OBJECT-ORIENTED PROGRAMMING WITH C#

EXCEPTION HANDLING

issntt@nus.edu.sg

Objectives

At the end of this lecture, you should be able to

- Understand exceptions in C#
- Analyze the flow of the programs implementing exception handling
- Recognize some exception types that are commonly thrown by C# runtime system
- Implement custom exceptions
- Implement robust C# programs by handling and generating exceptions

Problem

Consider the following program

```
public static void Main() {
   Console.WriteLine("Please enter a divisor");

   string input = Console.ReadLine();
   int divisor = int.Parse(input);

   int quotient = 10 / divisor;
   Console.WriteLine(quotient);
}
```

What issues may it have?

How can we improve?

Topics

- **Runtime Errors and Exceptions**
- Handling exceptions
- Exception class hierarchy
- finally block
- Throwing exceptions
- Defining custom exceptions
- Exception propagation

Scenario 1

We ask and use an integer input from users

What may go wrong?

Scenario 2

We implement writing some text to a file

What may go wrong?

Scenario 3

We implement downloading a picture from the Internet

What may go wrong?

Runtime vs Compile-time Errors

Errors are **commonplace** in programs and there are 2 types of them

Compile-time errors	Runtime errors
• Occur during **compilation**	• Occur during **program running**
• **Always have to be rectified** before programs can run	• **Hard to be predicted** with much certainty

If given a choice, which one do you prefer?

What are Exceptions?

Let’s say that some **runtime errors** happen when a program is running. For example:

- **Hardware errors**, e.g. a hard disk crash
- **Programming errors**, e.g. dividing by zero, assigning too many items to an array…
- …

What should the C# runtime system do?

Question

Consider the following program

```
public static void Main() {
   Console.WriteLine("Please enter a divisor");

   string input = Console.ReadLine();
   int divisor = int.Parse(input);

   int quotient = 10 / divisor;
   Console.WriteLine(quotient);
}
```

Can you guess?

What will the C# runtime do if `divisor` is 0?

Will the next line is executed?

Why?

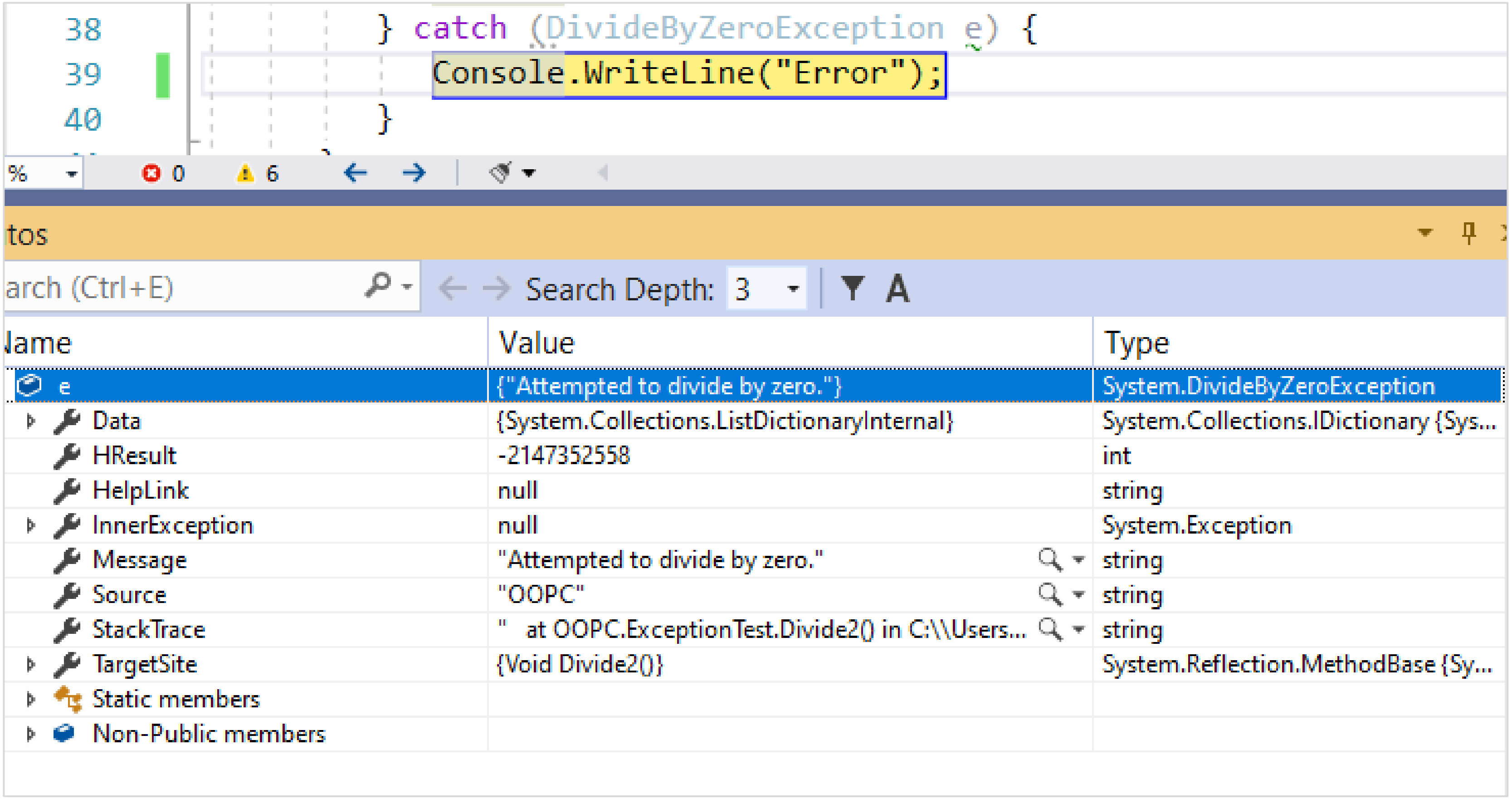
What are Exceptions?

