Telerik Academy

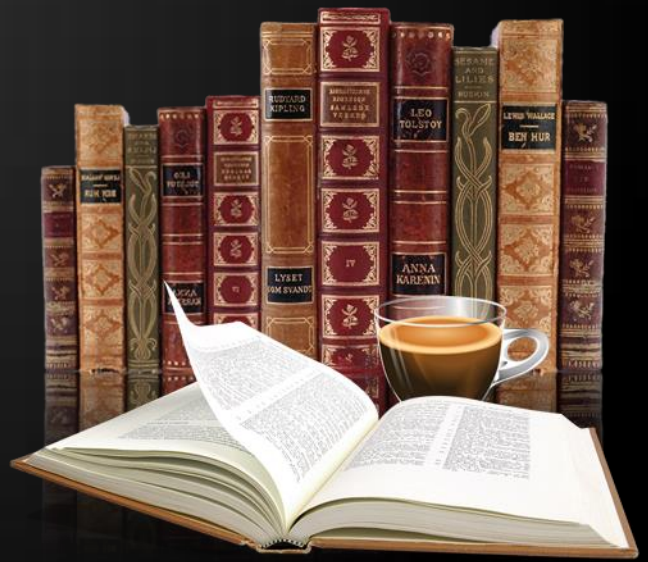
Defensive Programming, Assertions and Exceptions

Learn to Design Error Steady Code

Telerik Software Academy
Learning & Development
http://academy.telerik.com

Telerik Academy

Table of Contents



Telerik Academy

Defensive Programming

Using Assertions and Exceptions Correctly

What is Defensive Programming?

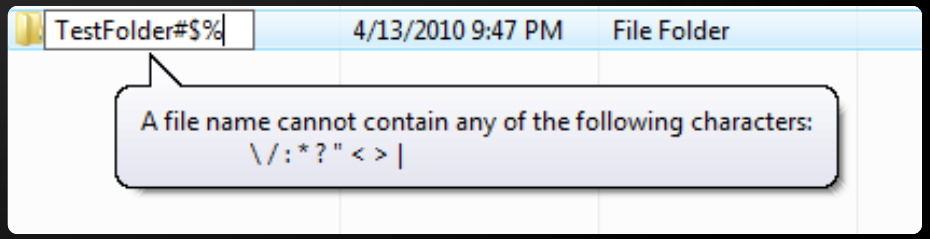
- Similar to defensive driving – you are never sure what other drivers will do

- Expect incorrect input and handle it correctly
- Think not only about the usual execution flow, but consider also unusual situations

Telerik Academy

Protecting from Invalid Input

- "Garbage in → garbage out" – Wrong!
  - Garbage in → nothing out / exception out / error message out / no garbage allowed in
- Check the values of all data from external sources (from user, file, internet, DB, etc.)



Telerik Academy

Protecting from Invalid Input (2)

- Check the values of all routine input parameters
- Decide how to handle bad inputs
  - Return neural value
  - Substitute with valid data
  - Throw an exception
  - Display error message, log it, etc.
- The best form of defensive coding is not inserting error at first place

Telerik Academy

Assertions

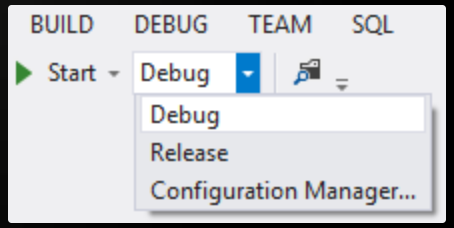
Checking Preconditions and Postconditions

Assertions

- Assertion – a statement placed in the code that must always be true at that moment

```
public double GetAverageStudentGrade()
{
  Debug.Assert(studentGrades.Count > 0,
     "Student grades are not initialized!");
  return studentGrades.Average();
}
```

- Assertions are used during development
  - Removed in release builds
- Assertions check for bugs in code



Assertions (2)

- Use assertions for conditions that should never occur in practice
  - Failed assertion indicates a fatal error in the program (usually unrecoverable)
- Use assertions to document assumptions made in code (preconditions & postconditions)

```
private Student GetRegisteredStudent(int id)
{
  Debug.Assert(id > 0);
  Student student = registeredStudents[id];
  Debug.Assert(student.IsRegistered);
}
```

Assertions (3)

- Failed assertion indicates a fatal error in the program (usually unrecoverable)
- Avoid putting executable code in assertions

