Exception Handling

CSIS 3540

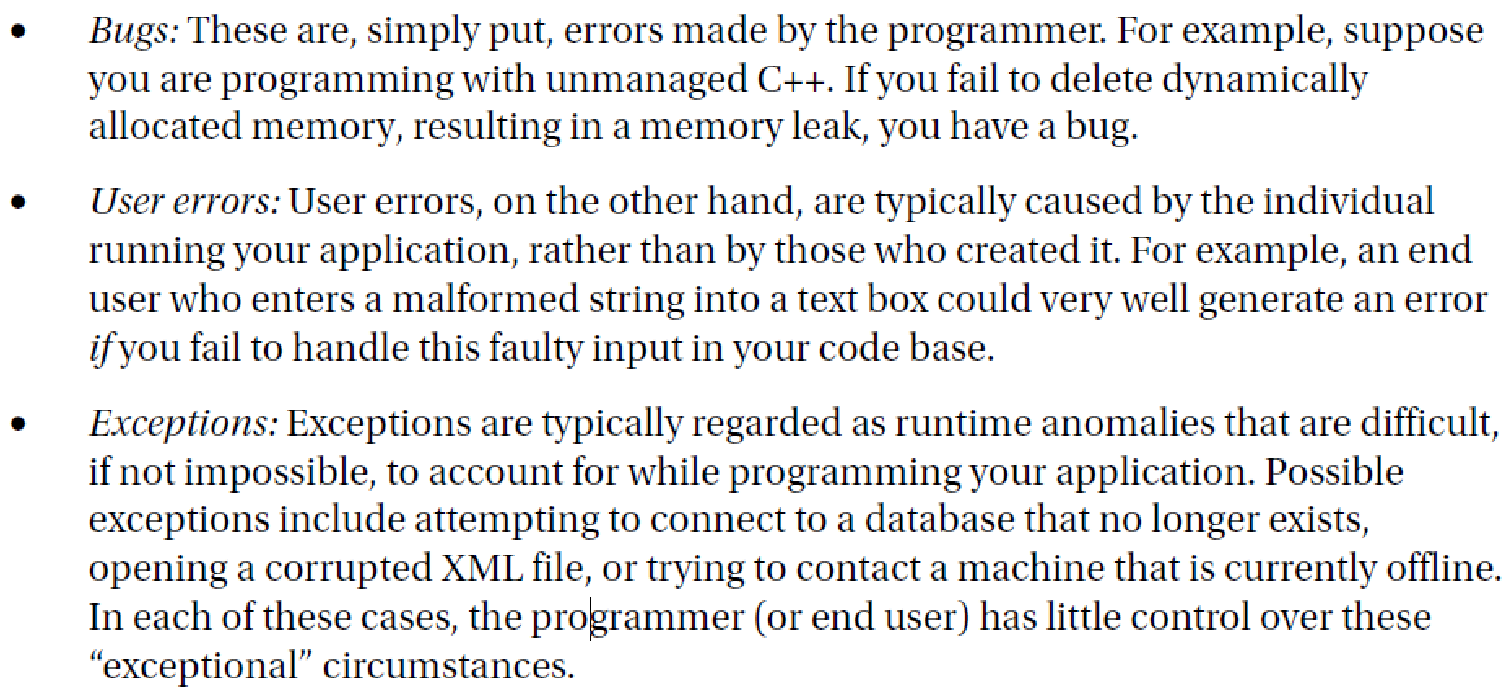
Client Server Systems

Class 03

Topics

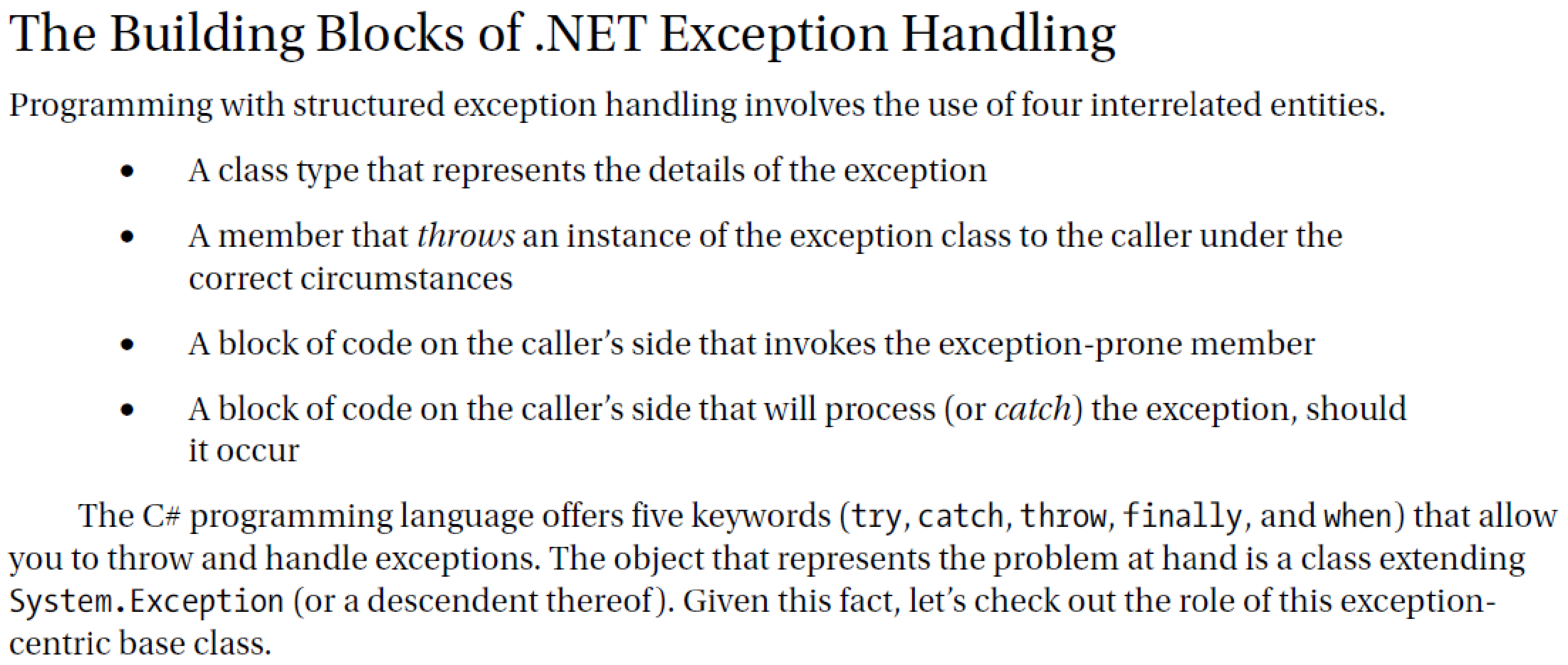
- Errors and Bugs
- Why exception handling
- Configuration
- System-level exceptions
- Application-level exceptions
- Multiple Exceptions
- Unhandled exceptions

Bugs, errors, and exceptions



- *Bugs:* These are, simply put, errors made by the programmer. For example, suppose you are programming with unmanaged C++. If you fail to delete dynamically allocated memory, resulting in a memory leak, you have a bug.
- *User errors:* User errors, on the other hand, are typically caused by the individual running your application, rather than by those who created it. For example, an end user who enters a malformed string into a text box could very well generate an error *if* you fail to handle this faulty input in your code base.
- *Exceptions:* Exceptions are typically regarded as runtime anomalies that are difficult, if not impossible, to account for while programming your application. Possible exceptions include attempting to connect to a database that no longer exists, opening a corrupted XML file, or trying to contact a machine that is currently offline. In each of these cases, the programmer (or end user) has little control over these "exceptional" circumstances.

Exception Handling overview



The Building Blocks of .NET Exception Handling

Programming with structured exception handling involves the use of four interrelated entities.

- A class type that represents the details of the exception
- A member that *throws* an instance of the exception class to the caller under the correct circumstances
- A block of code on the caller's side that invokes the exception-prone member
- A block of code on the caller's side that will process (or *catch*) the exception, should it occur

The C# programming language offers five keywords (`try`, `catch`, `throw`, `finally`, and `when`) that allow you to throw and handle exceptions. The object that represents the problem at hand is a class extending `System.Exception` (or a descendent thereof). Given this fact, let's check out the role of this exception-centric base class.

