

Agenda

- What is an Exception?
- Exception Handling in C#
- Common Exception Classes
 - `try - catch - finally` statements
- Unhandled Exceptions
- Handled Exceptions
- Propagating Exceptions
- User-Defined Exceptions
- Best Practices for Handling Exceptions



What is an Exception?

- Exception is
 - An error is any error or unexpected behavior that is encountered by an executing program.
 - Exceptions can be raised because of a fault in your code or in code that you call.
 - An exception is thrown from an area of code where a problem has occurred.
 - An Exception is an object that inherits from the System.Exception class.

Exception Handling in C#

- Exceptions are handled by using a try statement in C#.
- When an exception occurs, the system searches for the nearest catch clause that can handle the exception, as determined by the run-time type of the exception:
 - First, the current method searches for a lexically enclosing try statement, and the associated catch clauses of the try statement are considered in order.
 - If that fails, the method that called the current method searches for a lexically enclosing try statement that encloses the point of the call to the current method. This search continues until a catch clause is found to handle the current exception.

Exception Handling in C# (cont.)

- Before execution of the catch clause can begin, the system executes in order, any finally clauses that are associated with try statements that are more nested than the one that caught the exception.
- If no matching catch clause is found, one of two things occurs:
 - If the search for a matching catch clause reaches a static constructor or static field initializer, then a `System.TypeInitializationException` is thrown at the point that triggered the invocation of the static constructor.
 - If the search for matching catch clause reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

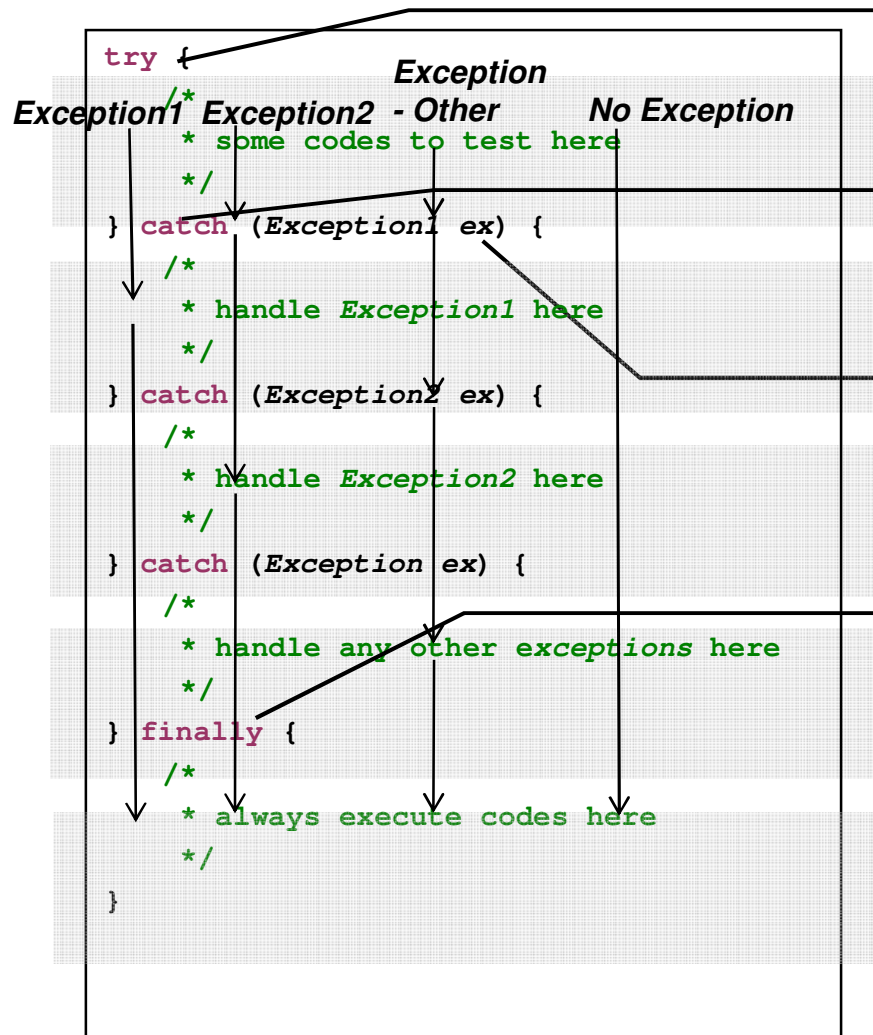
Common Exception Classes

- The most common exception classes in C# are:
 - `System.ArithmeticException`:
 - A base class for exceptions that occur during arithmetic operations, such as `System.DivideByZeroException` and `System.OverflowException`.
 - `System.ArrayTypeMismatchException`:
 - Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
 - `System.DivideByZeroException`:
 - Thrown when an attempt to divide an integral value by zero occurs.
 - `System.IndexOutOfRangeException`:
 - Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
 - `System.InvalidCastException`:
 - Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.

