Exception Handling

Handling Errors During the Program Execution
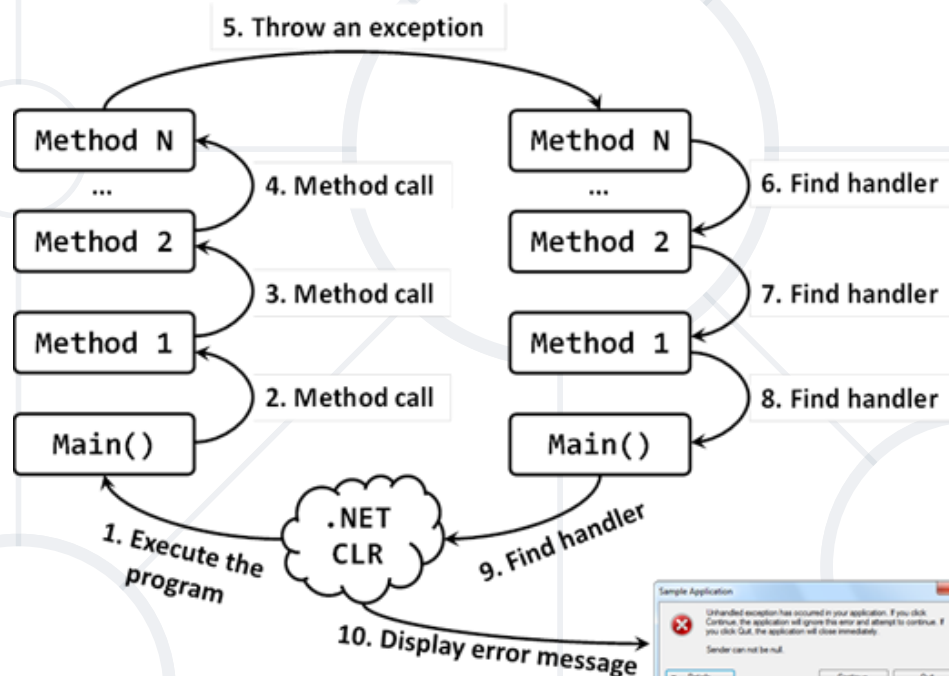
SoftUni Team

Technical Trainers

Software University

Table of Contents

1. What are Exceptions?
   - The **System.Exception** Class
   - Types of Exceptions and their Hierarchy
2. Handling Exceptions
   - **try-catch-finally**
3. Raising (Throwing) Exceptions
   - **throw new Exception(…)**
4. Exceptions: Best Practices

sli.do

#csharp-advanced

The Paradigm of Exceptions in OOP

What Are Exceptions?

What Are Exceptions?

- **Exceptions** handle errors and problems at runtime

- **Throw** an exception to signal about a problem

```
if (size < 0)
    throw new Exception("Size cannot be negative!");
```