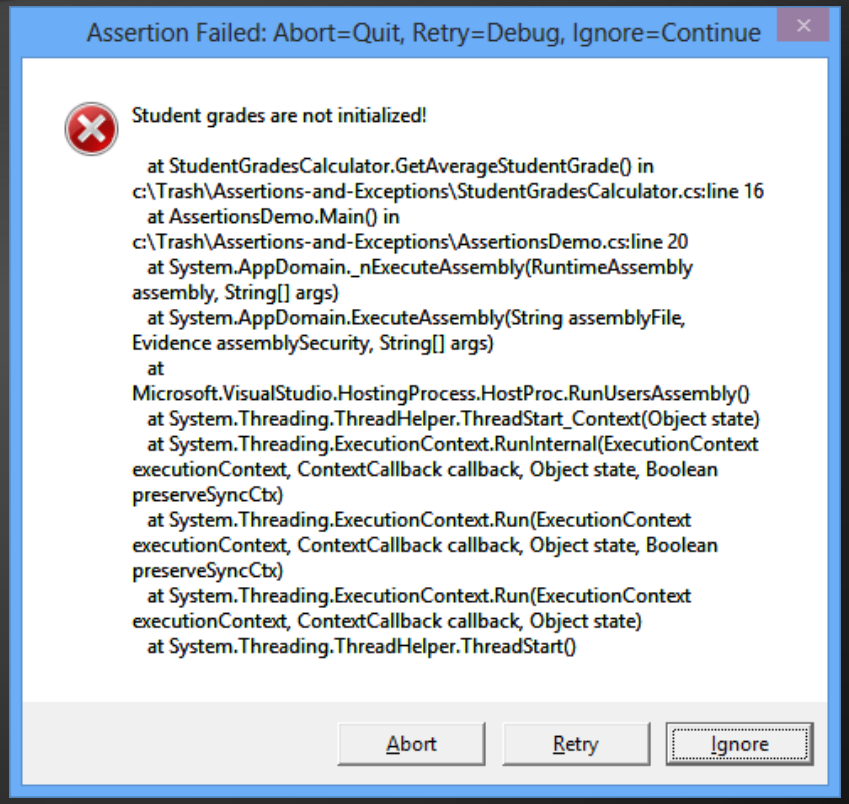
```
Debug.Assert(PerformAction(), "Could not perform action");
```

  - Won't be compiled in production. Better use:

```
bool actionPerformed = PerformAction();
Debug.Assert(actionPerformed, "Could not perform action");
```

- Assertions should fail loud
  - It is fatal error → total crash

Telerik Academy



Assertions

Live Demo

Telerik Academy

Exceptions

Best Practices for Exception Handling

Exceptions

- Exceptions provide a way to inform the caller about an error or exceptional events
  - Can be caught and processed by the callers
- Methods can throw exceptions:

```
public void ReadInput(string input)
{
  if (input == null)
  {
    throw new ArgumentNullException("input");
  }
  …
}
```

Exceptions (2)

- Use try-catch statement to handle exceptions:

```
void PlayNextTurn()
{
  try
  {
    readInput(input);
    …
  }
  catch (ArgumentException e)
  {
    Console.WriteLine("Invalid argument!");
  }
}
```

Exception thrown here

The code here will not be executed

- You can use multiple `catch` blocks to specify handlers for different exceptions
- Not handled exceptions propagate to the caller

Exceptions (3)

- Use finally block to execute code even if exception occurs (not supported in C++):

```
void PlayNextTurn()
{
  try
  {
    …
  }
  finally
  {
    Console.WriteLine("Hello from finally!");
  }
}
```

Exceptions can be eventually thrown here

The code here is always executed

- Perfect place to perform cleanup for any resources allocated in the try block

Exceptions (4)

- Use exceptions to notify the other parts of the program about errors
  - Errors that should not be ignored
- Throw an exception only for conditions that are truly exceptional
  - Should I throw an exception when I check for user name and password? → better return false
- Don't use exceptions as control flow mechanisms

Exceptions (5)

- Throw exceptions at the right level of abstraction

```
class Employee
{
  …
  public TaxId
  { get { throw new NullReferenceException(…); }
}
```

```
class Employee
{
  …
  public TaxId
  { get { throw new EmployeeDataNotAvailable(…); }
}
```

Exceptions (6)

- Use descriptive error messages
  - Incorrect example:

```
throw new Exception("Error!");
```

  - Example:

```
throw new ArgumentException("The speed should be a number " +
  "between " + MIN_SPEED + " and " + MAX_SPEED + ".");
```

  - Avoid empty catch blocks

```
try
{
  …
}
catch (Exception ex)
{
}
```

Exceptions (7)

- Always include the exception cause when throwing a new exception

```
try
{
  WithdrawMoney(account, amount);
}
catch (DatabaseException dbex)
{
  throw new WithdrawException(String.Format(
    "Can not withdraw the amount {0} from acoount {1}",
    amount, account), dbex);
}
```

We chain the original exception (the source of the problem)

Exceptions (8)

