

Exceptions in C#

Anomalies in programming

When program is executed **many kind of errors and anomalies may occur**. Anomaly is **unexpected programm behavior**, which is hard or even impossible to foresee.

C# allows a developer to predict an handle such anomalies in a friendly way.

Types of anomalies:

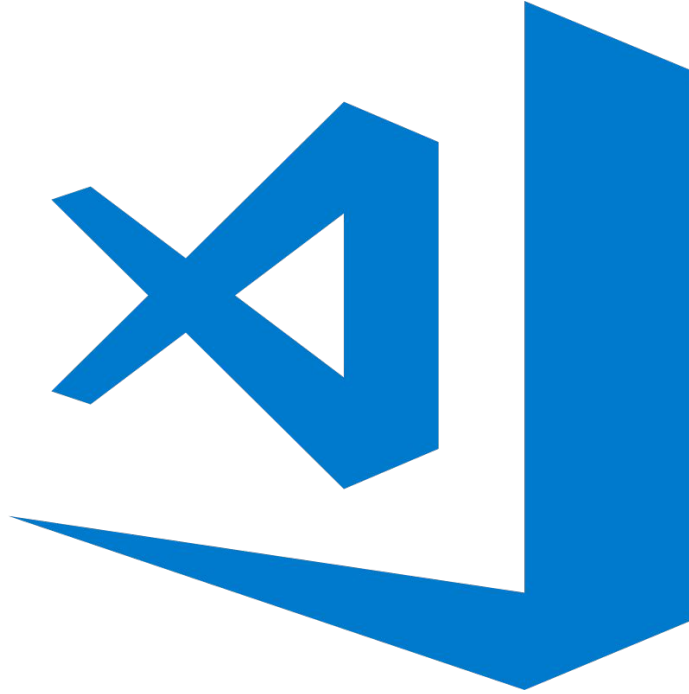
- 1) **Program errors** - happen when **developer writes some “bad”** code, e.g. divides a number by 0 or forgets to free a memory (in low level languages)
- 2) **User errors** - happen when a **user does something “bad”** with a program, e.g. enters invalid string in an input field
- 3) **Exceptions** - happen when something unexpected and unpredictable **happens in program execution**, e.g. tries to connect to DB which does not exist anymore

Exceptions in .NET

In .NET all unexpected or exceptional situations that occur when a program is running are called **Exceptions**.

.NET introduces very convenient, powerful and readable way to handle errors - **Structured Exception Handling**

Exceptions in C#: demo

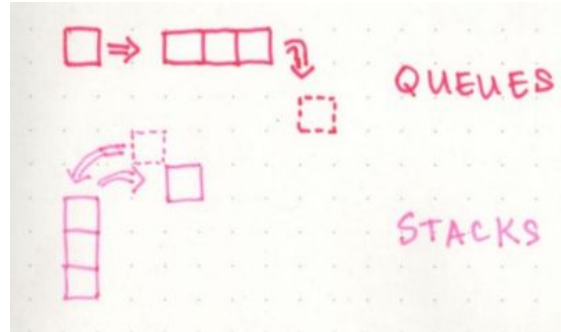
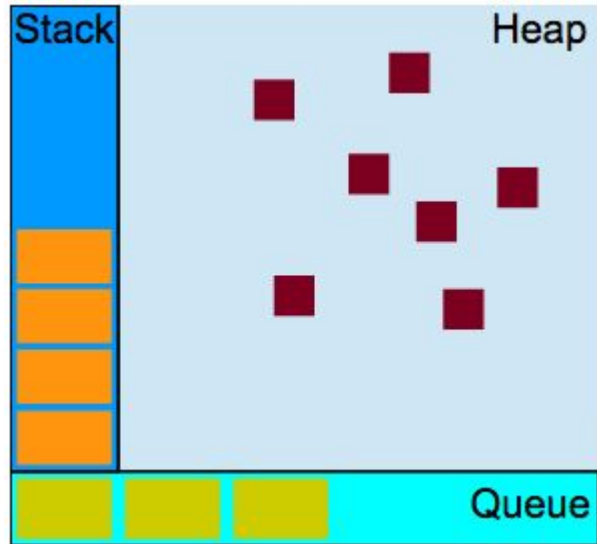


Exception object in C#

- **Data** - dictionary ({key,value} collection) of data associated with an exception. That data provides additional info for developer.
- **HelpLink** - URL for getting help about given exception
- **InnerException** - if exception was caused by another exception, the original exception will be saved in this property
- **Message** - text describing exception
- **Source** - name of the module which caused an exception
- **StackTrace** - list of the method calls that the application was in the middle of when an Exception was thrown.

StackTrace

-



StackTrace

```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}

static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}

static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```

Call Stack

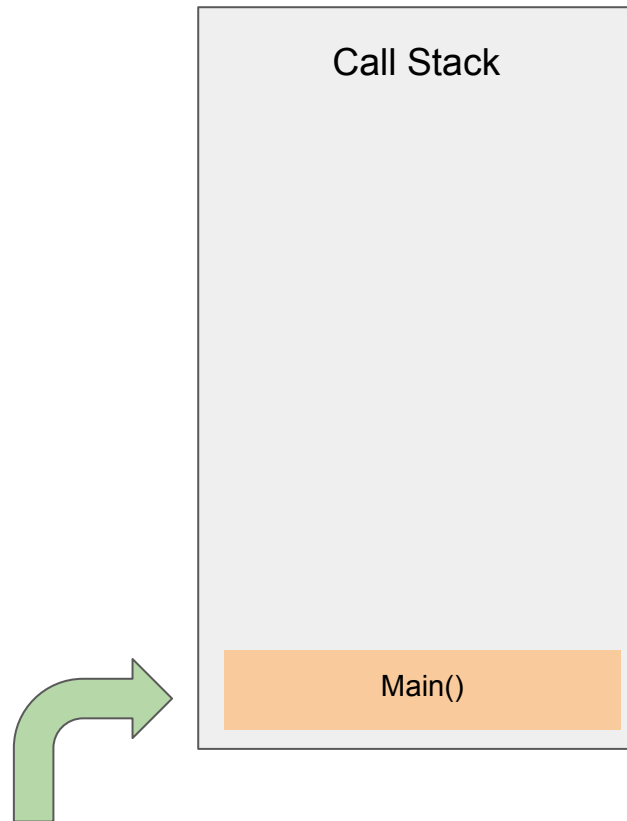
Call Stack