- **Catch** an exception to handle the problem

```
try {
    size = int.Parse(text);
} catch (Exception ex) {
    Console.WriteLine("Invalid size!");
}
```
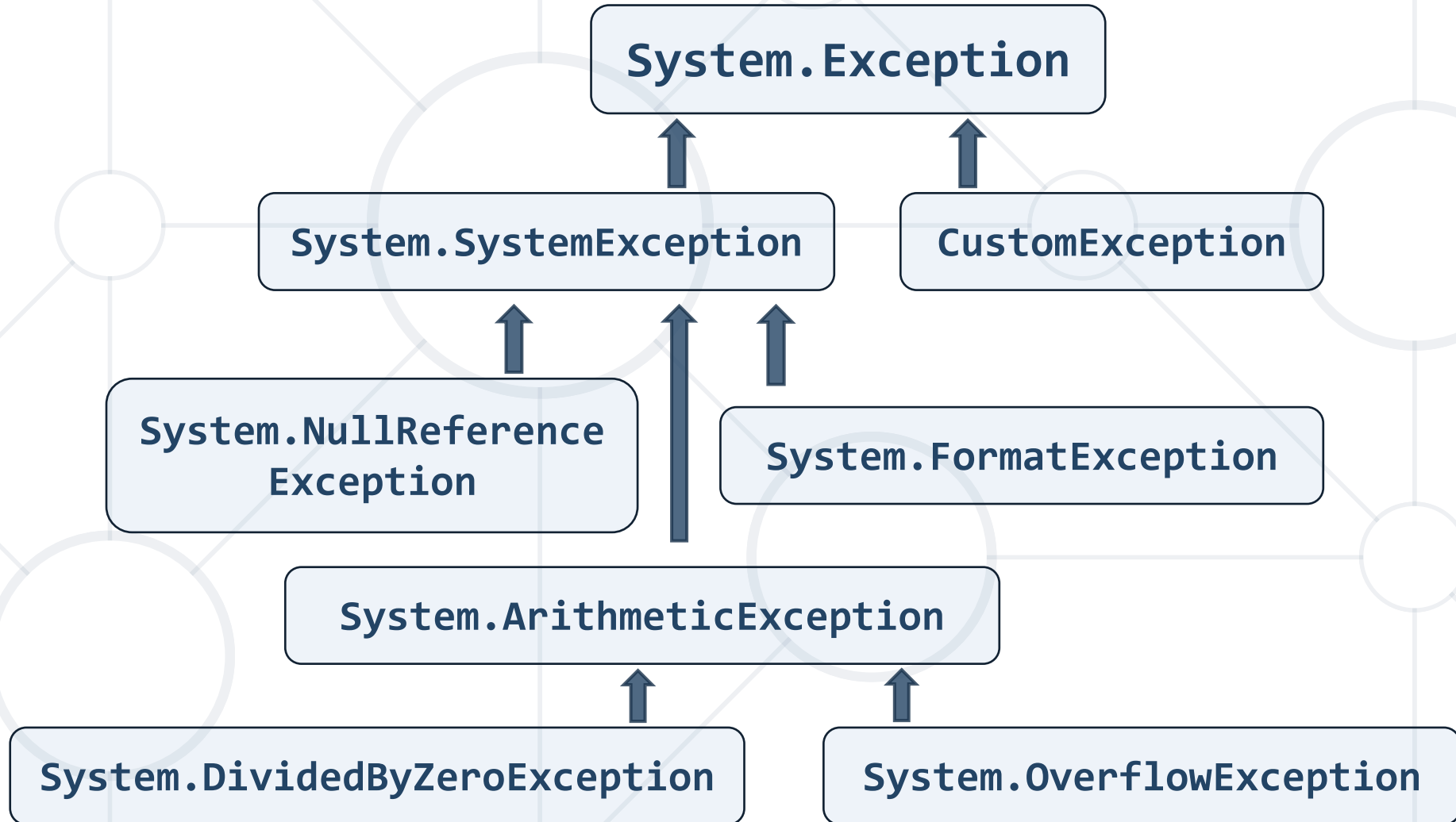
How Do Exceptions Work?

try { **Run this code**

catch { **Execute this code when there is an exception**

finally { **Always run this code**

The `System.Exception` Class

- Exceptions in C# are **objects**

- The **`System.Exception`** class is a base for all exceptions in CLR

  - Holds information about the **error**

  - **`Message`** – a text description of the exception

    - **`StackTrace`** – the snapshot of the stack at the moment of exception throwing

    - **`InnerException`** – exception that caused the current exception (if any)

Exception Hierarchy in .NET

```
System.Exception
```

```
System.SystemException
```

```
CustomException
```

```
System.NullReference
Exception
```

```
System.FormatException
```

```
System.ArithmeticException
```

```
System.DividedByZeroException
```

```
System.OverflowException
```

Catching and Processing Errors

Handling Exceptions

Handling Exceptions

- In C# exceptions can be handled by the **try-catch** construction

```csharp
try {
    // Do some work that can raise an exception
}
catch (SomeException) {
    // Handle the caught exception
}
```

- **catch** blocks can be used multiple times to process different exception types

Multiple Catch Blocks – Example

```csharp
string s = Console.ReadLine();
try {
  int.Parse(s);
  Console.WriteLine(
    "You entered a valid Int32 number {0}.", s);
}
catch (FormatException) {
  Console.WriteLine("Invalid integer number!");
}
catch (OverflowException) {
  Console.WriteLine(
    "The number is too big to fit in Int32!");
}
```

Handling Exceptions

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

```
try {
    // Do some work that can cause an exception
}
catch (ArithmeticException ae) {
    // Handle the caught arithmetic exception
}
```

- Handles **ArithmeticException** and its descendants **DivideByZeroException** and **OverflowException**

Find the Mistake!

```
string str = Console.ReadLine();
try {
    Int32.Parse(str);
}
catch (Exception) {        Should be last
    Console.WriteLine("Cannot parse the number!");
}
catch (FormatException) {        Unreachable code
    Console.WriteLine("Invalid integer number!");
}
catch (OverflowException) {        Unreachable code
    Console.WriteLine("The number is too big to fit in Int32!");
}
```

Handling All Exceptions

- For **handling all exceptions** (disregarding the exception type, even unmanaged) use the construction:

```
try
{
  // Do some work that can raise any exception
}
catch
{
  // Handle the caught exception
}
```

The Try-finally Statement

- The **try-finally** statement ensures the **finally** block is **always executed** (with or without exception):

```
try {
  // Do some work that can cause an exception
}
finally {
  // This block will always execute
}
```

