# C# Advance

**Windows Programming Course**

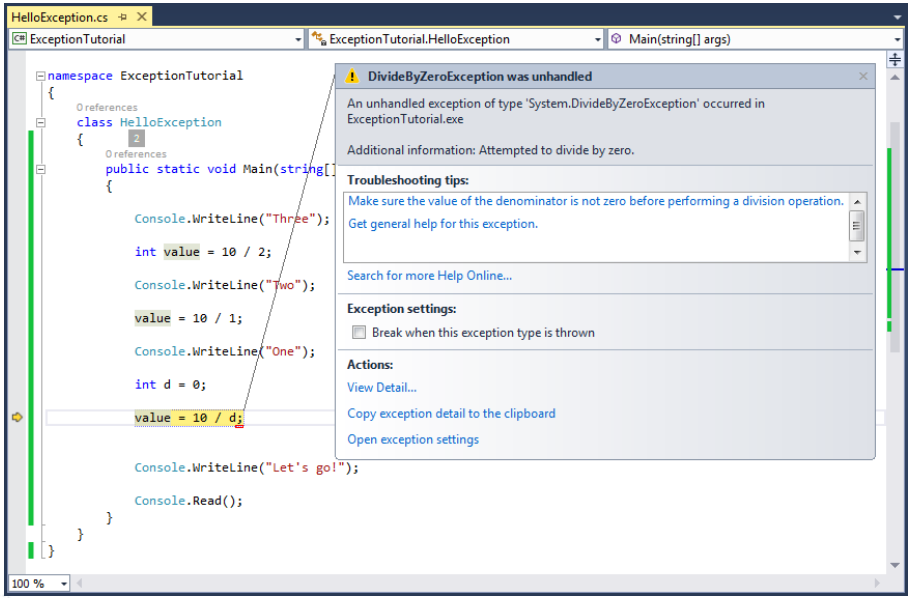
# Agenda

1. Exception Handling
2. Asynchronous Programming

# 1.
# Exception Handling

# Exception

An exception is an event that occurs during the execution of a program that stops the normal flow of instructions.



# Characteristics

When an error occurs within a method

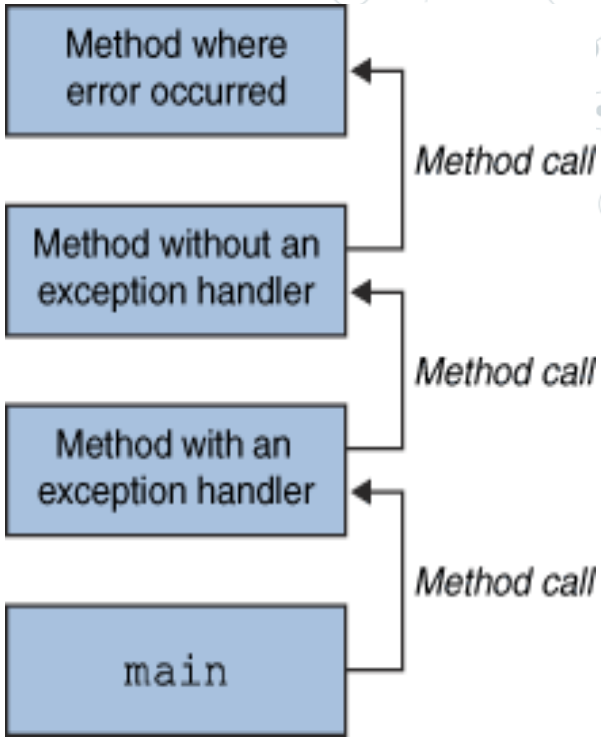
- The method creates an object and hands it off to the runtime system.
- The object, called an exception object, contains information about the error (type and the state of the program when the error occurred).

The process above is called throwing an exception.