System.Exception members

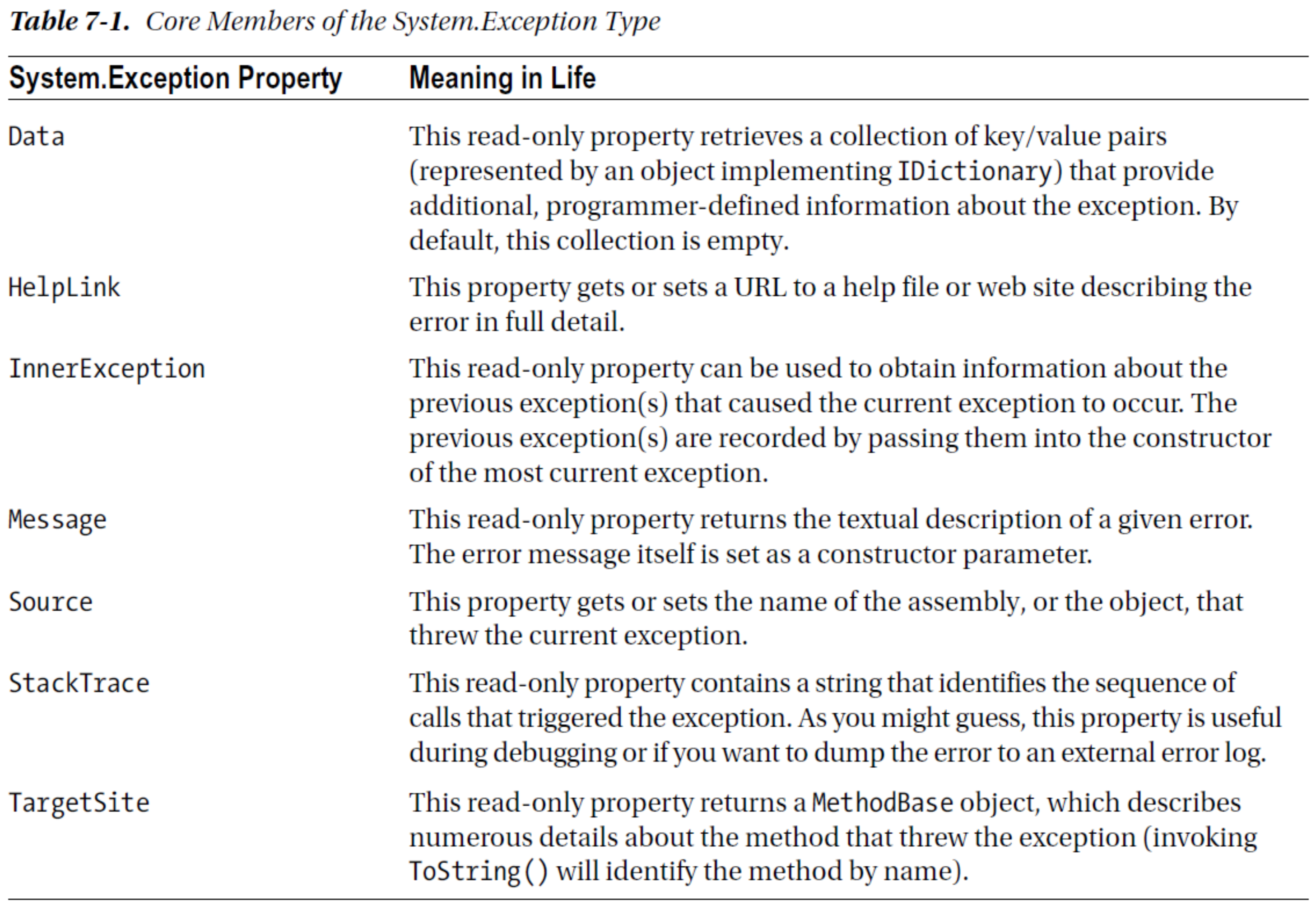


Table 7-1. *Core Members of the System.Exception Type*

System.Exception Property	Meaning in Life
`Data`	This read-only property retrieves a collection of key/value pairs (represented by an object implementing `IDictionary`) that provide additional, programmer-defined information about the exception. By default, this collection is empty.
`HelpLink`	This property gets or sets a URL to a help file or web site describing the error in full detail.
`InnerException`	This read-only property can be used to obtain information about the previous exception(s) that caused the current exception to occur. The previous exception(s) are recorded by passing them into the constructor of the most current exception.
`Message`	This read-only property returns the textual description of a given error. The error message itself is set as a constructor parameter.
`Source`	This property gets or sets the name of the assembly, or the object, that threw the current exception.
`StackTrace`	This read-only property contains a string that identifies the sequence of calls that triggered the exception. As you might guess, this property is useful during debugging or if you want to dump the error to an external error log.
`TargetSite`	This read-only property returns a `MethodBase` object, which describes numerous details about the method that threw the exception (invoking `ToString()` will identify the method by name).