- Used for execution of **cleanup code**, e. g. releasing resources

Try-finally – Example

```
static void TestTryFinally() {
  Console.WriteLine("Code executed before try-finally.");
  try {
    string str = Console.ReadLine();
    int.Parse(str);
    Console.WriteLine("Parsing was successful.");
    return; // Exit from the current method
  } catch (FormatException) {
    Console.WriteLine("Parsing failed!");
  } finally {
    Console.WriteLine("This cleanup code is always executed.");
  }
  Console.WriteLine("This code is after the try-finally block.");
}
```
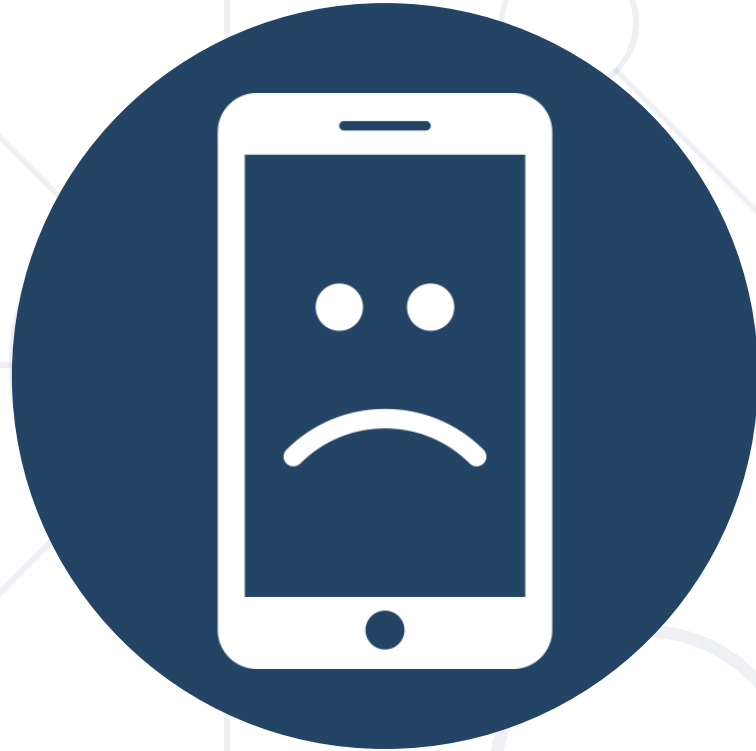
Throwing Exceptions
Using the **"throw"** Keyword

Using Throw Keyword

- **<u>Throwing an exception</u>** with an error message:

```
throw new ArgumentException("Invalid amount!");
```

- Exceptions can accept **message** + **another exception** (cause):

```
try {
    …
}
catch (SqlException sqlEx) {
    throw new InvalidOperationException("Cannot save invoice.",
sqlEx); }
```

- This is called "**chaining**" exceptions

Throwing Exceptions

- Exceptions are thrown (raised) by the **throw** keyword

- Notify the calling code in case of an error or problem

- When an exception is thrown:

  - The program execution stops

  - The exception travels over the stack

    - Until a matching **catch** block is reached to handle it

- Unhandled exceptions display an error message

Re-Throwing Exceptions

- Caught exceptions can be **re-thrown** again:

```
try {
    Int32.Parse(str);
}
catch (FormatException fe) {
    Console.WriteLine("Parse failed!");
    throw fe; // Re-throw the caught exception
}
```

```
catch (FormatException) {
    throw; // Re-throws the last caught exception
}
```

Throwing Exceptions – Example

```
public static double Sqrt(double value) {
   if (value < 0)
      throw new System.ArgumentOutOfRangeException("value",
         "Sqrt for negative numbers is undefined!");
   return Math.Sqrt(value);
}
static void Main() {
   try {
      Sqrt(-1);
   }
   catch (ArgumentOutOfRangeException ex) {
      Console.Error.WriteLine("Error: " + ex.Message);
      throw;
   }
}
```

Best Practices

Best Practices for Exception Handling

Using Catch Block

- **Catch** blocks should:
  - Begin with the exceptions lowest in the hierarchy
  - Continue with the more general exceptions
  - Otherwise, a compilation error will occur
- Each **catch** block should handle only these exceptions which it expects
  - If a method is not competent to handle an exception, it should leave it unhandled
  - Handling all exceptions disregarding their type is a popular **bad practice** (anti-pattern)!