# Characteristics (cont.)

After a method throws an exception

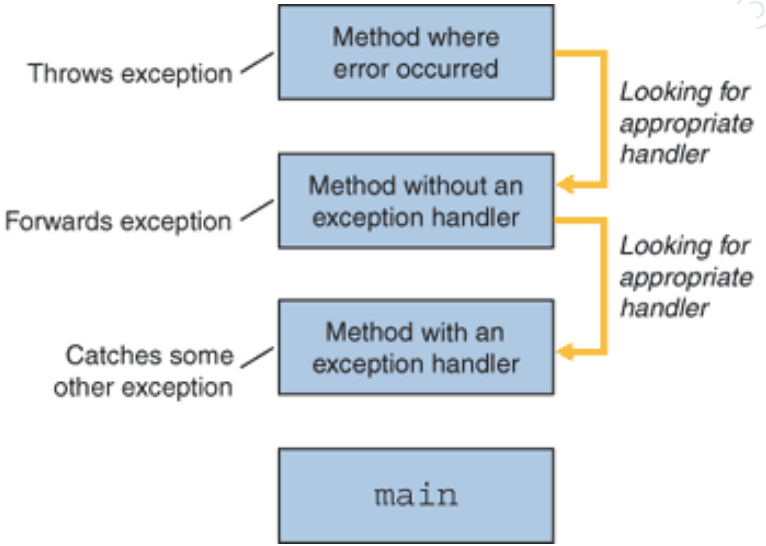
- The runtime system tries to find something to handle it.
- The set of those possible "somethings" is the ordered list of methods that had been called.
- The list of methods is known as the call stack (figure).



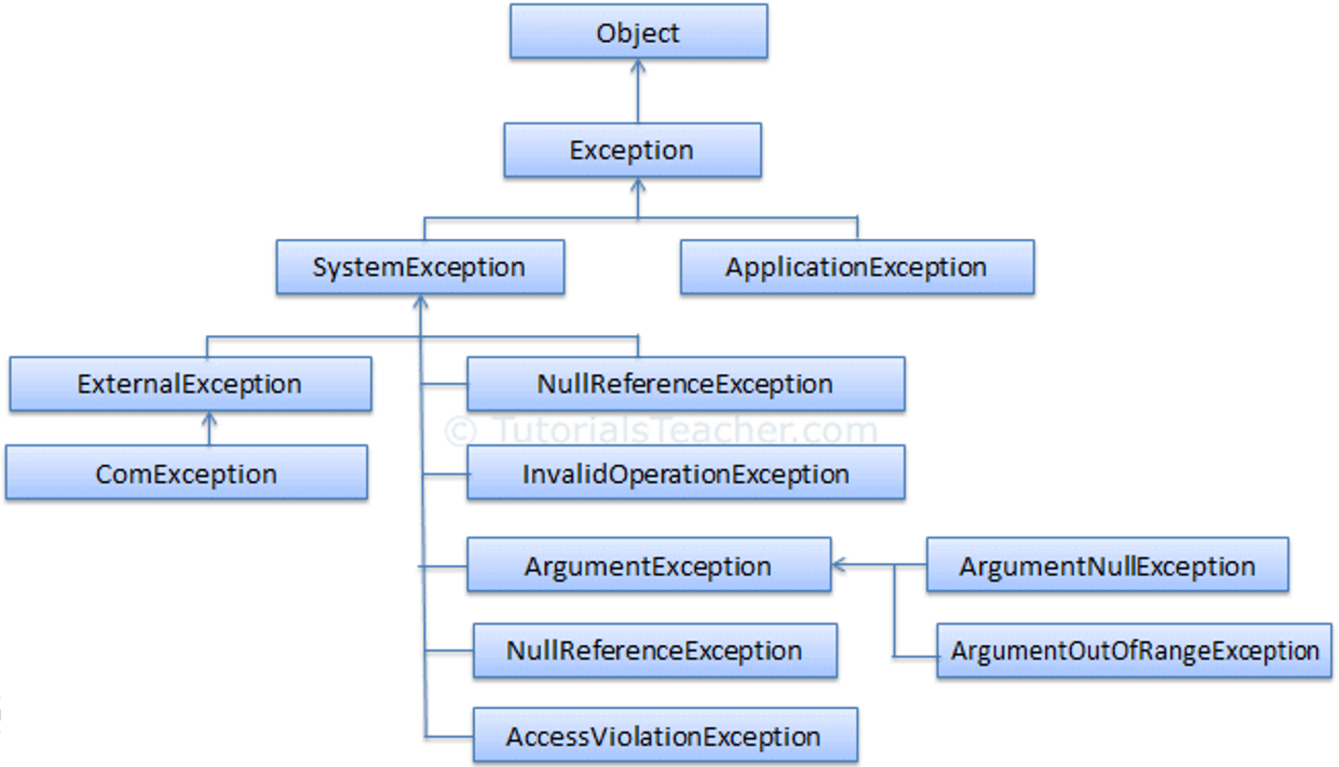
# Characteristics (cont.)

