**C# User-Defined Exceptions**

C# allows us to create user-defined or custom exception. It is used to make the meaningful exception. To do this, we need to inherit Exception class.

C# user-defined exception example

1. **using** System;
2. **public** **class** InvalidAgeException : Exception
3. {
4. **public** InvalidAgeException(String message)
5. : **base**(message)
6. {
8. }
9. }
10. **public** **class** TestUserDefinedException
11. {
12. **static** **void** validate(**int** age)
13. {
14. **if** (age < 18)
15. {
16. **throw** **new** InvalidAgeException("Sorry, Age must be greater than 18");
17. }
18. }
19. **public** **static** **void** Main(**string**[] args)
20. {
21. **try**
22. {
23. validate(12);
24. }
25. **catch** (InvalidAgeException e) { Console.WriteLine(e); }
26. Console.WriteLine("Rest of the code");
27. }
28. }

Output:

*InvalidAgeException: Sorry, Age must be greater than 18*

*Rest of the code*

**Custom Exceptions in C#**

Introduction

Exception handling is a fundamental aspect of software development, ensuring that your applications can gracefully respond to unexpected errors or exceptional situations. In C#, exceptions are used to handle such scenarios. While C# provides a rich set of built-in exception classes, there are times when you need to create custom exceptions tailored to your application's specific needs. This article explores the concept of custom exceptions in C# and how they can be used to improve error handling.

Understanding Exceptions in C#

Before diving into custom exceptions, let's briefly recap how exceptions work in C#. An exception is an abnormal or unexpected event that occurs during the execution of a program. It disrupts the normal flow of code and is used to indicate that something went wrong. Exceptions can be thrown explicitly using the throw keyword or can be raised implicitly by the runtime environment when an error occurs, such as accessing an out-of-bounds array index or dividing by zero.

C# provides a hierarchy of exception classes that cover various types of errors, with the base class being System.Exception. Derived classes, such as System.NullReferenceException or System.DividedByZeroException, represents specific error scenarios. Developers can catch and handle exceptions using the try-catch statement, which allows for graceful recovery or termination of the application.

Why Use Custom Exceptions?

While the built-in exception classes are versatile and cover a wide range of scenarios, there are situations where custom exceptions become essential. Here are some compelling reasons to create custom exceptions in your C# application:

1. **Specificity:** Custom exceptions allow you to provide more specific information about the error that occurred instead of relying on generic exceptions like System.Exception, you can create custom exceptions with meaningful names and additional details, making it easier to pinpoint the cause of the error.
2. **Control:** Custom exceptions provide you with greater control over how errors are handled in your application. You can define custom exception handling logic for each type of exception, ensuring that errors are dealt with in a way that aligns with your application's requirements.
3. **Documentation:**Well-named custom exceptions serve as self-documenting code. Developers working with your codebase can quickly understand the purpose of an exception and the circumstances under which it should be thrown.
4. **Maintainability:** Custom exceptions make your codebase more maintainable by centralizing error-handling logic. When you create custom exceptions for specific error scenarios, you reduce the complexity of try-catch blocks and improve code readability.

Creating Custom Exceptions in C#

To create a custom exception in C#, you need to define a new class that inherits from the System.Exception base class or one of its derived classes. Here's a simple example of creating a custom exception.

using System;

public class MyCustomException : Exception

{

public MyCustomException()

{

}

public MyCustomException(string message)

: base(message)

{

}

public MyCustomException(string message, Exception innerException)

: base(message, innerException)

{

}

}

C#

Copy

In this example, MyCustomException inherits from System.Exception and provides three constructors, allowing you to specify a custom error message and an inner exception if needed.

Throwing Custom Exceptions

Once you've defined your custom exception class, you can throw it using the throw keyword in your code. Here's an example of how to throw the MyCustomException.

public class CustomExceptionDemo

{

public void DoSomething()

{

// Simulate an error

if (someCondition)

{

throw new MyCustomException("An error occurred during 'DoSomething'.");

}

}

}

C#

Copy

Handling Custom Exceptions

To handle custom exceptions, you use the same try-catch blocks as you would with built-in exceptions. Here's an example of how to catch and handle the MyCustomException.

try

{

CustomExceptionDemo demo = new CustomExceptionDemo();

demo.DoSomething();

}

catch (MyCustomException ex)

{

Console.WriteLine($"Custom Exception Caught: {ex.Message}");

// Handle the exception or log it

}

C#

Copy

In this example, we catch the MyCustomException and provide custom handling logic, such as logging or displaying a user-friendly error message.

Best Practices for Custom Exceptions

When working with custom exceptions in C#, consider the following best practices:

1. **Choose Descriptive Names:** Give your custom exceptions meaningful names that convey the nature of the error.
2. **Provide Clear Messages:** Include informative error messages in your custom exceptions to aid debugging and troubleshooting.
3. **Inherit Appropriately:** Derive your custom exception class from the most appropriate base class, such as System.Exception or one of its derived classes.
4. **Use Inner Exceptions:**When necessary, include inner exceptions to provide additional context about the error.
5. **Document Exceptions:** Add XML comments to your custom exception classes to provide documentation for other developers.
6. **Centralize Exception Handling:**Centralize exception handling code to avoid duplicating error-handling logic throughout your application.
7. **Logging:** Consider using a logging framework to record exceptions and their details, aiding in debugging and post-mortem analysis.
8. **Testing:**Include unit tests for your custom exceptions and their handling code to ensure they work as expected.

Conclusion

Custom exceptions in C# are a powerful tool for improving error handling in your applications. They allow you to create specific, self-documenting, and maintainable error-handling mechanisms tailored to your application's unique requirements. By following best practices and using custom exceptions judiciously, you can enhance the reliability and maintainability of your C# codebase.