Choosing the Exception Type

- When an invalid parameter value is passed to a method:

    - **ArgumentException**, **ArgumentNullException**, **ArgumentOutOfRangeException**

- When requested operation is not supported

    - **NotSupportedException**

- When a method is still not implemented

    - **NotImplementedException**

- If no suitable standard exception class is available

    - Create own exception class (inherit **Exception**)

Exceptions – Best Practices (1)

- When raising an exception, always pass to the constructor a **good explanation message**

- When throwing an exception always pass a good description of the problem

  - The exception message should explain what causes the problem and how to solve it

    - Good: "*Size should be integer in range [1...15]*"

    - Good: "*Invalid state. First call Initialize()*"

    - Bad: "*Unexpected error*"

    - Bad: "*Invalid argument*"

Exceptions – Best Practices (2)

- Exceptions can decrease the application performance

    - Throw exceptions only in situations which are really exceptional and should be handled

    - Do not throw exceptions in the normal program control flow

    - The .NET runtime could throw exceptions at any time with no way to predict them

        - E. g. **System.OutOfMemoryException**

Creating Custom Exceptions

- Custom exceptions inherit an exception class (e. g. **System.Exception**)

```csharp
public class PrinterException : Exception
{
    public PrinterException(string msg)
        : base(msg) { … }
}
```

- Thrown just like any other exception

```csharp
throw new PrinterException("Printer is out of paper!");
```

Summary

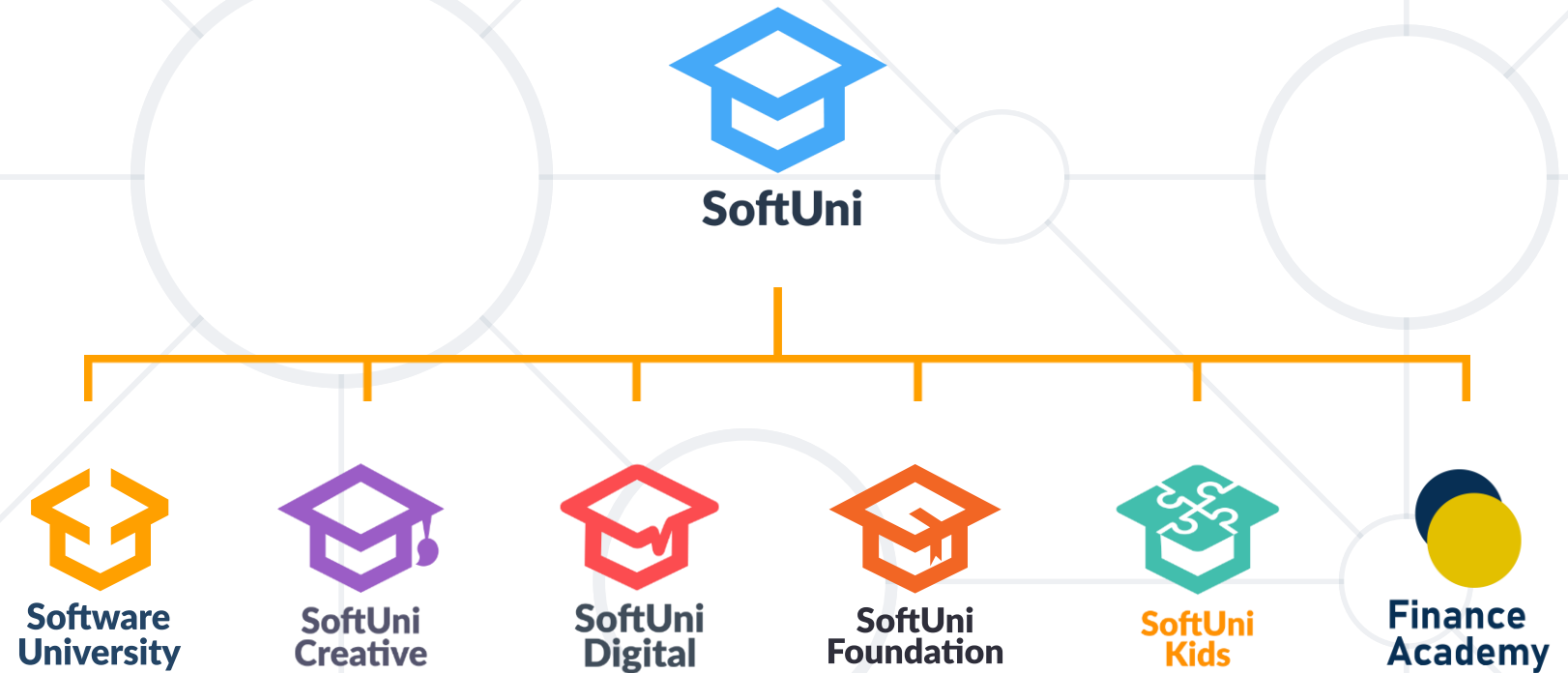- Exceptions provide a **flexible** error handling mechanism

```
throw new Exception("Invalid size!");
```

- **Try-catch** allows exceptions to be handled

```
try { … } catch (Exception ex) { … }
```

- Unhandled exceptions cause error messages

- **Try-finally** ensures a given code block is always executed

Questions?

SoftUni Diamond Partners

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**

- Unauthorized copy, reproduction or use is illegal

- © SoftUni – https://about.softuni.bg

- © Software University – https://softuni.bg