Deakin University

Object Oriented Development

OnTrack Submission

Exceptions and Error Handling

Submitted By:
Romil Bijarnia
s222528574
2024/08/20 01:52

Tutor:
Jotham Barazani

Outcome	**Weight**
Evaluate Code	♦♦♦◊◊
Principles	♦♦♦◊◊
Build Programs	♦♦♦◊◊
Design	♦♦♦◊◊
Justify	♦♦♦◊◊

nice assignment

August 20, 2024

Introduction

This report provides an analysis of various exception types in C# as part of the SIT232 Object-Oriented Development unit. The goal is to understand different exceptions that can occur during runtime, determine the responsible party for throwing each exception, and discuss appropriate handling strategies. This report is complemented by the accompanying C# code file (`Program.cs`) that demonstrates each exception in practice.

Exception Descriptions

1. NullReferenceException

- **Possible Situation**: This exception occurs when you attempt to access a member of a null object. For instance, trying to access the length of a null string (`string str = null; Console.WriteLine(str.Length);`).
- **Who Throws the Exception**: The runtime system throws this exception when the code attempts to dereference a null object.
- **Message to User**: "An operation was attempted on a null object. Please ensure the object is properly initialized before use."
- **Handling Strategy**: This exception should generally be caught and handled, as it indicates a logical error or unexpected null input from the user. Proper error messages should guide the user to correct the issue.
- **Avoidance**: To avoid this exception, always check for null before dereferencing objects. Defensive programming techniques, such as null checks and initializing objects properly, are essential.

2. IndexOutOfRangeException

- **Possible Situation**: This exception occurs when accessing an array or collection with an index that is out of its bounds. For example, trying to access the 10th element of an array that only has 5 elements (`int[] array = new int[5]; Console.WriteLine(array[10]);`).
- **Who Throws the Exception**: The runtime system throws this exception when the index is outside the valid range.
- **Message to User**: "Index is out of the valid range. Please ensure that the index is within the bounds of the array or collection."

- **Handling Strategy**: This exception should be caught to prevent the program from crashing. Proper validation of index values before accessing the collection is necessary.
- **Avoidance**: Always validate the index before accessing an array or collection. Use loops with well-defined boundaries to avoid accidentally exceeding the array limits.

3. StackOverflowException

- **Possible Situation**: This occurs due to excessive deep or infinite recursion. For example, a method repeatedly calling itself without a termination condition (`void CauseStackOverflow() { CauseStackOverflow(); }`).
- **Who Throws the Exception**: The runtime system throws this exception.
- **Message to User**: Typically, this exception causes program termination and should be avoided rather than handled.

- **Handling Strategy**: This exception cannot be caught by a `try-catch` block and should be avoided by ensuring that recursive methods have proper base cases to terminate the recursion.
- **Avoidance**: Ensure that any recursive function has a termination condition. Consider using iterative solutions for deep recursive algorithms.

4. OutOfMemoryException

- **Possible Situation**: This occurs when the system runs out of memory, for instance, when trying to allocate a very large array (`int[] largeArray = new int[int.MaxValue];`).
- **Who Throws the Exception**: The runtime system throws this exception.
- **Message to User**: "The system ran out of memory. Please try reducing the memory usage or optimizing the code."
- **Handling Strategy**: This exception usually indicates a severe issue and cannot be reliably caught. It should be avoided by managing memory usage more effectively.
- **Avoidance**: Avoid creating excessively large data structures. Consider optimizing data structures and algorithms to use less memory.

5. DivideByZeroException

- **Possible Situation**: This exception occurs when a division by zero is attempted (`int x = 10; int y = 0; Console.WriteLine(x / y);`).
- **Who Throws the Exception**: The runtime system throws this exception when a division by zero is attempted.
- **Message to User**: "Division by zero is not allowed. Please provide a non-zero divisor."
- **Handling Strategy**: This exception should be caught to prevent the program from crashing. Proper validation of the divisor before performing the division is necessary.
- **Avoidance**: Always check the divisor before performing division to ensure it is not zero.

6. ArgumentNullException

- **Possible Situation**: This exception is thrown when a method receives a `null` argument that it does not accept (`string str = null; PrintString(str);`).
- **Who Throws the Exception**: The programmer should throw this exception to indicate that a null argument is not allowed.
- **Message to User**: "An argument was null where it should not have been. Please provide a valid argument."
- **Handling Strategy**: This exception should be caught to ensure that the program handles invalid inputs gracefully.
- **Avoidance**: Always validate method arguments to ensure they are not null before proceeding with the logic.

7. ArgumentOutOfRangeException

- **Possible Situation**: This exception is thrown when an argument passed to a method is outside the acceptable range (`PrintNumberInRange(101);`).
- **Who Throws the Exception**: The programmer should throw this exception when an argument is outside the valid range.
- **Message to User**: "Argument is out of the acceptable range. Please provide a value within the allowed range."
- **Handling Strategy**: This exception should be caught to prevent incorrect usage of the method.
- **Avoidance**: Validate method arguments to ensure they are within the acceptable range before performing operations.

