CSC322

Lecture Note 5

Dr. Victor Odumuyiwa

Learning Outcomes

At the end of this class, you should be able to:

- deal with bugs,
- write defensive programs,
- handle exception.

DEALING WITH BUGS

How?

- Static typing
  - Statically typed vs. dynamically type language
- Documenting assumptions
- Test-first programing

Hacking vs. Software Engineering

Hacky codes

- Writing lots of code before testing it
- Keeping all details in your head, assuming you will remember them forever instead of writing them down in your code
- Assuming bugs will be nonexistent or else easy to find and fix

Software Engineering

- Write a little bit at a time, testing as you go
- Document the assumptions that your code depends on
- Defend your code against stupidity

Static checking

- Static checking: bugs are found automatically before programs run.
- Dynamic checking: bugs are found automatically when the code is executed.
- No checking: the language doesn’t help you find the error at all; you have to watch for it yourself, or end up with wrong answers.

How to avoid bugs

- Make bugs impossible
  - Static-typing eliminates many runtime errors
- Localize bugs
  - If you cannot prevent bugs, you can try to localize them to small part of the program so you won't have to look too hard to find the course of the bug
  - Check the precondition(s) (defensive programming)
  - Incremental development: build only a bit of your program at a time, and test that bit thoroughly
    - Unit testing
    - Regression testing
  - Modularity
  - Encapsulation
    - Minimize the scope of variables
      - Always declare a loop control variable in the for-loop initializer
      - Declare a variable only when your first need it, and in the innermost curly-brace block that you can
      - Avoid global variable
- Debug systematically

Two Program Aspects

- Normal program aspects
  - Situations anticipated and dealt with in a conventional program flow
  - Programmed with the use of selective and iterative control structures
- Exceptional program aspects
  - Situations anticipated but not dealt with "in normal ways" by the programmer
    - Leads to an exception
    - Recoverable via exception handling or non-recoverable
  - Situations not anticipated by the programmer
    - Leads to an exception
    - Recoverable via exception handling or non-recoverable
  - Problems beyond the control of the program

Example

```
public static long Factorial(int n){
  if (n == 0)
    return 1;
  else return n * Factorial(n - 1);
}
```

Likely problems with the factorial method

1. Negative input
   - -ve N leads to infinite recursion (stackoverflowException)
2. Wrong type of input
3. Wrong type of multiplication
   - The operator may be redefined or overloaded to non-multiplication
4. Numeric overflow of returned numbers
5. Memory problem
   - We may run out of RAM memory during computation
6. Loss of power
7. Machine failure

Dealing with the factorial method problems

- Problem1could be dealt with as a normal program aspect
- Problem 2 is prevented by the analysis of the compiler (static typing)
- Problem 3 is prevented by the compiler
- Problem 4 could be dealt with by the use of another type than *long*
- Problem 5 could be foreseen as an anticipated exception
- Problem 5 and 7 are beyond the control of the program
- In extremely critical application, it may be necessary to deal with (handle) problem 6 and 7

How are errors handled?

- Ignore
  - False alarm
- Report
  - Write a message on the screen or in a log - Helpful for subsequent correction of the source program
- Terminate
  - Stopped the program execution in a controlled and gentle way – Save data, close connections
- Repair
  - Recover from the error in the running program – Continue normal program execution when the error is solved.

Where are errors handled?

- Handle errors at the place in the program where they occur
  - If possible, this is the easiest approach
  - Not always possible nor appropriate
- Handle errors at another place
  - Along the calling chain
  - Separation of concern

How to avoid bugs

- Make bugs impossible
  - Static-typing eliminates many runtime errors
- Localize bugs
  - If you cannot prevent bugs, you can try to localize them to small part of the program so you won't have to look too hard to find the course of the bug
  - Check the precondition(s) (defensive programming)
  - Incremental development: build only a bit of your program at a time, and test that bit thoroughly
    - Unit testing
    - Regression testing
  - Modularity
  - Encapsulation
    - Minimize the scope of variables
      - Always declare a loop control variable in the for-loop initializer
      - Declare a variable only when your first need it, and in the innermost curly-brace block that you can
      - Avoid global variable
- Debug systematically

Defensive Programming

- Protect your code from bugs
- Use assertion
- “An assertion is code that’s used during development—usually a routine or macro—that allows a program to check itself as it runs. When an assertion is true, that means everything is operating as expected. When it’s false, that means it has detected an unexpected error in the code” (Code Complete).