```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}
```

```
static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}
```

→


```
static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```



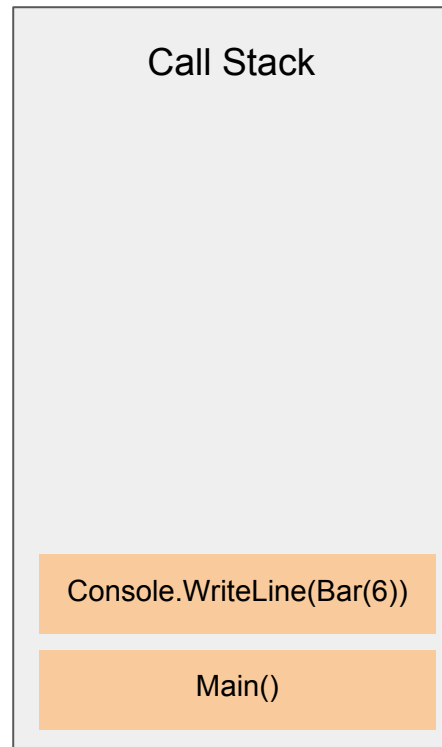
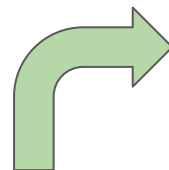
Call Stack

```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}
```

```
static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}
```




```
static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```



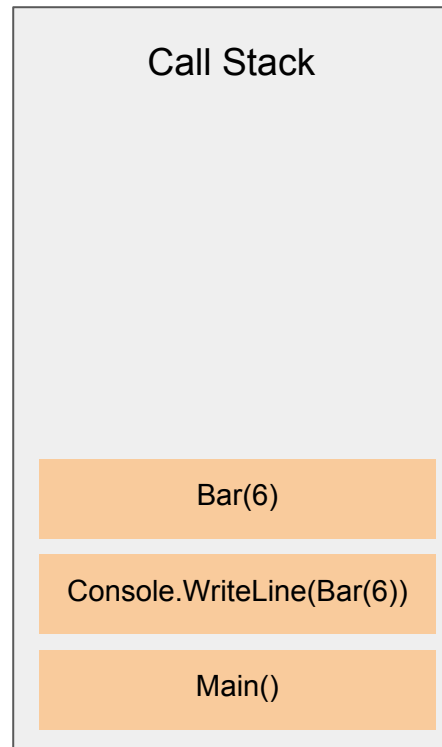
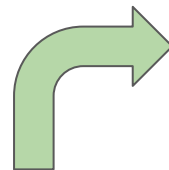
Call Stack

```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}
```




```
static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}
```

```
static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```



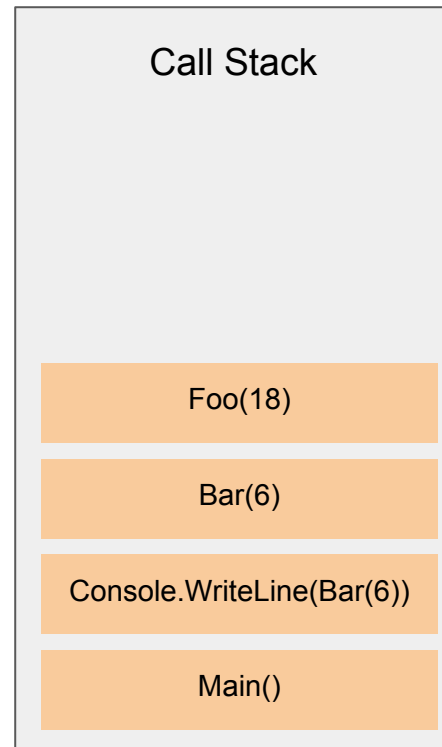
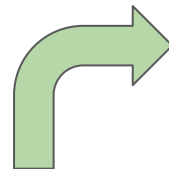
Call Stack

```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}
```




```
static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}
```

```
static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```



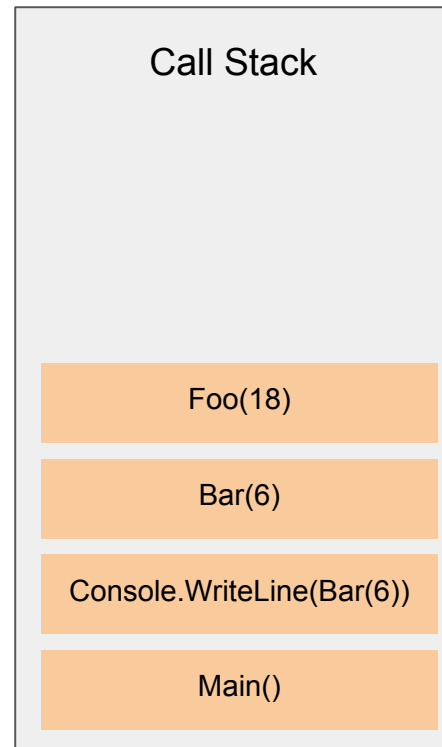
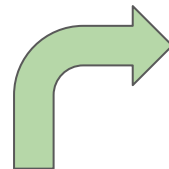
Call Stack