Common Exception Classes (cont.)

- Other common Exception classes in C#:
 - `System.NullReferenceException`:
 - Thrown when a null reference is used in a way that causes the referenced object to be required.
 - `System.OutOfMemoryException`:
 - Thrown when an attempt to allocate memory (via `new`) fails.
 - `System.OverflowException`:
 - Thrown when an arithmetic operation in a checked context overflows.
 - `System.StackOverflowException`:
 - Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
 - `System.TypeInitializationException`:
 - Thrown when a static constructor throws an exception, and no catch clauses exists to catch it.

try-catch()-finally



`try` block encloses the context where a possible exception can be thrown

each `catch()` block is an exception handler and can appear several times

Exception1 should not shadow *Exception2* which in turn should not shadow *Exception* (based on the exception hierarchy)

`finally` block is always executed before exiting the `try` statement. `finally` block is optional but must appear once after the `catch()` blocks

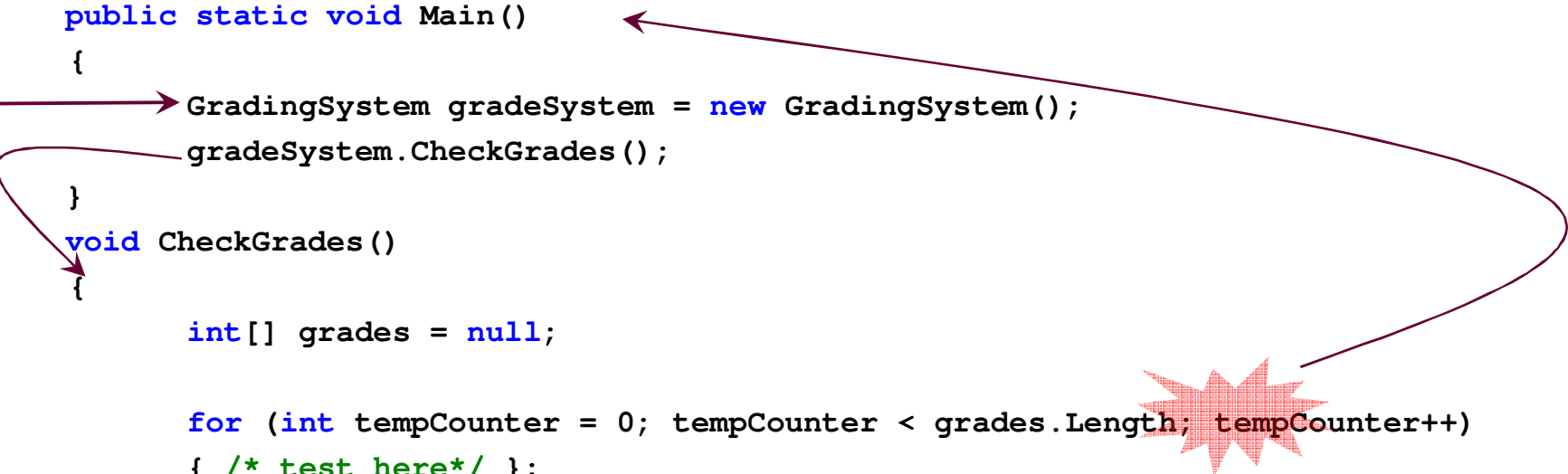
at least one `catch()` block must appear in the `try` statement but `finally`

block is optional.

Unhandled Exceptions

```
public class GradingSystem
{
    public static void Main()
    {
        GradingSystem gradeSystem = new GradingSystem();
        gradeSystem.CheckGrades();
    }
    void CheckGrades()
    {
        int[] grades = null;

        for (int tempCounter = 0; tempCounter < grades.Length; tempCounter++)
        { /* test here*/ };
    }
}
```



The diagram illustrates the execution flow of the provided C# code. A red arrow originates from the `Main()` method and points to the `new GradingSystem()` statement. Another red arrow points from the `CheckGrades()` method call in `Main()` to the `CheckGrades()` method definition. A third red arrow starts from the `grades.Length` property access within the `for` loop and points to a red starburst symbol, indicating the point where an unhandled exception is thrown.

