Expression Tree Code Demo (cont.)

Cpt S 321

Washington State University

Solutions we identified last time

- Throw more descriptive exceptions
- ~~2. Get rid of the hardcoded operators~~
- ~~5. Allow support for new operators without needing to change the logic in every method~~
- ~~1. Extract classes into separate files~~
- ~~3. Consider operator precedence/associativity~~
- ~~4. Parse the expression string and build the expression tree more elegantly~~
- Get rid of the redundant code

Where next?

~~+ 6. WE TESTED!~~

Let’s see some solutions

- What we did last time? We started testing:
    - We run the tests that we wrote the very first time that we started looking at the ExpressionTree code demo
    - Add tests to verify whether adding a new operator works properly
        - Test whether the precedence, associativity, etc. are the ones you set
        - Test whether more complex expressions that include the new operator are working as expected
        - Anything else?

Pointers for solutions ExpressionTreeCodeDemo

- **Handle exceptions properly**

- What’s the deal with exceptions?

- Exceptions are thrown to signal runtime errors (as opposed to C-style error code return which might get unnoticed whereas exceptions won’t)

Some examples of existing exceptions

- Exception class in the System namespace is the parent class for all existing exceptions in C#
  - SystemException
    - ArithmeticException
      - OverflowException
      - DivideByZeroException
    - ArgumentException
      - ArgumentNullException
      - ArgumentOutOfRangeException

…

Creating our own exceptions

- Sometimes the predefined exceptions do not fit our needs, in that case we can create our own exception types
  - Our exception must inherit from Exception (defined in the System namespace)
  - The name of the exception that we create must end with **Exception** (ex. MyCustomException)
  - Our exception must provide 3 constructors

Creating our own exceptions - example

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

An inner exception hold information about the previous exception (if any)

“Catching” or dealing with exceptions

- Managing a piece of code that can throw exception(s) is equivalent to placing that piece of code in a **try** block:

```
try {
    // exception may get thrown within execution of this block
}
catch (ExceptionA e) {
   // handle exception of type ExceptionA. Useful properties: e.Message and e.StackTrace.
}
catch (ExceptionB e) {
   // handle exception of type ExceptionB
}
finally { //A finally block always executes—whether or not an exception is thrown
           // and whether or not the try block runs to completion.
          // typically used for any sort of cleanup code
}
```

“Catching” or dealing with exceptions (cont.)

- If you don’t need to access properties of the exceptions, no need to declare a variable:

```
catch (OverflowException) // no variable
{
 ...
}
```

- Catching all exceptions:

```
catch {   // no type, no variable
...
}
```

Simple Example

```
class Test {
    static int MyCalculator (int numerator, int denominator) => numerator/denominator;
    static void Main(string[] args) {
        try {
                int result = MyCalculator (args[0], args[1]);
                Console.WriteLine (result);
        }
        catch (DivideByZeroException)
        {
                Console.WriteLine (”The denominator cannot be zero!");
        }
    }
}
```

Catching multiple exceptions

```
static void Main (string[] args)
{
        try
        {
                byte b = byte.Parse (args[0]);
                Console.WriteLine (b);
        }
        catch (IndexOutOfRangeException)
        {
                Console.WriteLine ("Please provide at least one argument");
        }
        catch (FormatException)
        {
                Console.WriteLine ("That's not a number!");
        }
        catch (OverflowException)
        {
                Console.WriteLine ("You've given me more than a byte!");
        }
}
```

The order is important!

Example with finally

```
static void ReadFile()
{
        StreamReader reader = null;
        try
        {
                reader = File.OpenText ("file.txt");
                if (reader.EndOfStream)
                        return;
                Console.WriteLine (reader.ReadToEnd());
        }
        finally {
                if (reader != null)
                        reader.Dispose();
        }
}
```

“Catching” or dealing with exceptions (cont.)

- What if you want to include a safety net to catch more general exceptions (such as System.Exception)?
    - You must put the more specific handlers first

Defensive programming

- Sometime we can/want to avoid throwing exceptions by using defensive programming

- Defensive programming is intended to ensure the continuing function of a piece of software under unforeseen circumstances. Defensive programming practices are often used where high availability, safety or security is needed.

- Checking for preventable errors is preferable to relying on try/catch blocks because exceptions are relatively expensive to handle, **taking hundreds of clock cycles or more**

```
class Test {
    static int MyCalculator (int numerator, int denominator) => numerator/denominator;

    static void Main(string[] args) {
        if(args[1]==0)
        {
                Console.WriteLine (”The denominator cannot be zero");
                // ask the user to re-enter the denominator, …
        }
        else
        {
                int result = MyCalculator(args[0], args[1]);
                Console.WriteLine (result);
        }
    }
}
```

Simple Example: re-written using defensive programming

Throwing exceptions

- Exceptions can be thrown either by the runtime or by you!

- Example:

```
static void Display (string name)
{
    if (name == null)
            throw new ArgumentNullException (nameof(name));
    Console.WriteLine (name);
}
```

Re-throwing exceptions

```
try {
        ...
}
catch (Exception ex)
{
        // Log error
        ...
        // Re-throw the same exception:
        throw;
}
```

Re-throwing exceptions (cont.)

```
try {
        ... // Parse a DateTime from XML element data
}
catch (FormatException e)
{
        // Re-throw a more specific exception:
        throw new XmlException ("Invalid DateTime", e);
}
```

Key Properties of System.Exception

- **StackTrace**: A string representing all the methods that are called from the origin of the exception to the catch block.

- **Message**: A string with a description of the error.

- **InnerException**: The inner exception (if any) that caused the outer exception. This, itself, may have another InnerException.

When to/not to deal with an exception

- When your code cannot recover from an exception, don't catch that exception, unless you need to log it in which case you need to re-throw it right away. Enable methods further up the call stack to recover if possible.

- Use exception handling if the event doesn't occur very often, that is, if the event is truly exceptional and indicates an error (such as an unexpected end-of-file).

Pointers for solutions ExpressionTreeCodeDemo

- **Get rid of redundant code**
  - **“Extract method” refactoring**:
    - When to use it: When we have a set of statements that can be grouped together in a **cohesive** method.
    - How to: Move those statements to a separate new method and replace all places where the redundant code occur with a call to this method. **In VS: select code and right click -> “Quick Actions and Refactorings…”-> “Extract Method”-> “Preview Changes” -> Apply (if ☺) or Cancel (if ☹)**
    - Note: Sometimes the code might be very similar but not identical. In those situations, think about whether you can identify what is different and whether you can have that passed as parameter(s) and still call the same method in those places.

ExpressionTreeCodeDemo

THE END