```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}

static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}

static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```



Call Stack

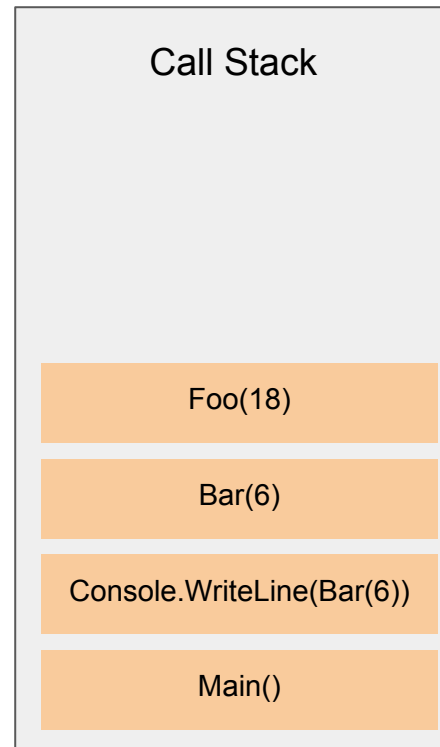
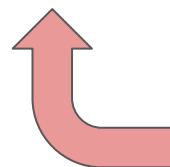


```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}

static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}


static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```

Nothing else to call
return "100" and
pop "Foo()" from the stack



Call Stack

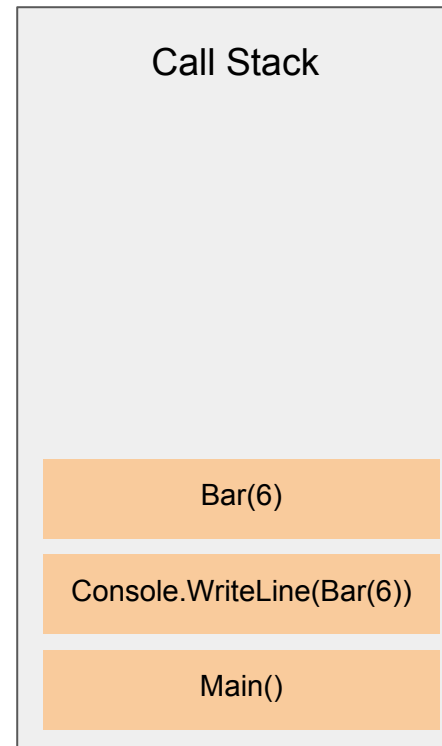
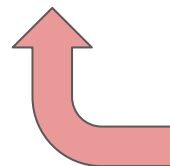
```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}
```



```
static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}
```

```
static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```

Nothing else to call
return "100" and
pop "Bar()" from the stack



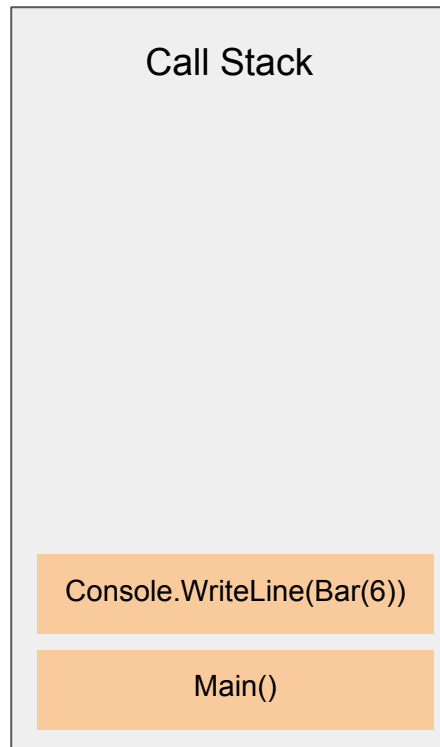
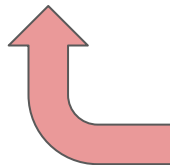
Call Stack

```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}
```

```
static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}
```

```
static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```

Nothing else to call
write "100" to console
pop "Console.WriteLine()" from the stack



Call Stack

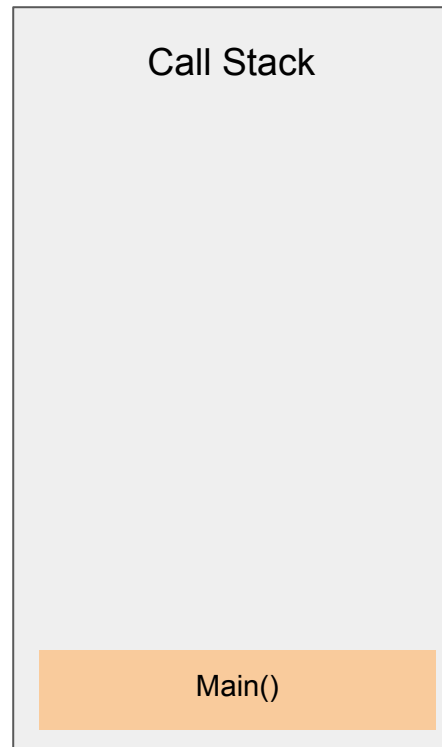
```
static int Foo(int a)
{
    var b = 5;
    return a * b + 10;
}
```