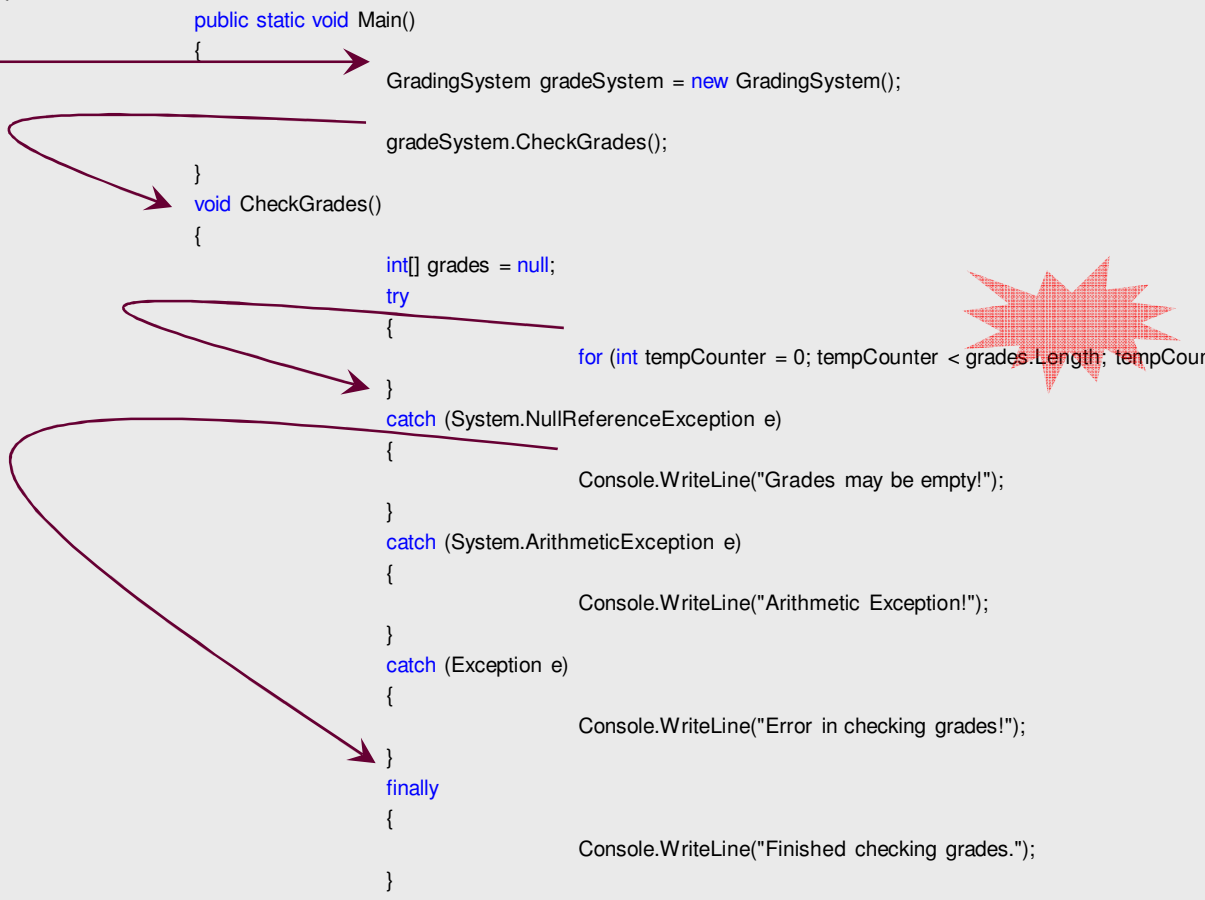
Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an object.

at TestConsoleApps.GradingSystem.CheckGrades()

at TestConsoleApps.GradingSystem.Main()

Handled Exceptions

```
using System;
public class GradingSystem
{
    public static void Main()
    {
        GradingSystem gradeSystem = new GradingSystem();
        gradeSystem.CheckGrades();
    }
    void CheckGrades()
    {
        int[] grades = null;
        try
        {
            for (int tempCounter = 0; tempCounter < grades.Length; tempCounter++) { /* test here*/ };
        }
        catch (System.NullReferenceException e)
        {
            Console.WriteLine("Grades may be empty!");
        }
        catch (System.ArithmeticException e)
        {
            Console.WriteLine("Arithmetic Exception!");
        }
        catch (Exception e)
        {
            Console.WriteLine("Error in checking grades!");
        }
        finally
        {
            Console.WriteLine("Finished checking grades.");
        }
    }
}
```



Grades may be empty!
Finished checking grades.