Assertions

- "An assertion usually takes two arguments: a boolean expression that describes the assumption that's supposed to be true and a message to display if it isn't."
- "Assertions are normally compiled into the code at development time and compiled out of the code for production."
- "During production, they are compiled out of the code so that the assertions don't degrade system performance."

(Code Complete)

Assertions contd.

- Never use assertion to test conditions that are external to your code such as the existence of files, the availability of the network etc.
- Assertions test the internal state of your program to ensure that it is within the bounds of its specification.
- Assertion failure therefore indicate bugs.
- External failures are not bugs, and there is no change you can make to your program in advance that will prevent them from happening.
- External failures should be handled using exceptions.

Code Contract

- "Code contracts provide a way to specify preconditions, postconditions, and object invariants in your code".
- The classes for code contracts can be found in the System.Diagnostics.Contracts namespace.

- Contract.requires()
- Contract.ensures()

The benefits of code contracts include the following:

- Improved testing: Code contracts provide static contract verification, runtime checking, and documentation generation.
- Automatic testing tools: You can use code contracts to generate more meaningful unit tests by filtering out meaningless test arguments that do not satisfy preconditions.
- Static verification: The static checker can decide whether there are any contract violations without running the program. It checks for implicit contracts, such as null dereferences and array bounds, and explicit contracts.
- Reference documentation: The documentation generator augments existing XML documentation files with contract information. There are also style sheets that can be used with Sandcastle so that the generated documentation pages have contract sections.

EXCEPTION

Exception

- Object-Oriented Exception handling
  - Representation of an error as an object
  - Classification of errors in class inheritance hierarchies

An error as an object

- Encapsulation of relevant error knowledge
  - Place of occurrence (class, method, line number)
  - Kind of error
  - Error message formulation
  - Call stack information
- Transportation of the error
  - From the place of origin to the place of handling
  - Via a special *throw* mechanism in the language

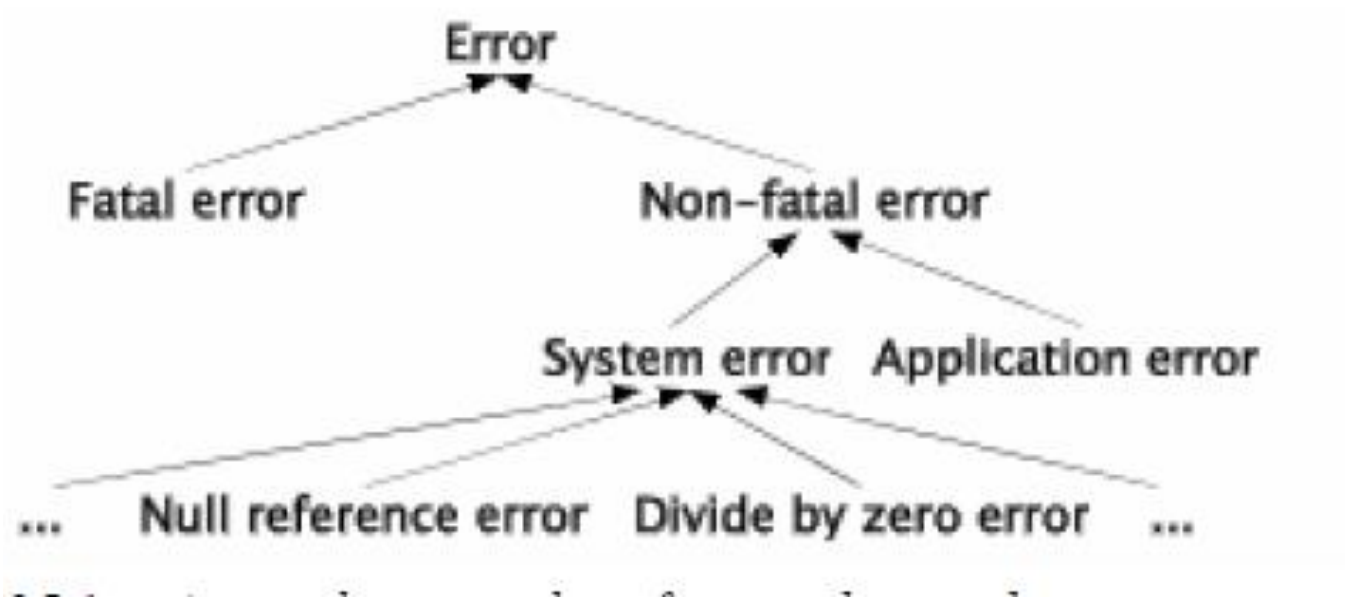
The hierarchy of Exceptions in C# (Exception class)