- C# exception handling is **object-oriented**
- When a runtime error occurs within a method, the C# runtime generates (**throws**) an **Exception object**
- And **embed** the **information** about the exception, such as
  - The **exception type**
  - The **location** where exception occurs
  - The current **state of the program**
- C# runtime **stops** executing the subsequent unhandled lines of code

Why such information is embedded into the exception object?

What are Exceptions?

An exception is an **object** that **represents** a **run-time error**



```
} catch (DivideByZeroException e) {
    Console.WriteLine("Error");
}
```

tos

arch (Ctrl+E) Search Depth: 3

lame	Value	Type
e	{"Attempted to divide by zero."}	System.DivideByZeroException
Data	{System.Collections.ListDictionaryInternal}	System.Collections.IDictionary {Sys...
HResult	-2147352558	int
HelpLink	null	string
InnerException	null	System.Exception
Message	"Attempted to divide by zero."	string
Source	"OOPC"	string
StackTrace	" at OOPC.ExceptionTest.Divide2() in C:\\Users...	string
TargetSite	{Void Divide2()}	System.Reflection.MethodBase {Sy...
Static members		
Non-Public members		

Topics

- Runtime Errors and Exceptions
- **Handling exceptions**
  - Rationale
  - try-catch blocks
- Exception class hierarchy
- finally block
- Throwing exceptions
- Defining custom exceptions
- Exception propagation

Question

The exception object is thrown to the **running program**. As the developers of the program, what should we do?

A) Force C# runtime to **continue** as if there is nothing special happens

B) Provide **alternative codes** for the program to execute

C) Make the program **stop**

Exception Handling

The **running program** can **deal with the exception** in one of the following ways

Option A	*Option B1*	*Option B2*
Ignore it	**Handle** it **where it occurs**	**Pass it on** and **handle** it **in another place** in the program

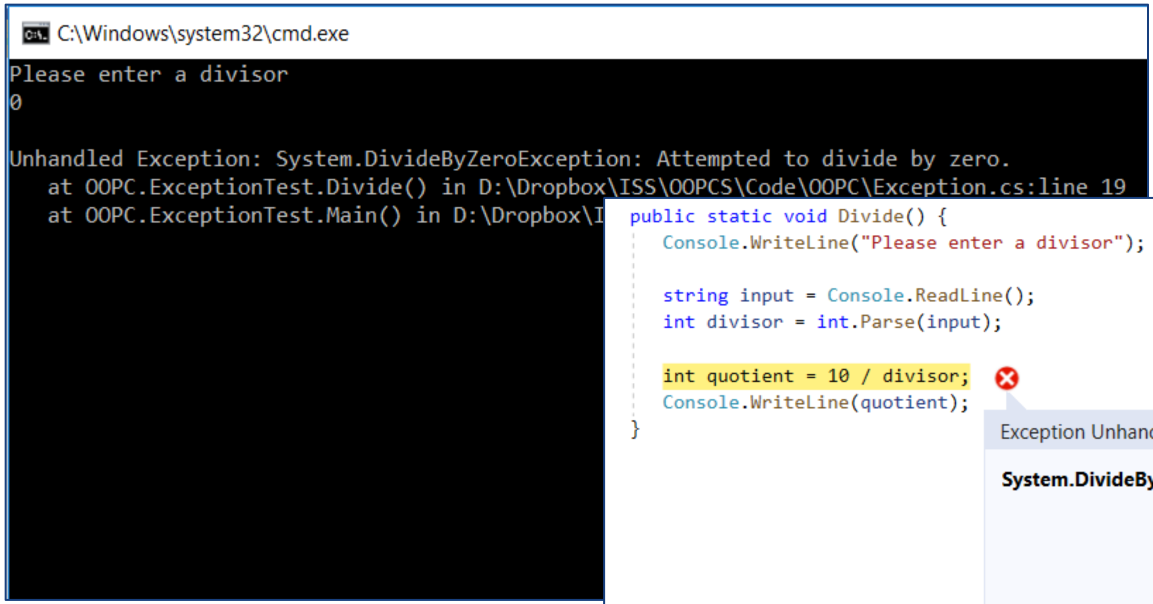
The manner to process an exception is an important **design consideration**

Topics

- Runtime Errors and Exceptions
- **Handling exceptions**
  - **Rationale**
  - try-catch blocks
- Exception class hierarchy
- finally block
- Throwing exceptions
- Defining custom exceptions
- Exception propagation

Why Handling Exceptions?

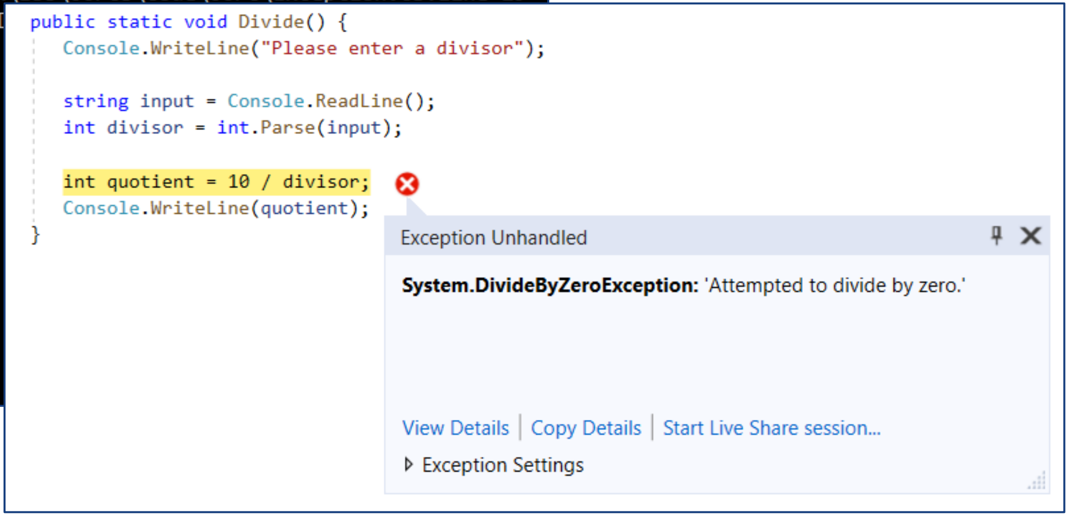
If **ignored** (*option A*), the uncaught exception will be handled by the runtime's **default-exception handler**



