multumesc!