8. FormatException

- **Possible Situation**: This exception occurs when the format of an argument is invalid, for instance, when trying to parse a string into an integer (`int number = int.Parse("NotANumber");`).
- **Who Throws the Exception**: The runtime system throws this exception when the format is invalid.
- **Message to User**: "Invalid format. Please ensure the input matches the required format."
- **Handling Strategy**: This exception should be caught to allow the user to correct their input.
- **Avoidance**: Validate the format of user inputs before processing them.

9. ArgumentException

- **Possible Situation**: This exception is thrown when an argument provided to a method is not valid (`CreateArray(-5);`).
- **Who Throws the Exception**: The programmer should throw this exception when an argument does not meet the method's requirements.
- **Message to User**: "Invalid argument. Please provide a valid value."
- **Handling Strategy**: This exception should be caught to prevent incorrect usage of the method.
- **Avoidance**: Validate all method arguments to ensure they meet the method's requirements before proceeding.

10. SystemException

- **Possible Situation**: This is the base class for many other exceptions, and it can be thrown for various system-level errors (`throw new SystemException("This is a custom system exception");`).
- **Who Throws the Exception**: The programmer can throw this exception to indicate a generic system error.
- **Message to User**: "A system error occurred. Please contact support."
- **Handling Strategy**: This exception should be caught when you want to handle a generic system error, but specific exceptions are preferred.
- **Avoidance**: Prefer using more specific exceptions rather than SystemException.

Conclusion

Exception handling is a critical aspect of software development that ensures the robustness and reliability of applications. By understanding and properly handling exceptions, developers can create programs that are resilient to errors and provide meaningful feedback to users. This report and the accompanying code demonstrate the handling of common exceptions in C# and provide best practices for avoiding these exceptions.

```csharp
using System;

class Program
{
    static void Main(string[] args)
    {
        // NullReferenceException
        try
        {
            string str = null;
            Console.WriteLine(str.Length);
        }
        catch (NullReferenceException ex)
        {
            Console.WriteLine("The following error detected: " + ex.GetType() + "
            ↪  with message \"" + ex.Message + "\"");
        }

        // IndexOutOfRangeException
        try
        {
            int[] array = new int[5];
            Console.WriteLine(array[10]);
        }
        catch (IndexOutOfRangeException ex)
        {
            Console.WriteLine("The following error detected: " + ex.GetType() + "
            ↪  with message \"" + ex.Message + "\"");
        }

        // StackOverflowException (cannot be caught by a catch block)


        // DivideByZeroException
        try
        {
            int x = 10;
            int y = 0;
            Console.WriteLine(x / y);
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine("The following error detected: " + ex.GetType() + "
            ↪  with message \"" + ex.Message + "\"");
        }

        // ArgumentNullException
        try
        {
            string str = null;
            PrintString(str);
        }
        catch (ArgumentNullException ex)
        {
            Console.WriteLine("The following error detected: " + ex.GetType() + "
            ↪  with message \"" + ex.Message + "\"");
        }

        // ArgumentOutOfRangeException
        try
        {
            PrintNumberInRange(101);
        }
        catch (ArgumentOutOfRangeException ex)
        {
            Console.WriteLine("The following error detected: " + ex.GetType() + "
            ↪  with message \"" + ex.Message + "\"");
        }

        // FormatException
        try
        {
            int number = int.Parse("NotANumber");
        }
        catch (FormatException ex)
        {
            Console.WriteLine("The following error detected: " + ex.GetType() + "
            ↪  with message \"" + ex.Message + "\"");
        }

        // ArgumentException
        try
        {
            CreateArray(-5);
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("The following error detected: " + ex.GetType() + "
            ↪  with message \"" + ex.Message + "\"");
        }

        // SystemException (base class for many exceptions)
        try
        {
            throw new SystemException("This is a custom system exception");
        }
        catch (SystemException ex)
        {
            Console.WriteLine("The following error detected: " + ex.GetType() + "
            ↪  with message \"" + ex.Message + "\"");
        }
    }

    static void CauseStackOverflow()
    {
        CauseStackOverflow(); // Recursive call without termination condition causes
        ↪  StackOverflowException
    }

    static void PrintString(string str)
    {
        if (str == null)
            throw new ArgumentNullException(nameof(str), "String cannot be null");
        Console.WriteLine(str);
    }

    static void PrintNumberInRange(int number)
    {
        if (number < 0 || number > 100)
            throw new ArgumentOutOfRangeException(nameof(number), "Number must be
            ↪  between 0 and 100");
        Console.WriteLine(number);
    }

    static int[] CreateArray(int size)
    {
        if (size < 0)
            throw new ArgumentException("Array size cannot be negative",
            ↪  nameof(size));
        return new int[size];
    }
}
```