C:\Windows\system32\cmd.exe

```
Please enter a divisor
0

Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
   at OOPC.ExceptionTest.Divide() in D:\Dropbox\ISS\OOPCS\Code\OOPC\Exception.cs:line 19
   at OOPC.ExceptionTest.Main() in D:\Dropbox\I
```



```
public static void Divide() {
    Console.WriteLine("Please enter a divisor");

    string input = Console.ReadLine();
    int divisor = int.Parse(input);

    int quotient = 10 / divisor;
    Console.WriteLine(quotient);
}
```

Exception Unhandled

System.DivideByZeroException: 'Attempted to divide by zero.'

So, just ignore and let the **default exception handler** handles **all exceptions** for us. Is it alright?

Topics

- Runtime Errors and Exceptions
- **Handling exceptions**
  - Rationale
  - ***try-catch* block**
- Exception class hierarchy
- finally block
- Throwing exceptions
- Defining custom exceptions
- Exception propagation

`try-catch` Block

To handle an exception in a more **user - friendly** way, we need to **catch** the exception

Image by StartupStockPhotos from Pixabay

The users must be **happy!!!**

`try-catch` Block

1. The `try` block **contains statements** that may **throw exceptions**
2. An exception thrown inside the `try` block can be **caught** and **handled** in the `catch` block
3. The exception object ***e*** with all the **information** is provided by C# runtime system

```
try
{
  1 // Main logic code
}
catch (3 Exception_Type e)
{
  2 // Exception handling
    // code
}
// execution continues
```

Be aware of **which statements** may **throw exceptions**

Example

This example shows how to handle the exception when **accessing** an **element outside the bound** of an array

```
public static void AccessInvalidElement(int i)
{
    try
    {
        int[] myNumbers = { 1, 2, 3 };
      1 int number = myNumbers[i];
        Console.WriteLine(number);
    }
    catch (3 IndexOutOfRangeException e)
    {
      2 Console.WriteLine(e.Message);
    }
}
```

```
Index was outside the bounds of the array.
```

Quiz

Using `try-catch` block to improve the robustness of the following program when **user input is 0**

```
public static void Divide() {
   Console.WriteLine("Please enter a divisor");

   string input = Console.ReadLine();
   int divisor = int.Parse(input);

   int quotient = 10 / divisor;
   Console.WriteLine(quotient);
}
```

The exception type for dividing by zero is *DivideByZeroException*

Which statements may throw exceptions?

Multiple `catch` Blocks

- **More than one** `catch` blocks can be used to catch **different types** of exceptions
- Upon exception thrown, the **first** `catch` block with a **compatible exception** is **called**

```
try
{
    // Main logic code
}
catch (Exception_Type1 e1)
{
    // Exception handling
}
catch (Exception_Type2 e2)
{
    // Exception handling
}
catch (Exception_Type3 e3)
{
    // Exception handling
}
// execution continues
```

Quiz

Using `try-catch` blocks to improve the robustness of the following program when **user input is 0**, or user input is **not** in a **correct integer format**

```
public static void Divide() {
   Console.WriteLine("Please enter a divisor");

   string input = Console.ReadLine();
   int divisor = int.Parse(input);

   int quotient = 10 / divisor;
   Console.WriteLine(quotient);
}
```

The exception type for incorrect integer format is *`FormatException`*

Which statements may throw exceptions?

`try-catch` Block

- If the statements in a `try` block **executes successfully** without throwing any exceptions,
  - the subsequent `catch` blocks will **not** execute, and
  - the flow continues to the statement **after the last `catch`** block
- Otherwise, an exception is thrown, and program flow will go to the respective `catch` block
- After executing a `catch` block, the flow is **never** returned to the `try` block. Instead,
  - it proceeds on to the statement **after the last `catch`** block, and
  - the program continues thereafter without terminating

Next

DivideByZeroException,
IndexOutOfRangeException,
FormatException,
Exception…

Oh my!

How can we know which Exception type to catch?