Throwing an exception

- throw new Exception(string s)
- Exceptions should be thrown only when something terminal occurs
  - IE, the program should terminate
- Other errors should use a softer internal recovery scheme.

Setting Exception properties

- In addition to a string argument for the Exception constructor, properties can be set
  - HelpLink
  - Data
    - Use of Data.Add() method to add key/value pairs
- See **SimpleException** using **CommonCar**

```
/// <summary>
/// Accelerate method.
///     Tests to see if beyond max speed, and throws an exception.
///     Virtual to allow for override (custom exception)
/// </summary>
/// <param name="increasedSpeed">Amount of acceleration</param>

public virtual void Accelerate(int increasedSpeed)
{

    if (carIsDead)
        Console.WriteLine("{0} is dead - not working!", CarName);
    else
    {
        CurrentSpeed += increasedSpeed;
        if (CurrentSpeed >= maxSpeed)
        {
            carIsDead = true;
            CurrentSpeed = 0;

            // We need to call the HelpLink property, thus we need
            // to create a local variable before throwing the Exception object.
            Exception ex =
                new Exception(string.Format("{0} has overheated!", CarName))
                {
                    HelpLink = "http://www.CarsRUs.com"
                };

            // Stuff in custom data regarding the error.
            ex.Data.Add("TimeStamp",
                string.Format("The car exploded at {0}", DateTime.Now));
            ex.Data.Add("Cause", "You have a lead foot.");
            throw ex;
        }
        else
            Console.WriteLine($"Accelerated by {increasedSpeed} for {this}");
    }
}
```

Catching exceptions

- Use of try/catch, as below.
- Exception e
  - Properties:
    - TargetSite - type, name, and parameter types of the method that threw the exception.
    - Message – string that was passed to constructor when exception thrown
    - Source – program name
    - StackTrace – trace of callers
    - HelpLink – set by thrower
    - Data – set by thrower

```
// Speed up past the car's max speed to
// trigger the exception.
try
{
    for (int i = 0; i < 10; i++)
        myCar.Accelerate(10);
}
catch (Exception e)
{
    // show the data that an uncaught exception would also show when program crashes.

    Console.WriteLine("\n*** {0}: ERROR ***", Console.Title);
    Console.WriteLine("Member name: {0}", e.TargetSite);
    Console.WriteLine("Class defining member: {0}",
        e.TargetSite.DeclaringType);
    Console.WriteLine("Member type: {0}", e.TargetSite.MemberType);
    Console.WriteLine("Message: {0}", e.Message);
    Console.WriteLine("Source: {0}", e.Source);
    Console.WriteLine("Stack: {0}", e.StackTrace);
    Console.WriteLine("Help Link: {0}", e.HelpLink);
    Console.WriteLine("\n-> Custom Data:");
    foreach (DictionaryEntry de in e.Data)
        Console.WriteLine("-> {0}: {1}", de.Key, de.Value);
}
```

System and Application Exceptions

- Most exceptions are derived from System.Exception
  - EG, IndexOutOfRangeException
- System.SystemException may be useful
  - Determined at runtime
  - If(MyException is SystemException)
- Application
  - Extend Exception or ApplicationException
  - EG
    - public class CarIsDeadException : ApplicationException
    - Add custom members
    - throw CarIsDeadException
    - When catching, check to see if Exception is CarIsDeadExeption
      - catch (CarIsDeadException e)
- See
  - **CustomException**

Multiple Exceptions

- Sometimes multiple exceptions can result from a try clause
- Catch clauses should be ordered from most specific to general
    - IE, SystemIOException THEN Exception
- finally clause is always executed
- See
    - **ProcessMultipleExceptions**

VS and unhandled exceptions

- If an exception is thrown, VS will show you the exception in detail.
- You can then catch this in subsequent changes to code.