```
static int Bar(int x)
{
    var y = 3;
    return Foo(x * y);
}
```

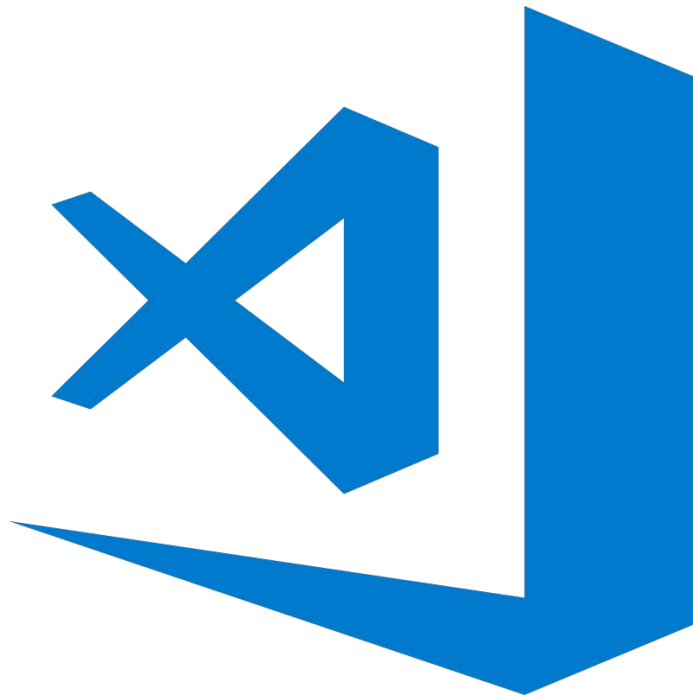
```
static void Main(string[] args)
{
    Console.WriteLine(Bar(6));
}
```



Nothing else to call

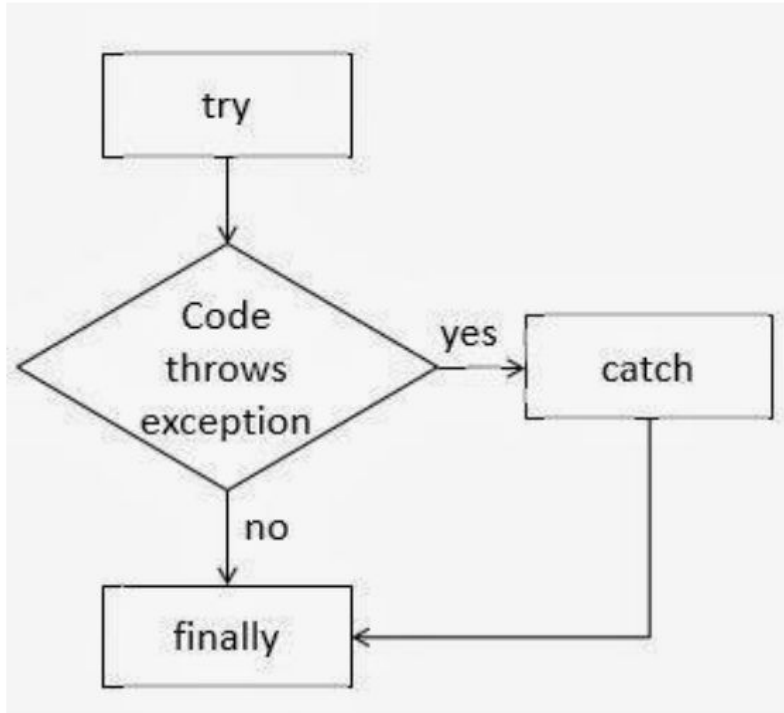


Call stack: demo



Exception handling

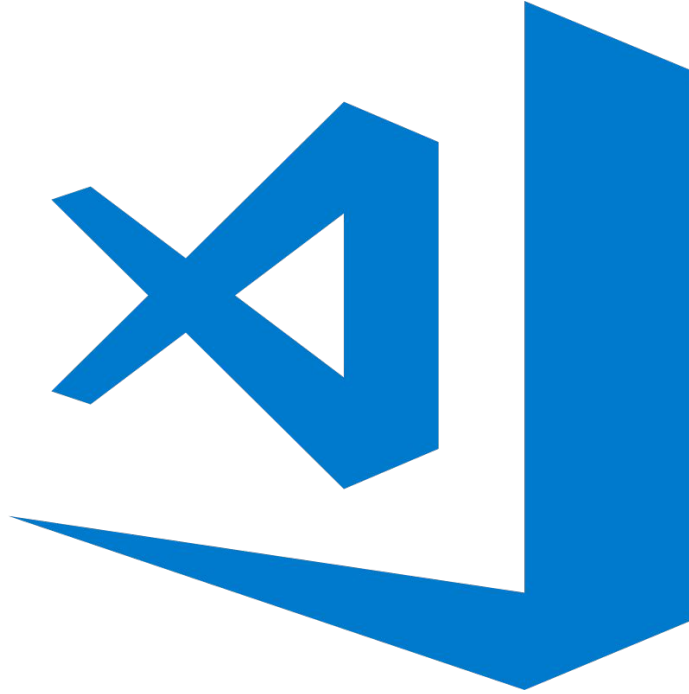
- try
- catch
- finally



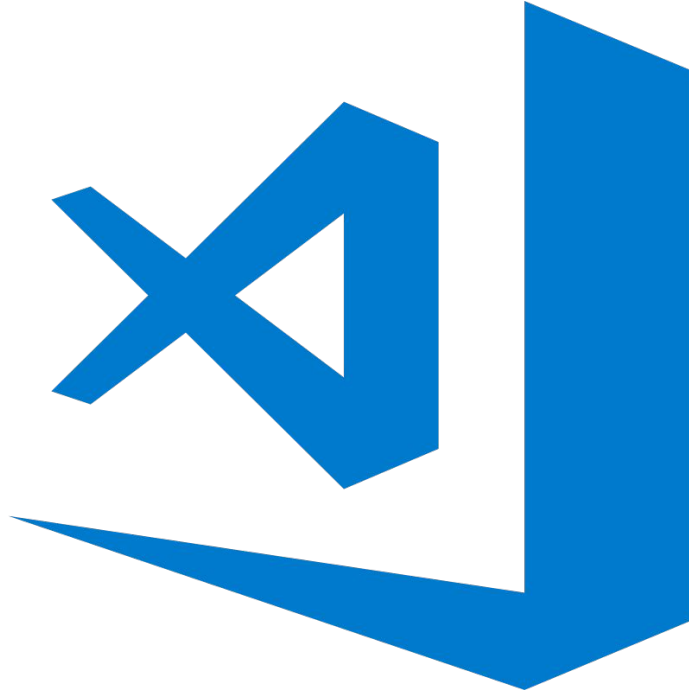
Exception handling