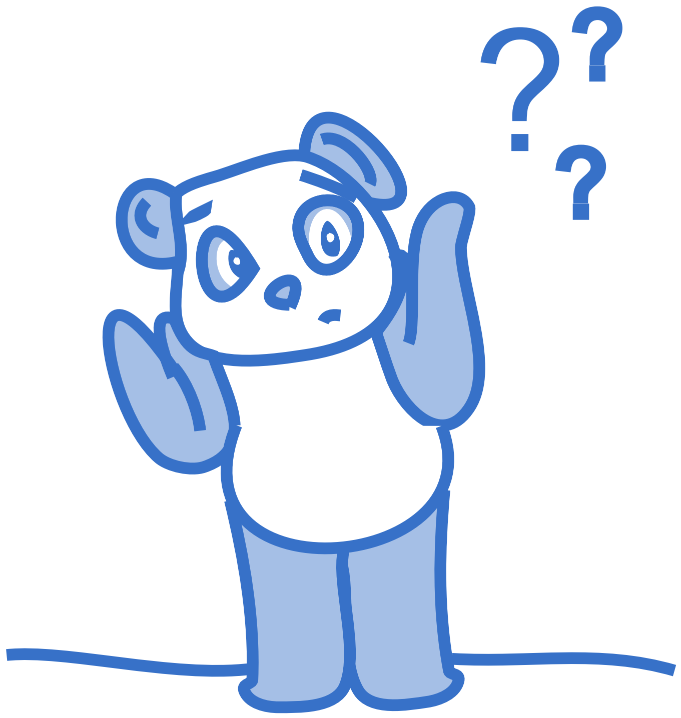


Image by Clker-Free-Vector-Images from Pixabay

Topics

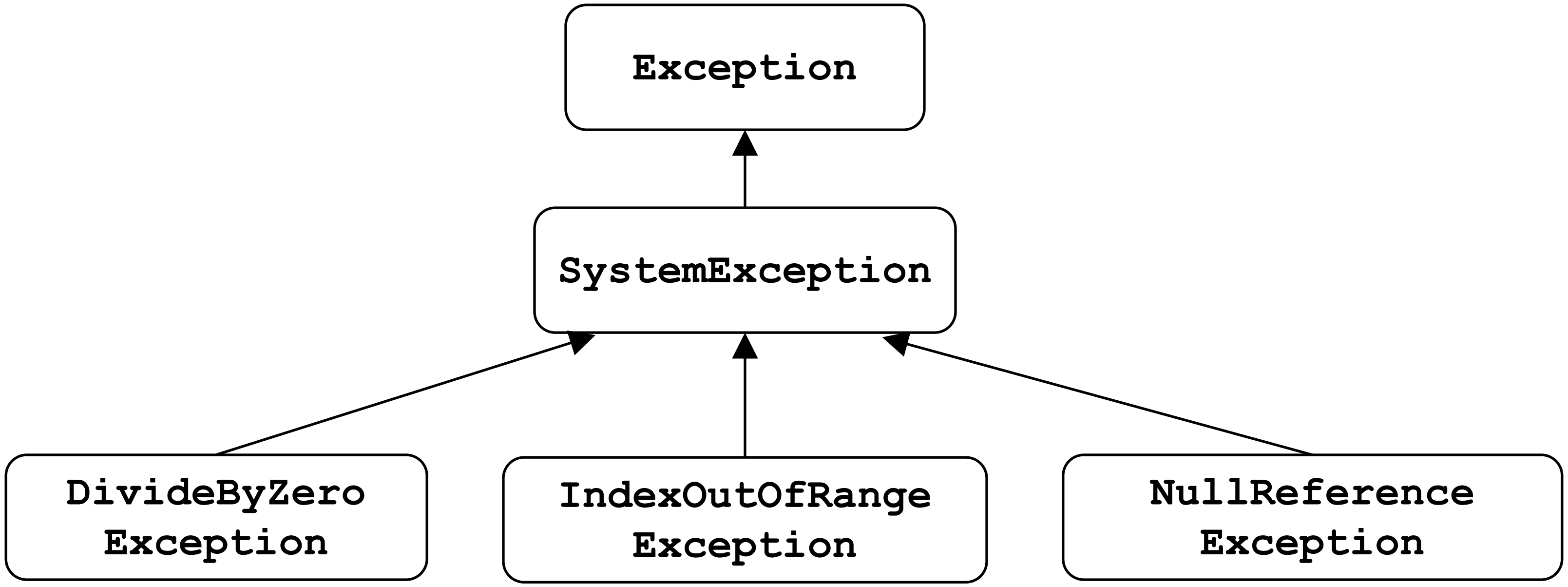
- Runtime Errors and Exceptions
- Handling exceptions
- **Exception class hierarchy**
  - Information in an Exception object
- finally block
- Throwing exceptions
- Defining custom exceptions
- Exception propagation

Exception Class Hierarchy

- An exception thrown can be one of the **pre-defined types** in the .NET Class Library
- The **ultimate ancestor** class in the exception hierarchy is the `Exception` class

Exception Class Hierarchy

Exceptions are **objects**, which are **defined** by **classes**.
The **root class** for exceptions is *`Exception`*



