Defensive programming. Exceptions. Optimizations

Defensive programming

- Using assertions and exceptions correctly

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

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.)

Protecting from Invalid Input

- 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

Assertions

- Check preconditions and postconditions

Assertions

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

```
public double getAverageGrade() {
	assert studentGrades.size > 0;
	return studentGrades.average();
}
```

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

Assertions

- 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)

```
Student getRegisteredStudent(int id) {
        assert id > 0;
        Student student = registeredStudents[id];
        assert student.isRegistered();

        return student;
}
```

Assertions

- Failed assertion indicates a fatal error in the program (usually unrecoverable)
- Avoid putting executable code in assertions, it won’t be compiled/executed in production
  - assert invokeAction();
- Assertions should fail loud
  - It is fatal error, total crash

Exceptions

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:
  - throw new IllegalArgumentsException();

Exceptions

- Use try-catch statement to handle exceptions:

```
try {
        // … code that can throw
        // this line possibly won't be executed if exception is thrown
} catch (IllegalArgumentException e) {
        // do something meaningful
}
// this line will be executed if catch block does not throw an exception
```

Exceptions

- You can use multiple `catch` blocks to specify handlers for different exceptions
- Not handled exceptions propagate to the caller
- Use finally block to execute code even if exception occurs (not supported in C++):
  - Perfect place to perform cleanup for any resources allocated in the try block

Exceptions

- 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? Or better return false?
- Don’t use exceptions as control flow mechanisms

Exceptions

- Throw exceptions at the right level of abstraction

```
class Student {
    double getAverageMark() {
        ///…
        throw new DBException();
        // or new StudentMarksCalculationExcetion();
}
```

Exceptions

- Use descriptive error messages
  - Incorrect example: throw new Exception(“Error”);
- Example: new Exception(“maxConnections should up to 10);
- AVOID empty catch blocks

Exceptions

- Always include the exception cause when throwing a new exception

```
class Student {
	double getAverageMark() {
		try {
			///…
		} catch (DBException e) {
			throw new StudentMarksCalculationExcetion(e);
		}
}
```

Exceptions

- Catch only exceptions that you can process correctly
- Do not catch all exceptions

Exceptions

- Have an exception handling strategy for all unexpected / unhandled exceptions:
  - Consider logging (e.g. SLF4J, logback, log4net, NLog)
  - Display to the end users only messages that they could understand

Error Handling Techniques

- 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)

Error Handling Techniques

- 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#/Java prefer throwing an exception when the input data / internal object state are(or will get) invalid
  - Exceptions are used in C# and Java instead of preconditions checking
  - Prefer using unit testing for testing the code instead of postconditions checking (postconditions checking can be a great Documentation!!!)
- 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, Java 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 non-OOP languages use error codes
- In JavaScript use exceptions: try-catch-finally

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

Error Barricades

- Barricade your program to stop the damage caused by incorrect data
  - Public methods/functions -> safe data -> private method
- 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 ones

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

Performance

What is Performance?

- Computer performance is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used
- An aspect of software quality that is important in human–computer interactions

Good Computer Performance

- Good computer performance:
  - Short response time for a given piece of work
  - High throughput (rate of processing work)
  - Low utilization of computing resource(s)
  - High availability of the computing system or application
  - Fast (or highly compact) data compression and decompression
  - High bandwidth / short data transmission time

Actual vs. Perceived Performance

- Example: *“Vista's file copy performance is noticeably worse than Windows XP”* – false:
  - Vista uses algorithm that perform better in most cases
  - Explorer waits 12 seconds before providing a copy duration estimate, which certainly provides no sense of smooth progress
  - The copy dialog is not dismissed until the write-behind thread has committed the data to disk, which means the copy is slowest at the end

Is Performance Really a Priority?

- Performance improvements can reduce readability and complexity
  - *Premature optimization is the root of all evil*
  - *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.*

How to Improve Performance?

- Software requirements
  – Software cost vs. performance
- System design
  – Performance-oriented architecture
  – Resource-reducing goals for individual subsystems, features, and classes
- Class and method design
  – Data structures and algorithms

How to Improve Performance?

- External Interactions
  - Operating system
  - External devices – storage, network, Internet
- Code Compilation / Code Execution
  - Compiler optimizations
- Hardware
  - Very often the cheapest way
- Code Tuning

Code Tuning

- What is code / performance tuning?
  - Modifying the code to make it run more efficiently (faster)
  - Not the most effective / cheapest way to improve performance
  - Often the code quality is decreased to increase the performance
- The 80 / 20 principle
  - 20% of a program's methods consume 80% of its execution time

Systematic Tuning – Steps

- Systematic code tuning follows these steps:
  1. Assess the problem and establish numeric values that categorize acceptable behavior
  2. Measure the performance of the system before modification
  3. Identify the part of the system that is critical for improving the performance
     - This is called the bottleneck
  4. Modify that part of the system to remove the bottleneck

Systematic Tuning – Steps

5. Measure the performance of the system after modification
6. If the modification makes the performance better, adopt it

   If the modification makes the performance worse, discard it

Code Tuning Myths

- "Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code"
  - For loop vs inlining it

Code Tuning Myths

- "A fast program is just as important as a correct one" → false!
  - The software should work correctly!