```
try
{
    /* Code block which may cause exception */
}
catch(/* Exception */)
{
    /* Code block which will handle exception */
}
finally
{
    /* Code block which will always execute */
}
```

Exception handling in C#: demo



What will be the output?: demo



What will be the output?

```
static void Main(string[] args)
{
    try {
        Foo();
    } catch (Exception ex) {
        Console.WriteLine("Caught in Main");
    }

    Console.ReadKey();
}

static void Foo()
{
    try {
        Bar();
    } catch (Exception ex) {
        Console.WriteLine("Caught in Foo");
    }
}

static void Bar()
{
    Divide(10, 0);
}
```

What will be the output?

```
static void Main(string[] args)
{
    try {
        Foo();
    } catch (Exception ex) {
        Console.WriteLine("Caught in Main");
    }
}
```

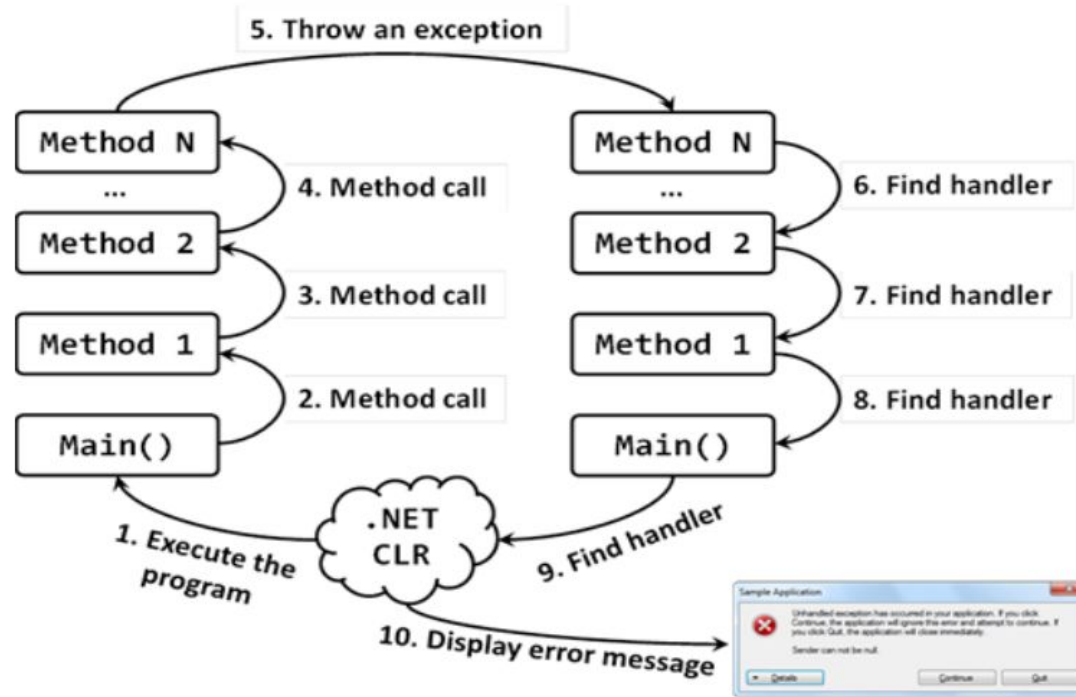
"Caught in Foo"

```
Console.ReadKey();
}
```

```
static void Foo()
{
    try {
        Bar();
    } catch (Exception ex) {
        Console.WriteLine("Caught in Foo");
    }
}
```