Small part of the Exception class hierarchy

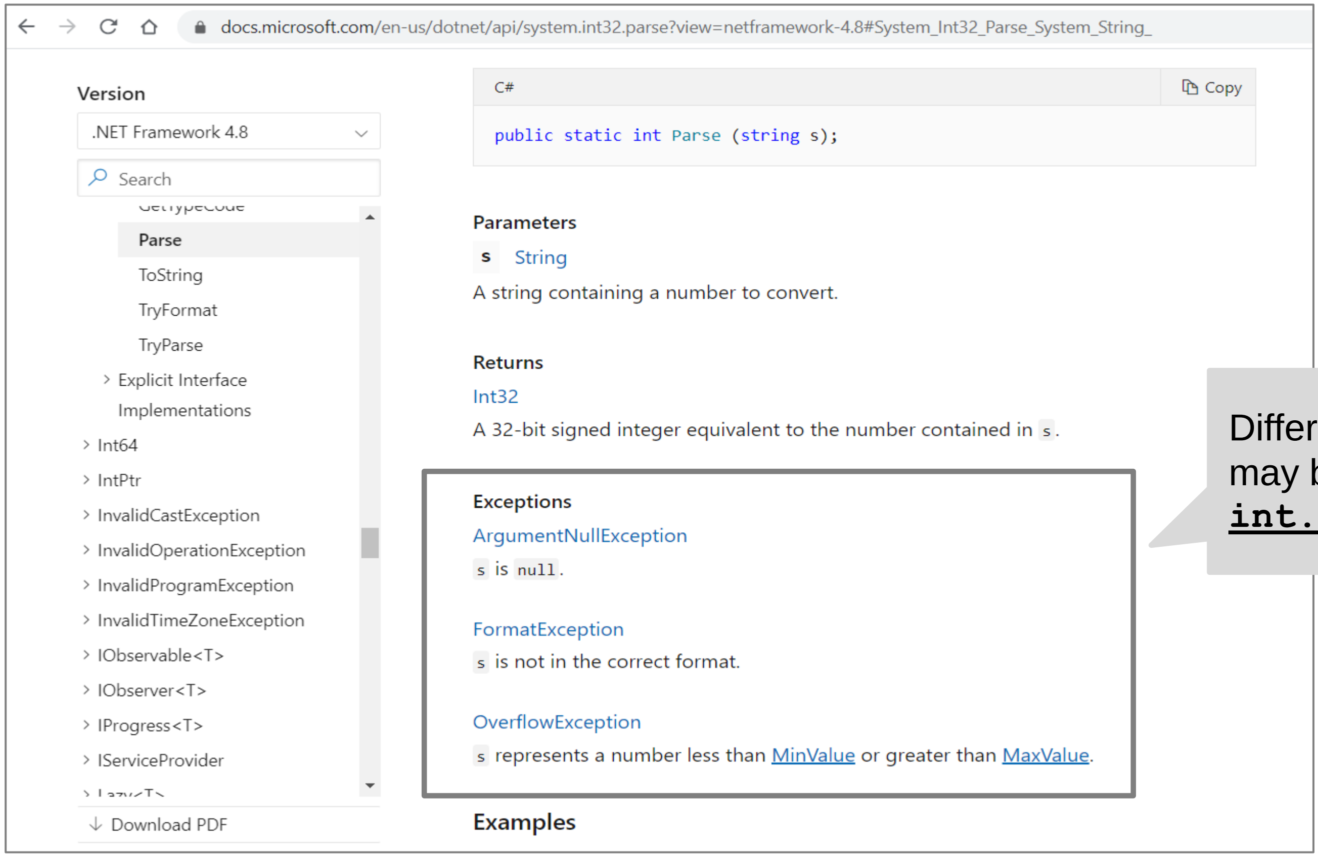
Each derived class is specified to handle a **specific exception**

Quiz What is the output of the program when user input is 0?

```
public static void Divide4() {
   Console.WriteLine("Please enter a divisor");
   try {
      string input = Console.ReadLine();
      int divisor = int.Parse(input);
      int quotient = 10 / divisor;
      Console.WriteLine(quotient);
   }
   catch (FormatException e) {
      Console.WriteLine(
         "Sorry, your input is not in a correct format");
      Console.WriteLine(e);
   }
   catch (IndexOutOfRangeException e) {
      Console.WriteLine("Sorry, unable to divide by 0");
   }
   Console.WriteLine("End of program");
}
```

Exception Class Hierarchy

Possible exceptions to be thrown can be found in the **.NET documentation**


docs.microsoft.com/en-us/dotnet/api/system.int32.parse?view=netframework-4.8#System_Int32_Parse_System_String_

Version

.NET Framework 4.8

Search

GetTypeCode
Parse
ToString
TryFormat
TryParse
Explicit Interface Implementations
Int64
IntPtr
InvalidCastException
InvalidOperationException
InvalidProgramException
InvalidTimeZoneException
IObservable<T>
IObserver<T>
IProgress<T>
IServiceProvider
Lazy<T>

Download PDF

C# Copy

```
public static int Parse (string s);
```

Parameters

s String

A string containing a number to convert.

Returns

Int32

A 32-bit signed integer equivalent to the number contained in s.

Exceptions

ArgumentNullException

s is null.

FormatException

s is not in the correct format.

OverflowException

s represents a number less than MinValue or greater than MaxValue.

Examples

Different exceptions that may be thrown for **`int.parse(string))`**

Exception Class Hierarchy

Catching a parent class exception can catch **all exceptions of its derived classes** (Why?)

- So, if we put `Exception` as **exception type** of the `catch` block, the `DivideByZeroException` thrown will be caught
- Generally, putting `Exception` in the `catch` block will **capture all exceptions** thrown

Question

So, for **all programs**, why don't we just put *`Exception`* in the `catch` block to **catch all exceptions**?

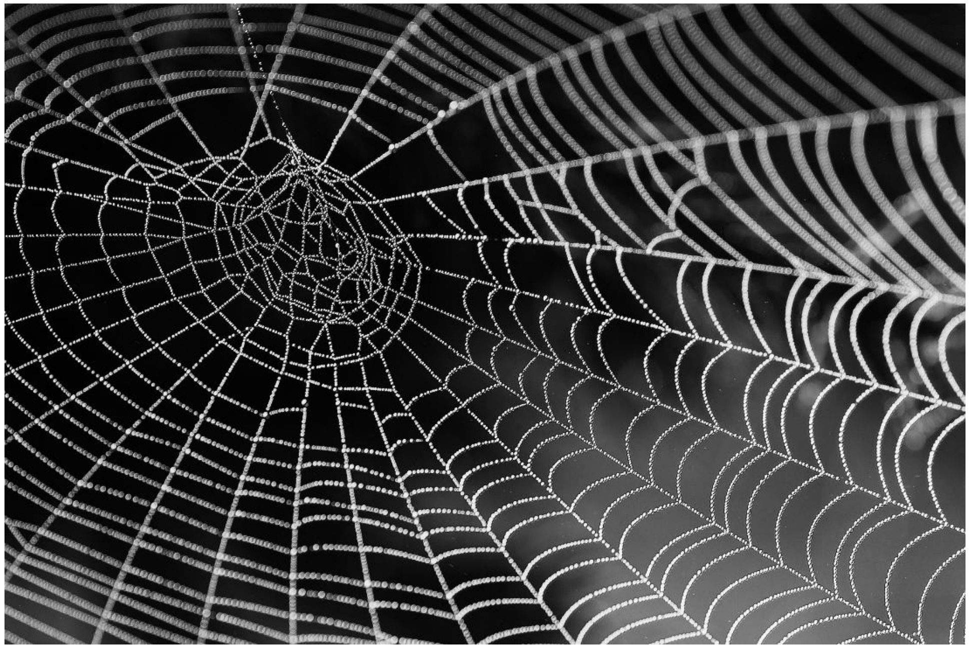


Image by ClaudiaWollesen from Pixabay

Exception Class Hierarchy

We can, but we may

- **Lose** some specific information
- Want to have **different ways to handle** different types of exceptions