Code Tuning Myths

- "Certain operations are probably faster or smaller than others" → false!
  - E.g. "add" is faster than "multiply"
  - Always measure performance!
- "You should optimize as you go" → false!
  - It is hard to identify bottlenecks before a program is completely working
  - Focus on optimization detracts from other program objectives
  - Performance tuning breaks code quality!

When to Tune the Code?

- Use a high-quality design
  - Make the program right
  - Make it modular and easily modifiable
  - When it’s complete and correct, check the performance
- Consider compiler optimizations
- Measure, measure, measure
- Write clean code that’s easy to maintain
- Write unit tests before optimizing

When to Tune the Code?

- Junior developers think that "selection sort is slow"? Is this correct?
  - Answer: depends!
  - Think how many elements you sort
  - Is "selection sort" slow for 20 or 50 elements?
  - Is it slow for 1,000,000 elements?
  - Shall we rewrite the sorting if we sort 20 elements?
- Conclusion: never optimize unless the piece of code is proven to be a bottleneck!

Measurement

- Measure to find bottlenecks
- Measurements need to be precise
- Measurements need to be repeatable

Optimize in Iterations

- Measure improvement after each optimization
- If optimization does not improve performance → revert it
- Stop testing when you know the answer

Optimize in Iterations

- Profiler
- Trace, call stack

Do We Need Optimizations?

- Which language is fast?
- Is it worthwhile to benchmark programming constructs?
  - We should forget about small optimizations
    - Say about 97% of the time: premature optimization is the root of all evil
  - At all levels of performance optimization
    - You should be taking measurements on the changes you make

Benchmarking time

- Measure the time elapsed – Stopwatch in C#, currentTimeMillis() in Java
- Useful for micro-benchmarks optimization

Optimization Tips

- Static fields are faster than instance fields
- Instance methods are always slower than static methods
  - To call an instance method, the instance reference must be resolved first
  - Static methods do not use an instance reference
- It is faster to minimize method arguments
  - Even use constants in the called methods instead of passing them arguments
  - This causes less stack memory operations

Optimization Tips

- When you call a method in your program
  - The runtime allocates a separate memory region to store all local variable slots
  - This memory is allocated on the stack
  - Sometimes we can reuse the same variable
- Constants are fast
  - Constants are not assigned a memory region, but are instead considered values
  - Injected directly into the instruction stream

Optimization Tips

- The `switch` statement compiles in a different way than `if`-statements typically do
  - Some switches are faster than `if`-statements
- Using two-dimensional arrays is relatively slow
  - We can explicitly create a one-dimensional array and access it through arithmetic
  - The .NET Framework enables faster accesses to jagged arrays than to 2D arrays, but Jagged arrays may cause slower garbage collections

Optimization Tips

- `StringBuilder` can improve performance when appending strings
  - Using `char[]` may be the fastest way to build up a string
- If you can store your data in an array of bytes, this allows you to save memory
  - Smallest unit of addressable storage – byte
- Simple array `T[]` is always faster than `List<T>`
  - Using efficient data structures (e.g. `HashSet<T>` and `HashMap<K,T>`) may speed-up the code

Optimization Tips

- Use lazy evaluation (caching)

```
public int getSize() {
    if (this.size == 0) {
        this.size = calculateSize();
    }
    return this.size;
}
```

- for-loops are faster than foreach loops
  - foreach uses enumerator
- C# structs are slower (in most cases)
  - Structs are copied in their entirety on each function call or return value

Optimization Tips

- Instead of testing each case using logic, you can translate it through a lookup table
  - It eliminates costly branches in your code
- It is more efficient to work with a char instead of a single-char string
- Don't do unnecessary optimizations!
- Measure after each change

Which is the fastest?

```
string str = new string('a', 5000000);
int count = 0;
for (int i = 0; i < str.Length; i++)
  if (str[i] == 'a')
    count++;
```

```
int count = 0;
int len = str.Length;
for (int i = 0; i < len; i++)
  if (str[i] == 'a')
    count++;
```

```
int count = 0;
foreach (var ch in str)
  if (ch == 'a')
    count++;
```

Optimization Tips – Inline Code

- Manual or compiler optimization that replaces a method call with the method body
  - You can manually paste a method body into its call spot or let the compiler to decide
- Typically, inlined code improves performance in micro-benchmarks
  - ... but makes the code hard to maintain!
- In .NET you can force code inlining (with Attributes)

THANK YOU