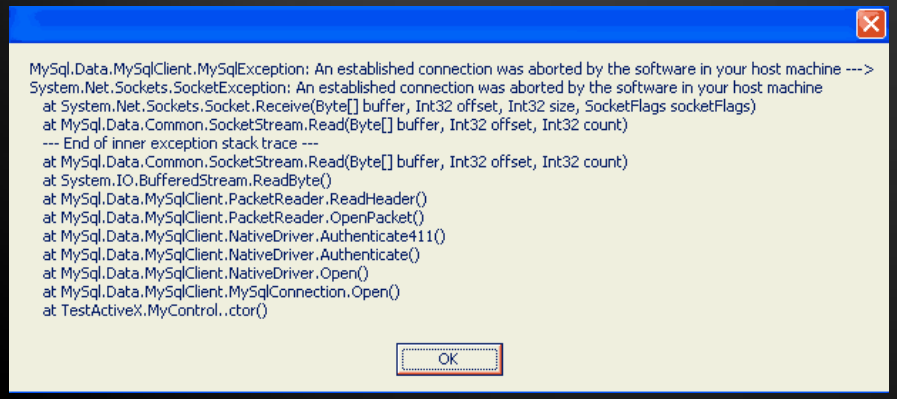
- Catch only exceptions that you are capable to process correctly
  - Do not catch all exceptions!
  - Incorrect example:

```
try
{
  ReadSomeFile();
}
catch
{
  Console.WriteLine("File not found!");
}
```

  - What about `OutOfMemoryException`?

Exceptions (9)

- Have an exception handling strategy for all unexpected / unhandled exceptions:
  - Consider logging (e.g. Log4Net)
  - Display to the end users only messages that they could understand



or

Telerik Academy

Exceptions

Live Demo

(Decompiling `System.DateTime`)

Telerik Academy

Error Handling Strategies

Assertions vs. Exceptions vs. Other Techniques



Error Handling Techniques

- How to handle errors that you expect to occur?
  - Depends on the situation:
    - Throw an exception (in OOP)
      - The most typical action you can do
    - Return a neutral value, e.g. `-1` in `IndexOf(…)`
    - Substitute the next piece of valid data (e.g. file)
    - Return the same answer as the previous time
    - Substitute the closest legal value
    - Return an error code (in old languages / APIs)
    - Display an error message in the UI
    - Call method / Log a warning message to a file
    - Crash / shutdown / reboot

Assertions vs. Exceptions

- Exceptions are announcements about error condition or unusual event
  - Inform the caller about error or exceptional event
  - Can be caught and application can continue working
- Assertions are fatal errors
  - Assertions always indicate bugs in the code
  - Can not be caught and processed
  - Application can't continue in case of failed assertion
- When in doubt → throw an exception

Assertions in C#

- Assertions in C# are rarely used
  - In C# prefer throwing an exception when the input data / internal object state are invalid
    - Exceptions are used in C# and Java instead of preconditions checking
  - Prefer using unit testing for testing the code instead of postconditions checking
- Assertions are popular in C / C++
  - Where exceptions & unit testing are not popular
- In JS there are no built-in assertion mechanism

Error Handling Strategy

- Choose your error handling strategy and follow it consistently
  - Assertions / exceptions / error codes / other
- In C#, .NET and OOP prefer using exceptions
  - Assertions are rarely used, only as additional checks for fatal error
  - Throw an exception for incorrect input / incorrect object state / invalid operation
- In JavaScript use exceptions: try-catch-finally
- In non-OOP languages use error codes

Robustness vs. Correctness

- How will you handle error while calculating single pixel color in a computer game?
- How will you handle error in financial software? Can you afford to lose money?
- Correctness == never returning wrong result
  - Try to achieve correctness as a primary goal
- Robustness == always trying to do something that will allow the software to keep running
  - Use as last resort, for non-critical errors

Assertions vs. Exceptions

```
public string Substring(string str, int startIndex, int length)
{
  if (str == null)
  {
    throw new NullReferenceException("Str is null.");
  }
  if (startIndex >= str.Length)
  {
    throw new ArgumentException(
      "Invalid startIndex:" + startIndex);
  }
  if (startIndex + count > str.Length)
  {
    throw new ArgumentException("Invalid length:" + length);
  }
  …
  Debug.Assert(result.Length == length);
}
```

Check the input and preconditions

Perform the method main logic

Check the postconditions

Error Barricades

- Barricade your program to stop the damage caused by incorrect data

- Consider same approach for class design
  - Public methods → validate the data
  - Private methods → assume the data is safe
  - Consider using exceptions for public methods and assertions for private

Being Defensive About Defensive Programming

- Too much defensive programming is not good
  - Strive for balance
- How much defensive programming to leave in production code?
  - Remove the code that results in hard crashes
  - Leave in code that checks for important errors
  - Log errors for your technical support personnel
  - See that the error messages you show are user-friendly

Telerik Academy

Defensive Programming

http://academy.telerik.com

Homework

- For the exercises use the Visual Studio solution "9. Assertions-and-Exceptions-Homework.zip".

1. Add assertions in the code from the project "`Assertions-Homework`" to ensure all possible preconditions and postconditions are checked.
2. Add exception handling (where missing) and refactor all incorrect error handling in the code from the "`Exceptions-Homework`" project to follow the best practices for using exceptions.

Telerik Academy

Free Trainings @ Telerik Academy

- C# Programming @ Telerik Academy
  - csharpfundamentals.telerik.com

- Telerik Software Academy
  - academy.telerik.com

- Telerik Academy @ Facebook
  - facebook.com/TelerikAcademy

- Telerik Software Academy Forums
  - forums.academy.telerik.com