The runtime system searches the call stack for a method that contains a block of code that can handle the exception (exception handler).

- Start with the method in which the error occurred.
- Go backwards in the call stack.
- When an appropriate handler is found, pass the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.
- If no appropriate exception handler is found, the runtime system (and, consequently, the program) terminates.



# Exception Types



# Exception Properties

The **Messaage** property gives brief description of the problem.

The **StackTrace** property is extremely useful when identifying the reason caused the exception.

File names and line numbers are accessible only if the compilation was in Debug mode.

```
Exception caught: Input string was not in a correct format.
   at System.Number.ParseInt32(String s, NumberStyles style,
NumberFormatInfo info)
   at System.Int32.Parse(String s)
   at ExceptionsTest.CauseFormatException() in
c:\consoleapplication1\exceptionstest.cs:line 8
   at ExceptionsTest.Main(String[] args) in
c:\consoleapplication1\exceptionstest.cs:line 15
```

# Catching Exeptions

*catch(ExceptionName e){}* block is an exception handler.

One *try* can have many *catch* blocks.

When catching an exception of a particular class, all its inheritors (child exceptions) are caught too

```
try {
  // Do some work that can raise exception
}
catch (DivideByZeroException ex) {
 // Handle the catch  DivideByZeroException ex
}
catch (Exception ex) {
  // Handle the catch Exception ex
}
finally {
  // This block always executes
}
```

# Throwing Exceptions

Exceptions are `thrown` (raised) by throw keyword in C#.

e.g.:

```
if (amount < 0) {
   throw new ArgumentException("Invalid amount!");
}
```

# 2.
# Asynchronous Programming

# Synchronous Programming

Executing program components sequentially

- Actions happen one after another
- Uses a single thread of a single process

Components wait for previous components to finish

Program resources are accessible at all points

# Synchronous Programming Problems

Problems of Sync Programming

- If one component blocks, entire program blocks
- UI may become unresponsive
- No utilization of multi-core systems
- CPU demainding tasks delay execution of all other tasks
- Accessing resources blocks entire program, especially problematic with web resources

# Synchronous Programming Problems (cont.)

Resource access problems

- Resource may be large
  - While loading, UI blocks
  - Program stops responding
- Resources may be web-based
  - Slow connections mean slow loading
  - Server may hang
- While accessing, sync programs stop all other operations

# Asynchronous Programming

Program components can execute in parallel

- Some actions run alongside other actions
- Each action happens in a separate thread

Independent components don’t wait for each other

Program resource shared between threads

- If one thread uses a resources, other shouldn’t use it.

# Async Progamming Benefits

Benefits of Async programming

- If component blocks, other components still run
  - Until they need a resource from blocked component
- UI runs separately
  - Always responsive
- Ultilization of multi-core systems
  - Each core executes one or several threads

# Async Progamming Difficulties

Hard to imagine which code run at some time

Hard to notify a “component” has executed

- So far, callbacks were the way
- Have ensure callback runs in appropriate context (e.g. same thread it started in)

Hard to protect resources

- One thread uses a resources, others wait for it
  - Hard to synchronize resource access
- Deadlocks and race conditions can occur

# Asynchronous Programming in C#

Thread – smallest, independent piece of a prorgam, executable by OS

- “Subprocess”
- Usually contained inside a process

C# is a multi-threaded language by design

- Process/program can manage multiple thread
- Threads execute in parallel (handle by OS & CPU)
- Thread control mechanisms are avaliable in C#.

# Asynchronous Programming in C# (cont.)

## Event-Based Asynchronous Pattern

```
private static void EventBasedAsyncPattern()
{
  using (var client = new WebClient())
  {
    client.DownloadStringCompleted += (sender, e) =>
    {
      Console.WriteLine(e.Result.Substring(0, 100));
    };
    client.DownloadStringAsync(new Uri(url));
    Console.WriteLine();
  }
}
```

# Asynchronous Programming in C# (cont.)

Task-Based Asynchronous Pattern (modern approach)

```
private static async Task TaskBasedAsyncPatternAsync()
{
  Console.WriteLine(nameof(TaskBasedAsyncPatternAsync));
  using (var client = new WebClient())
  {
    string content = await client.DownloadStringTaskAsync(url);
    Console.WriteLine(content.Substring(0, 100));
    Console.WriteLine();
  }
}
```

## Read more...

- Asynchronous programming with async and await

# Thanks!

## Any questions?

You can find me at:

tranminhphuoc@gmail.com