```
static void Bar()
{
    Divide(10, 0);
}
```

Method call and exception handling process



System.Exception

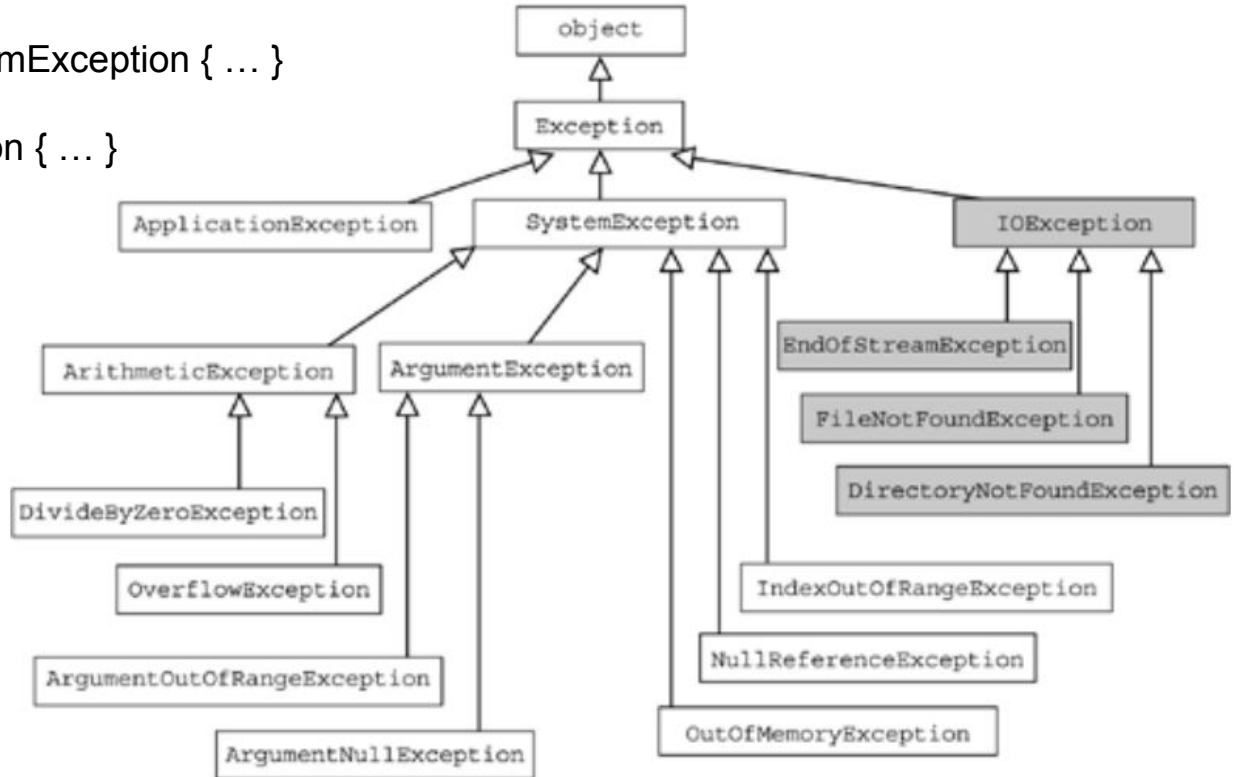
- Base class for all Exceptions in .NET
- All the other exceptions are derived (OOP inheritance) from this class

Built in Exception hierarchy

```
class DivideByZeroException : ArithmeticException { ... }
```

```
class ArithmeticException : SystemException { ... }
```

```
class SystemException : Exception { ... }
```



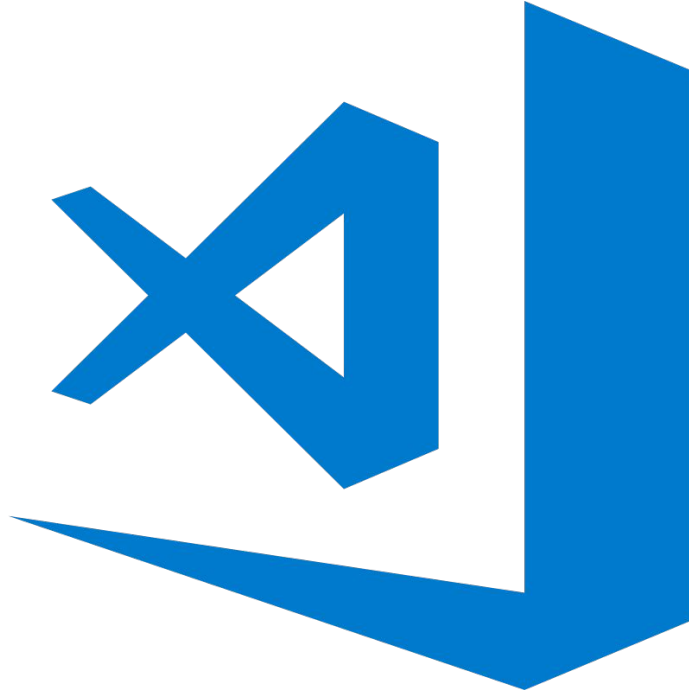
Common Exception class descriptions

Type of exception	Description
System.Exception	The base class for all other exception classes.
System.ApplicationException	The base class for all custom exception classes you create.
System.SystemException	The base class for all predefined exceptions in the System namespace.
System.ArithmeticException	The base class for arithmetic, casting, or conversion operation errors.
System.OverflowException	The exception thrown when there is an overflow in arithmetic, casting, or conversion operation.
System.ArgumentException	The exception thrown when an invalid argument is passed to a method.
System.IndexOutOfRangeException	The exception thrown when an attempt is made to access an array element that is out of the bounds of the array.
System.StackOverflowException	The exception thrown when the stack overflows with too many pending method calls. This can occur if you have a recursive, infinite loop in your code.

Multiple exception handling

- One “try” statement can “catch” multiple Exceptions of different types
- Every “catch” **must handle unique type** of Exception
- “catch” statements are matched from top to bottom one by one, however **only one block executes**. Other blocks are ignored.

Multiple exception handling: demo



Throwing exceptions

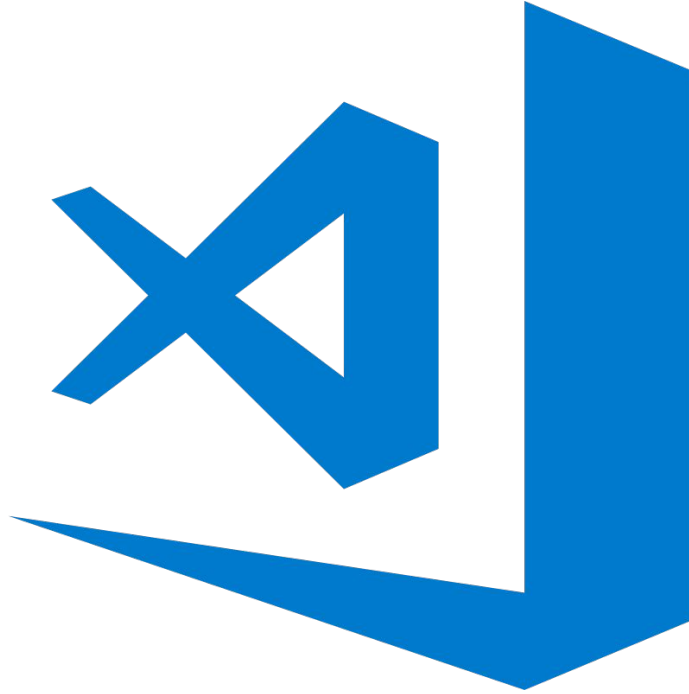
Exceptions might be generated by developer himself.

Generation of exception is called “throwing” (throwing an exception)

Throwing an exception stops execution of the block immediately