- ApplicationException
  - Your own exception types
- SystemException
  - ArgumentException
    - ArgumentNullException
    - ArgumentOutOfRangeException
  - DivideByZeroException
  - IndexOutOfRangeException
  - NullReferenceException
  - RankException
  - StackOverFlowException
  - IOException
    - EndOfStreamException
    - FileNotFoundException
    - FileLoadException

try-catch statement

- It allows us handle certain exceptions instead of stopping the program

```
try{
}
catch(exception-type-1 name){
}
Catch(exception-type- 2 name){
}
```


Error
Fatal error
Non-fatal error
System error
Application error
...
Null reference error
Divide by zero error
...

- When the CLR encounters an exceptional code, it halts execution and displays information about the error that occurs. This information is known as a *stack trace*.

Propagation of exceptions in C#

- Handling exception outside of where they occur.
- Handling exception where methods are invoked.

Check code

Try-catch-finally

- At least one catch or finally clause must appear in a try statement.
- The finally clause will be executed in all cases, both in case of errors, in case of error-free execution of try part, and in cases where control is passed out of try by means of a jumping command.

Raising and throwing exceptions in C#

A Throw statement

```
...
throw new MyException("Description of problem");
...
```

Definition of the exception class

```
class MyException: ApplicationException{
  public MyException(String problem):
    base(problem){
  }
}
```

Recommendations about exception handling

- Control flow
  - Do not use throw and try-catch as iterative or conditional control structures
  - Normal control flow should be done with normal control structures
- Efficiency
  - It is time consuming to throw an exception
  - It is more efficient to deal with the problem as a normal program aspect-if possible
- Naming
  - Suffix names of exception classes with "Exception"
- Exception class hierarchy
  - Your own exception classes should be subclasses of ApplicationException
  - Or alternatively (as of a more recent recommendation) of Exception

- Exception classes
  - Prefer predefined exception classes instead of programming your own exception classes
  - Consider specialization of existing and specific exception classes
- Catching
  - Do not catch exceptions for which there is no cure
  - Leave such exceptions to earlier (outer) parts of the call-chain
- Burying
  - Avoid empty handler exceptions – exception burying
  - If you touch an exception without handling it, always re-throw it.

- "Use try/catch blocks around code that can potentially generate an exception, and use a finally block to clean up resources, if necessary. This way, the try statement generates the exception, the catch statement handles the exception, and the finally statement closes or de-allocates resources whether or not an exception occurs".
- "In catch blocks, always order exceptions from the most specific to the least specific. This technique handles the specific exception before it is passed to a more general catch block".

Source: http://msdn.microsoft.com/en-us/library/seyhszts.aspx

Resources

- http://www.codeproject.com/Articles/3009/C-Documenting-and-Commenting
- https://msdn.microsoft.com/en-us/library/ms182532.aspx
- https://msdn.microsoft.com/en-us/library/dd264808.aspx