“Minimalism isn't always the right choice,
but it's rarely the wrong choice”
Spartanization

What it is

• A refactoring method
• A powerful tool when used right
• Often in conflict with what you’ve learnt in MATAM

But it’s not

• A style convention
• A practice to adhere to at all costs
• Something to get worked up about
What are (some of) the reasons for complex code?

• Inability see all the code at once
  • Long methods
  • Many variables / method arguments
  • Error handling

• Complex state
  • mutability in general
  • Hidden side effects

• Complex controls
  • Nested If-statements
  • Complex Loops (Index iterations)
  • Conditional expressions are vague
Spartan Programming

• Principles:
  • Eliminate duplications
  • Minimize:
    • #lines
    • #columns
    • #tokens
    • #variables
    • #branches
    • #nested statements

• How?
  • Everything should be as short as possible, but not shorter
  • Minimize comments
  • Frugal use of variables
  • Small interfaces
  • Minimal use of control
  • Monadic Design
  • Think of code as Lines *spent* not lines written
Minimize comments

- Comments are *usually* a **code smell**
  - Your code should be understandable without reading walls of text
  - If it isn’t, make it so; if you can’t, ask yourself **why**
- Comments are a subtle form of code duplication
  - If the code changes, they should change too
  - But you don’t always remember to...
- Instead, use comments to explain decisions that are out of the ordinary, to ensure they don’t accidentally deleted or changed
  - `// the algorithm works better with a linked list than an array list`
  - `List<T> list = new LinkedList<>();`
- But! You should still use **javadoc** to document all **accessible** classes and methods that are part of your API
- Rule of thumb: **Javadoc**s explain **what**, **code** explains **how**, **comments** explain **why**
Frugal Use of Variables

- Minimize:
  - Number of variables: inline what is possible, minimize what cannot
    - “But naming a variable explains its meaning!” – if that variable is inlined, then there is nothing that needs to be explained
  - Life time of variable
    - Scope and context where can they be used
    - Prefer local variables to fields, scoped variables to local
  - **Everything should live in the smallest possible context**
    - Reduces namespace pollution
    - Can use extra context to reduce name size (see next slide)
Small Interfaces

- **Avoid Arrays!**
- **Minimize mutability**
  - Minimize side-affecting parameters
  - Parameters that can be mutated, or mutate the object accepting them
- **Parameter kind priority**
  - Input parameters
  - Output parameters
  - Input-output parameters
  - Pass by name (e.g. file names as Strings)
  - But you should have a really good reason why you are using the above three...
- **Minimize the number of method parameters**
  - Avoid flags, i.e., Boolean parameters
  - Use a method with a different name
  - `open()` and `close()` are better than `setOpened(boolean b)`
  - Avoid consecutive parameters of the same type
  - Prefer builder objects to method overloads
Small Interfaces –
Short method names

- Shorter method names are easier to remember and think about
- Prefer one word verb method names
  - e.g., map, filter, reduce
- Use two words to differentiate methods
  - e.g., map, flatMap; fold, foldLeft, foldRight;
- Methods that return a modified copy should be in the passive voice, e.g., sorted, reversed
- Add question modifier for methods returning boolean
  - e.g., isEmpty, hasValues
- Use class context to cut words
  - Url.getUrlAddress() vs. Url.getAddress()
Small Interfaces –
Short variable names

• The previous discussion also applies to variable and field names, but there is more to add...
• Use $ for a “special” variable
  • e.g., return value of a function, object undergoing test, temporary variable
• Use **generic names** whenever possible
  • If a method can accept any value, there’s no need for a long name
  • Compare:
    isPalindrome(String possiblePalindrome) with isPalindrome(String s)
    or isPrime(int number) with isPrime(int n)
• Use a **qualified name** if there are expectations:
  connectTo(String urlAddress)
• But prefer not to have any expectation! If you can’t use a generic name, always ask yourself “**why?**”
Minimal use of Control

- **Prefer declarative code to imperative code**
- Avoid conditionals
  - *Ternary operators* (? : ) are *evil* when composed
  - Avoid *else* as much as possible
  - Avoid complicated boolean expressions; avoid two or more connectors
- If possible, avoid *if* altogether
  - Use more OO approaches, such as polymorphism, or Monads
- Early exits:
  ```java
  if ($ != null) { // bad
    // code
  }
  if ($ == null) // good
    return;
  //code
  ``
- Using compact blocks and eliminating dead branches (i.e., ones that are impossible to reach) iteratively, we can greatly reduce the horizontal complexity of our program!
Minimal Control – Branches

- You can use code coverage tools to find *impossible paths*
- You can use Monads to replace if/else controls
- Before:

```java
public Double getLatInNorthernHemisphere() {
    Location l = getLocation();
    if (l == null || l.latitude < 0) {
        return null;
    }
    return l.latitude;
}
```

- After:

```java
public Optional<Double> getLatInNorthernHemisphere() {
    return getLocation()
        .map(l -> l.latitude)
        .filter(l -> l >= 0);
}
```
Minimal Control – Monads

- Avoid loops
  - Almost all looping constructs can be achieved using lambdas
  - Stream’s API is actually very thin compared to other languages, but you can wrap it to add missing functionality:
    - e.g., zip, take, fold, mkString
- Avoid nesting
  - Can usually be replaced with composing higher order functions
  - A Fluent API is a great help here...
- Do not use null except when explicitly required by legacy APIs
  - When returning a collection, an empty collection is better than null
    - Collections.emptySet(), Collections.emptyList()
  - Optional is far better and safer than a nullable value
- Question to consider: is it better to throw an exception or return an Optional?
  - What’s common should be easy, and what’s uncommon should be possible

Author: Gal Lalouche - Technion 2016 ©
Minimal Control – Exceptions

- Fail as early as possible on bad input
  - Avoid explicitly failing if possible, e.g., checked exceptions
  - But do not “fail silently”!
- Do not use exceptions as a control mechanism
  - e.g., try/catch instead of if/else
- Use try-with-resource statements (try in Java, using in C#) as an alternative to try/finally

```
try (BufferedReader br =
    new BufferedReader(new FileReader(path))) {
    return br.readLine();
}
```

- Do not use checked exceptions in a public API
  - 99% of the time, the API client will want to crash on error
  - The remaining 1% can be handled using try/catch anyway
  - Prefer a Try monad.
Careful use of Screen Space

- **K&R Style**
- Eliminate unnecessary parenthesis/braces
  - Yes, it’s okay to remove them if the statement is only one line
  - In fact, you should **strive** to make it only a single statement
    - Fluent API helps immensely
- **Refactor code into helper methods**
  - Shortens the primary method
  - But try to avoid short, single-use methods
- **“Hand rule”: Width, Height**
  - *You should be able to completely obscure a method by placing your hand on the screen.*
- **Know your standard library!**
  - Many common patterns are already implemented; there’s rarely a need to reinvent the wheel
- **Know your IDE!**
  - IDEs have many refactoring features
  - Don’t be afraid to learn the keyboard shortcuts
  - Mastering what your IDE has to offer will make your life easier!
Extra Reading

- GNU parser case study
- Spartan programming
- Succinctness is power – talks about programming languages, but it applies naturally to DSLs as well
- http://blog.codinghorror.com/spartan-programming/
- Eclipse plugin
- Writing code that is easy to delete