```
void Foo()  
{  
    throw new Exception("Some exception message");  
}
```

Throwing exceptions: demo



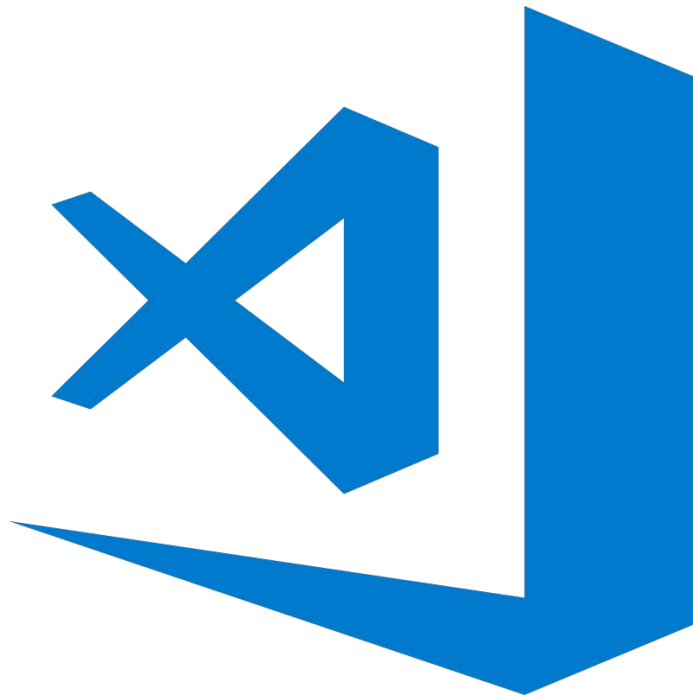
Custom exceptions

Developer can define custom, own made Exceptions

Custom exceptions are thrown and caught the same as built-in

All custom exceptions must be derived from “System.Exception” class or any this class child

Custom exceptions: demo

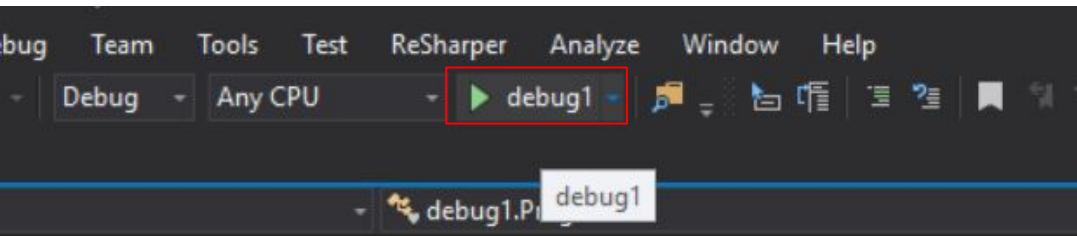
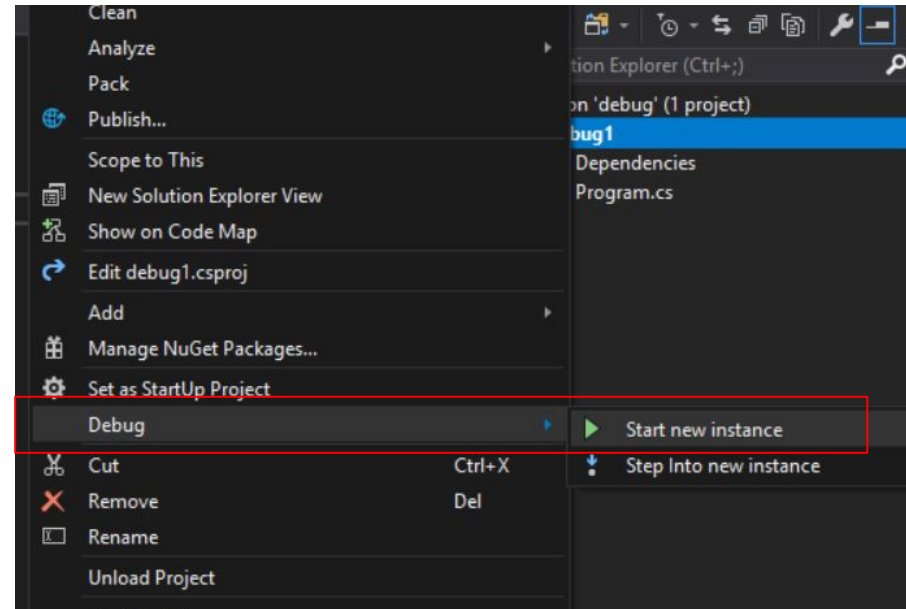
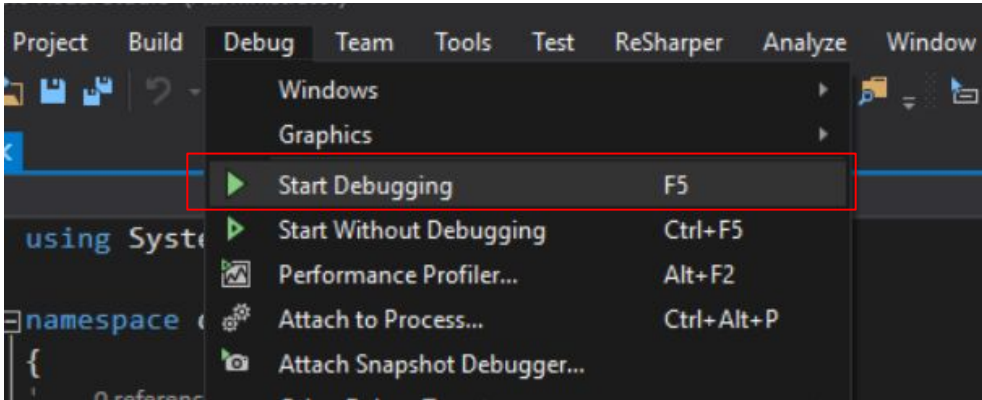


Debugging

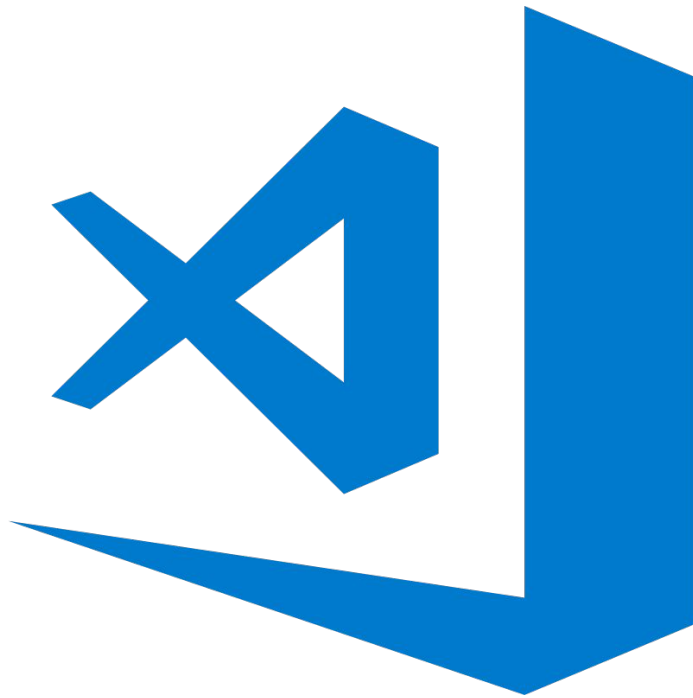
Why?

- Writing code is usually 10% of time. Finding and fixing bugs is 90%
 - To fix bugs first you need to identify defects (why does the bug happens)
 - Finding bugs and fixing them is closely related (hence the name: de-bug)
-
- Debugging also helps you to understand code and logic better