A **better** practice

- Specify `catch` block for **each exception** that we **can expect**
- Add the **general** *`Exception`* class for **other** exceptions that may occur

Topics

- Runtime Errors and Exceptions
- Handling exceptions
- **Exception class hierarchy**
  - **Information in an Exception object**
- finally block
- Throwing exceptions
- Defining custom exceptions
- Exception propagation

Information in an Exception object

The *Exception* class has various properties and methods to **set and retrieve** common **information** contained in an exception object

Message	Gets a **message** that describes the current exception
Source	Gets or sets the **name** of the **application** or the object that **causes** the **error**
StackTrace	Gets a **string** representation of the **frames** on the **call stack** at the time the current exception was thrown.
TargetSite	Gets the **method** that **throws** the current exception
GetType()	Inherited from Object. Gets the **type** of the current exception instance
ToString()	Overridden. Creates and returns a string representation of the current exception

Topics

- Runtime Errors and Exceptions
- Handling exceptions
- Exception class hierarchy
- ***finally* block**
- Throwing exceptions
- Defining custom exceptions
- Exception propagation

`finally` Block

The `finally` block is **always executed** regardless of whether an exception occurred or not

```
try
{
    // Main logic code
}
catch (Exception_Type1 e1)
{
    // Exception handling code
}
catch (Exception_Type2 e2)
{
    // Exception handling code
}
finally
{
    // Clean up code
}
// execution continues
```

Because always being executed, `finally` block is usually used for **cleaning up** allocated resources such as closing a file

`finally` Blocks

1. If **no exception** is generated, the statements in the `finally` block are **executed after the statements in the `try` block complete**
2. If an **exception is generated**, the statements in the `finally` block are executed **after the statements in the appropriate `catch` block complete**
3. If an **exception is uncaught**, the `finally` block will **still be executed**

Quiz

What is the output if the following method is executed?

```
public static void TestFinally1() {
  int[] arr = new int[3];
  try {
    Console.WriteLine("Enter try block.");
    for (int i = 0; i < arr.Length; i++) {
      arr[i] = i;
      Console.WriteLine(arr[i]);
    }
    Console.WriteLine("Exit try block.");
  }
  catch (IndexOutOfRangeException e) {
    Console.WriteLine("Exception caught.");
  }
  finally {
    Console.WriteLine("CleanUp.");
  }
  Console.WriteLine("End of method. ");
}
```

Quiz

What is the output if the following method is executed?

```csharp
public static void TestFinally2() {
  int[] arr = new int[3];
  try {
    Console.WriteLine("Enter try block.");
    for (int i = 0; i < 5; i++) {
      arr[i] = i;
      Console.WriteLine(arr[i]);
    }
    Console.WriteLine("Exit try block.");
  }
  catch (IndexOutOfRangeException e) {
    Console.WriteLine("Exception caught.");
  }
  finally {
    Console.WriteLine("CleanUp. ");
  }
  Console.WriteLine("End of method. ");
}
```

Quiz

What is the output if the following method is executed?

```
public static void TestFinally3() {
  int[] arr = new int[3];
  try {
    Console.WriteLine("Enter try block.");
    for (int i = 0; i < 5; i++) {
      arr[i] = i;
      Console.WriteLine(arr[i]);
    }
    Console.WriteLine("Exit try block.");
  }
  catch (DivideByZeroException e) {
    Console.WriteLine("Exception caught.");
  }
  finally {
    Console.WriteLine("CleanUp. ");
  }
  Console.WriteLine("End of method. ");
}
```

Why is *“End of method.”* not in the output?

Next

So C# runtime system throws exceptions and we can handle them

Can we **throw exceptions ourselves**?

Image by Keith Johnston from Pixabay

Topics

- Runtime Errors and Exceptions
- Handling exceptions
- Exception class hierarchy
- finally block
- **Throwing exceptions**
- Defining custom exceptions
- Exception propagation

Throwing Exceptions

We can **explicitly throw** an exception object, which has the **same effect** as the ones thrown by runtime system

```
1 ExceptionType ex =
      new ExceptionType(
                   "More information");
2 throw ex;
```

1. Create an exception by instantiating an instance of exception

2. Throw the instance

Quiz

What is the output if the following method is executed and the user input is 0?

```
public static void TestThrowException() {
   try {
      Console.WriteLine("Please enter a divisor");
      string input = Console.ReadLine();
      int divisor = int.Parse(input);

      if (divisor == 0) {
         throw new DivideByZeroException(
               "You ask for an invalid division");
      }

      int quotient = 10 / divisor;
      Console.WriteLine(quotient);
   }
   catch (DivideByZeroException e) {
      Console.WriteLine("Exception");
      Console.WriteLine(e.Message);
   }
}
```

Next

So far, we can catch and also explicitly throw **pre-defined exceptions**

Can we define **our own exceptions**?