Propagating Exceptions

```
using System;
public class GradingSystem
{
    public static void Main()
    {
        GradingSystem gradeSystem = new GradingSystem();
        try
        {
            gradeSystem.CheckGrades();
        }
        catch (Exception e)
        {
            // must handle the exception thrown
            Console.WriteLine(e.Message);
        }
    }

    void CheckGrades()
    {
        int[] grades = {81,0,75};
        try
        {
            for (int tempCounter = 0; tempCounter < grades.Length; tempCounter++)
            {
                if (grades[tempCounter] <= 0)
                {
                    throw new Exception("Invalid grade!");
                }
            }
        }
        catch (System.NullReferenceException e)
        {
            Console.WriteLine("Grades may be empty!");
        }
        catch (System.ArithmeticException e)
        {
            Console.WriteLine("Problem while executing!");
        }
        catch (Exception e)
        {
            Console.WriteLine("Can't handle error here! Rethrowing...");
            throw new Exception(e.Message);
        }
    }
}
```

The diagram illustrates the flow of an exception. A red starburst marks the point where an exception is thrown in the `CheckGrades` method. A curved arrow shows the exception being propagated from the `try` block in `CheckGrades` to the `catch` block in `Main`. Another curved arrow shows the exception being propagated from the `catch` block in `Main` back to the `try` block in `Main`.

Can't handle error here! Rethrowing...
Invalid Grade!

User-Defined Exceptions

- Developers can write their own exceptions to programmatically distinguish between some error conditions. These user-defined exceptions can be created by deriving them from the `ApplicationException` class.
- It is a good practice to end the user-defined exception class name with the word “**Exception**”. This is useful in distinguishing between classes.
- It is advisable to implement the following three common constructors in every user-defined exception:
 - Default constructor without parameters
 - Constructor which takes a string message as a parameter
 - Constructor that takes a string message and inner exception as parameters

User-Defined Exceptions (cont.)

- Following is an example of a User-Defined Exception.

```
using System;
public class EmployeeListNotFoundException: ApplicationException
{
    public EmployeeListNotFoundException()
    {
    }

    public EmployeeListNotFoundException(string message)
        : base(message)
    {
    }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {
    }
}
```

Best Practices for Handling Exceptions

(1 of 4)

- The following list contains suggestions on best practices for handling exceptions:
 - Know when to set up a try / catch block:
 - For example, you can programmatically check for a condition that is likely to occur without using exception handling.
 - In other situations, using exception handling to catch an error condition is appropriate.
 - Use try / finally blocks around code that can potentially generate an exception and centralize your catch statements in one location:
 - In this way, the try statement generates the exception, the finally statement closes or deallocates resources, and the catch statement handles the exception from a central location.

Best Practices for Handling Exceptions

(2 of 4)

- The following list contains suggestions on best practices for handling exceptions:
 - Always order exceptions in catch blocks from the most specific to the least specific:
 - This technique handles the specific exception before it is passed to a more general catch block.
 - Try to avoid empty catch blocks.
 - End exception class names with the word "Exception".
 - In C#, use at least the three common constructors when creating your own exception classes.
 - In most cases, use the predefined exceptions types. Define new exception types only for programmatic scenarios.

Best Practices for Handling Exceptions

(3 of 4)

- More suggestions on best practices for handling exceptions:
 - Do not derive user-defined exceptions from the Exception base class:
 - For most applications, derive custom exceptions from the ApplicationException class.
 - Include a localized description string in every exception:
 - When the user sees an error message, it is derived from the description string of the exception that was thrown, rather than from the exception class.
 - Use grammatically correct error messages, including ending punctuation:
 - Each sentence in a description string of an exception should end in a period.

Best Practices for Handling Exceptions

(4 of 4)

- More suggestions on best practices for handling exceptions:
 - Provide Exception properties for programmatic access:
 - Include extra information in an exception (in addition to the description string) only when there is a programmatic scenario where the additional information is useful.
 - Return null for extremely common error cases.
 - Design classes so that an exception is never thrown in normal use.
 - The stack trace begins at the statement where the exception is thrown and ends at the catch statement that catches the exception:
 - Be aware of this fact when deciding where to place a throw statement
 - Use exception builder methods:
 - It is common for a class to throw the same exception from different places in its implementation.
 - To avoid excessive code, use helper methods that create the exception and return it.