 - Going through the code as it runs makes it easy to visualise the flow

How to start debugging



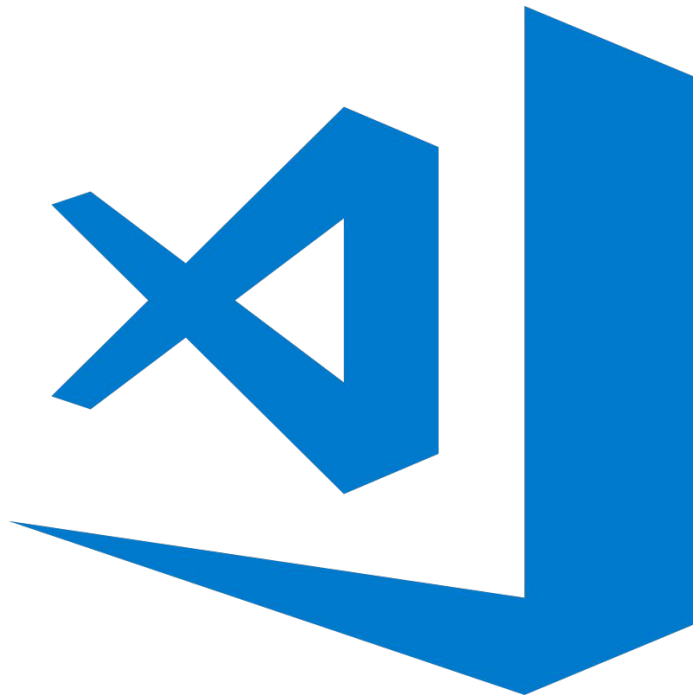
Debugging: demo



Debugging process

- **Step over** - (*F10*) will go to next line, execute it and stop
- **Step into** - (*F11*) will go to next line, if it is function, will go inside, execute and stop
- **Step out** - (*Shift + F11*) if debugger is currently in some function, will execute it until the end and return to the line of code where this function was called
- **Continue** - (*F5*) will continue running your application before it hits another breakpoint

Debugging process: demo



What is available in debug?

- **See** and **edit** current data and values
- Immediate windows to experiment with current data and values
- Call stack
- Diagnostics
- Stats

What is available in debug: demo