Topics

- Runtime Errors and Exceptions
- Handling exceptions
- Exception class hierarchy
- finally block
- Throwing exceptions
- **Defining custom exceptions**
- Exception propagation

User-Defined Exceptions

- We can define our own exceptions by **deriving** from `Exception` class
  - Able to add other fields and properties

- Then, throw our custom exceptions and catch them in the **usual manner**

A Custom Exception

1. Create a class that **extends** ***Exception*** **class**
2. Implement some **constructors** (2 in this case)

   call the **parent's constructors**

```
public class 1 MyException : Exception
{
    public 2 MyException() : base()
    {
        // empty body
    }

    public 2 MyException(
        string message) : base(message)
    {
        // empty body
    }
}
```

Can we add **attributes** to custom exception classes?

Topics

- Runtime Errors and Exceptions
- Handling exceptions
- Exception class hierarchy
- finally block
- Throwing exceptions
- Defining custom exceptions
- **Exception propagation**

Exception Propagation

- Up to now, all methods **where exceptions are thrown** catch them (*option B1*)
- If a method does not catch a thrown exception, the exception will be **propagated to the calling method**
- The **propagation continues** until
  - The exception is **caught** (*option B1 and B2*), or
  - The exception **reaches the outermost level** and therefore **handled by the default exception handler** (*option A*)

Exception Propagation Example

```
public static void Main()
{
   try {
      Method1();
   } catch (Exception e) {
      Console.WriteLine(e.Message);
   }
}
static void Method1()
{
   Method2();
}
static void Method2()
{
   throw new Exception(
         "Exception thrown in Method2()");
}
```

Uncaught exceptions are **propagated** to the **calling method**

```
Exception thrown in Method2()
```

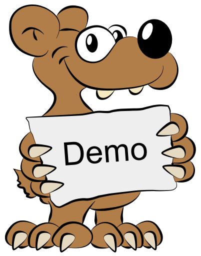
Quiz

What is the output of the following program?

```
public class ExcPropagation {
  public void M1() {
    try {
      Console.WriteLine("Enter try block of M1.");
      M2();
      Console.WriteLine("Exit try block of M1.");
    }
    catch (DivideByZeroException e) {
      Console.WriteLine("Enter catch block of M1.");
      Console.WriteLine("Exception from: {0}",
                              e.TargetSite);
      Console.WriteLine("Exit catch block of M1.");
    }
    Console.WriteLine("Exit M1.");
  }

  public void M2() {
    Console.WriteLine("Enter M2.");
    int y = 0;
    int x = 10 / y; // Exception!!!
    Console.WriteLine("Exit M2.");
  }
}
```

```
public static void Main() {
  Console.WriteLine("Enter Main.");
  ExcPropagation mc = new ExcPropagation();
  try {
    mc.M1();
  }
  catch (Exception e) {
    Console.WriteLine("Enter catch block of Main.");
    Console.WriteLine("Exception from: {0}",
                             e.TargetSite);
    Console.WriteLine("Exit catch block of Main.");
  }
  Console.WriteLine("Exit Main.");
}
```



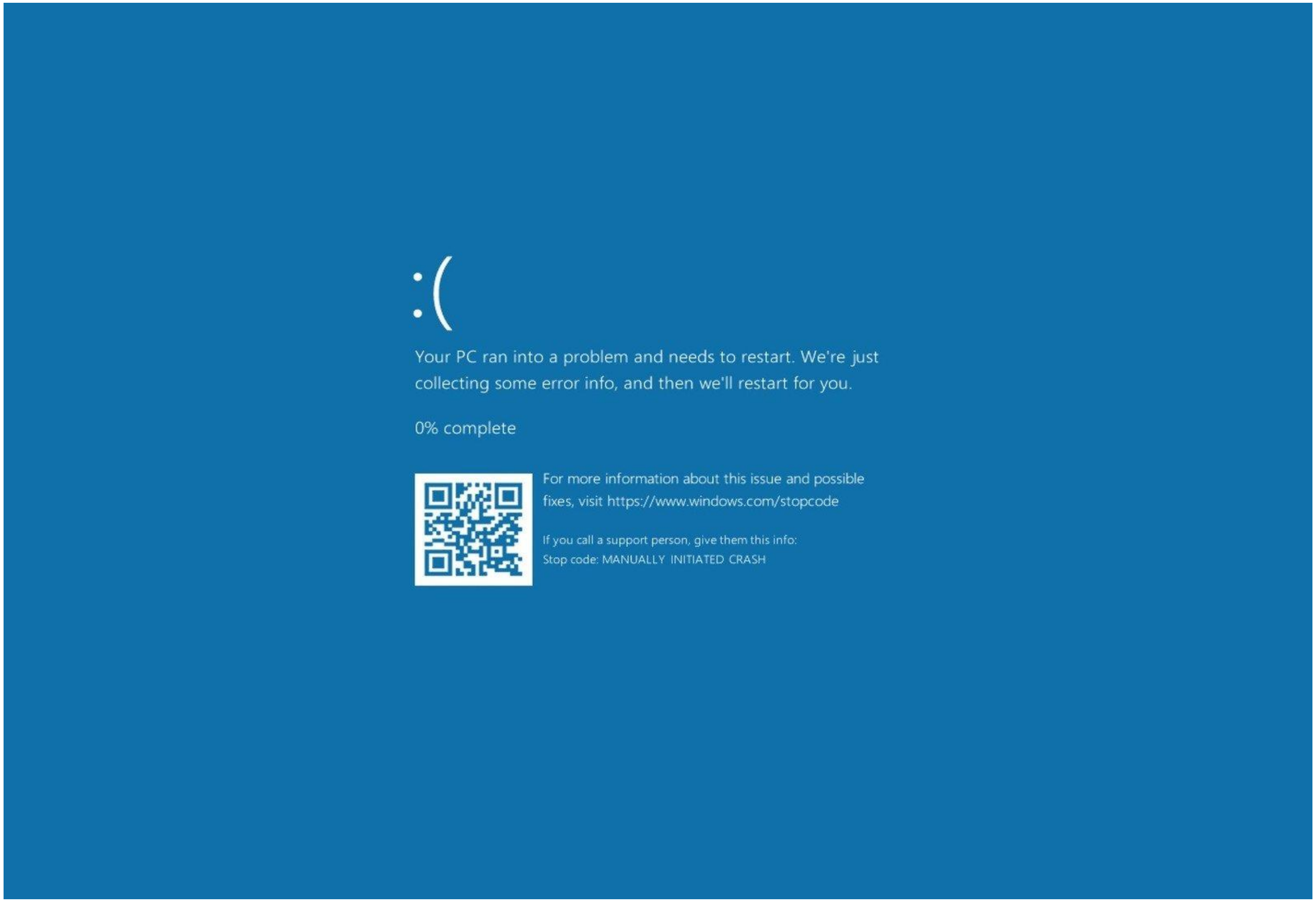
Exception Propagation Benefits

- Allow the **separation** of identifying/reporting errors from reacting to errors – possibly in **separate classes**
  - **Error is identified in one place**
  - **Exception handler is located in another place**
  - Linkage is through the exception object that is thrown
- Therefore, increase the **robustness** of programs without making code logic more complex
  - Instead of every piece of program needs to check for unusual cases, they can **focus on their job**
- One of the **most important characteristics** of C# Exception Handling

Readings

- Visual C# 2012 How to Program, 5th edition – Chapter 13, Exception Handling: A Deeper Look, *Paul Deitel and Harvey Deitel*
- Exception Class https://learn.microsoft.com/en-us/dotnet/api/system.exception?view=net-6.0

Uncaught Exceptions in real world



:(

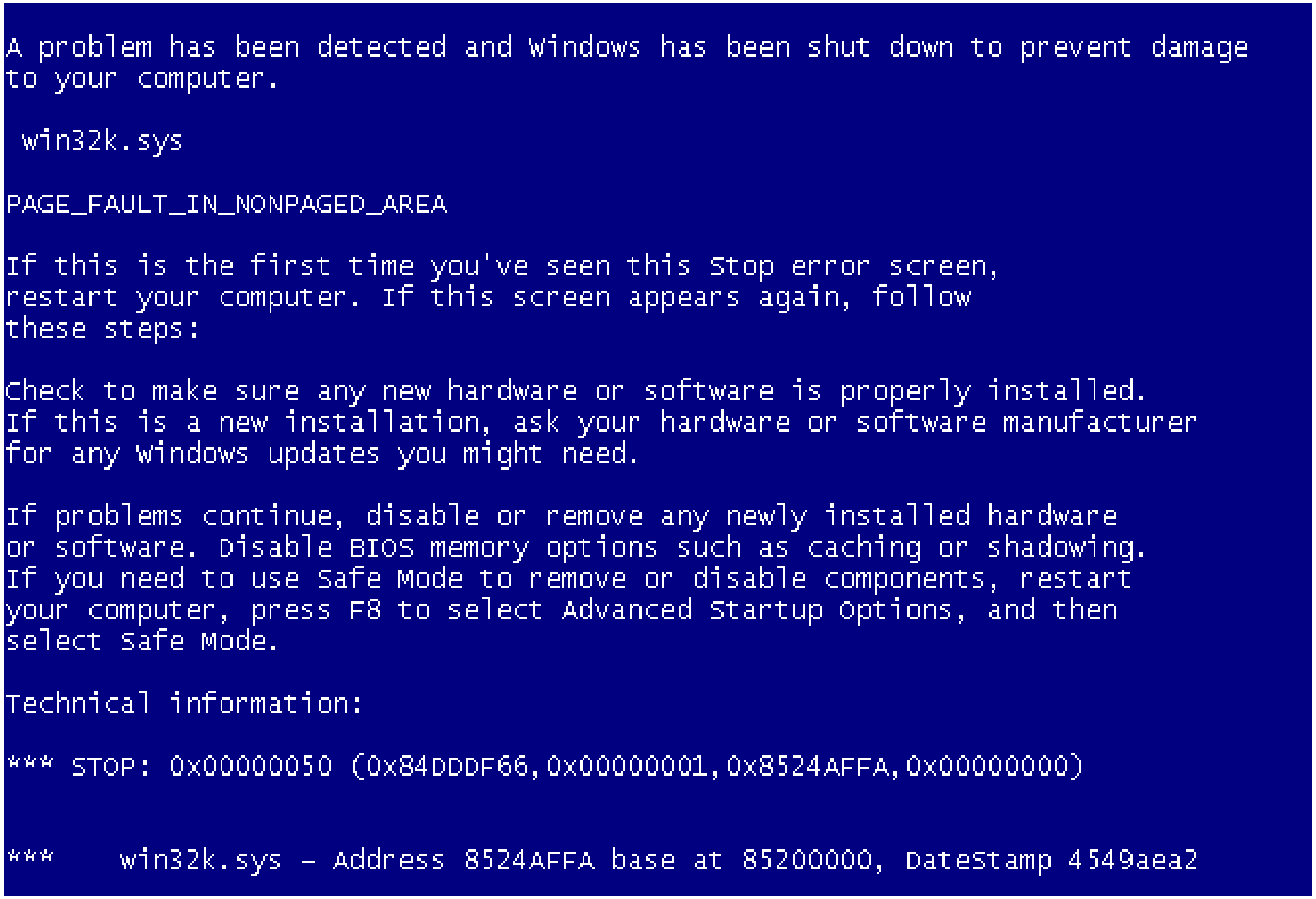
Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

0% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:
Stop code: MANUALLY INITIATED CRASH

Uncaught Exceptions in real world



```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

 win32k.sys

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0x84DDDF66,0x00000001,0x8524AFFA,0x00000000)


***    win32k.sys - Address 8524AFFA base at 85200000, DateStamp 4549aea2
```