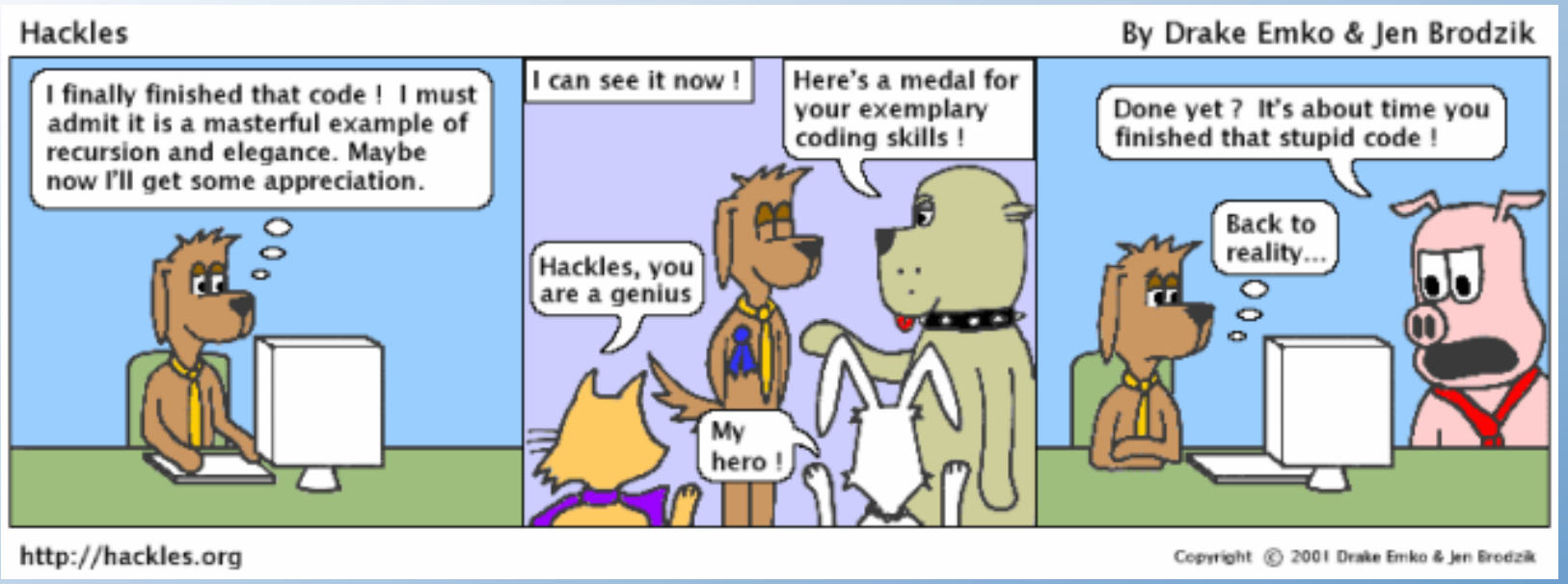
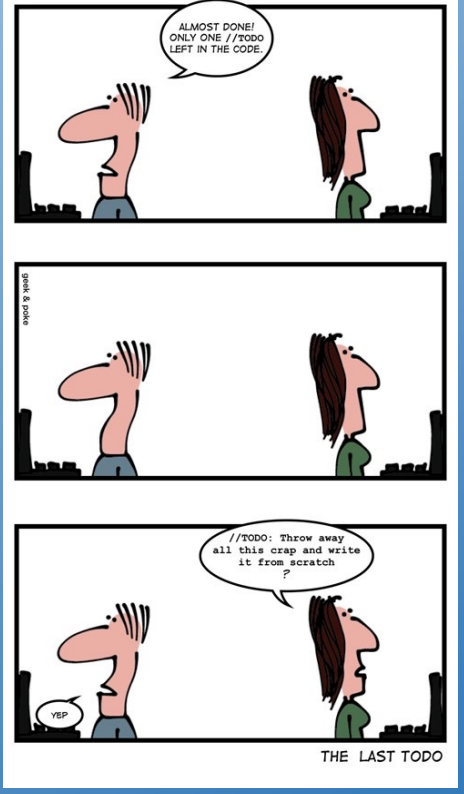
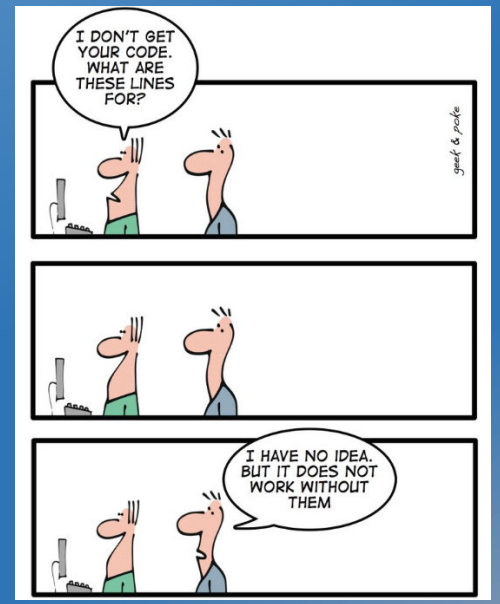


Coding

CSCI2340: Software Engineering of Large Systems

Steven P. Reiss

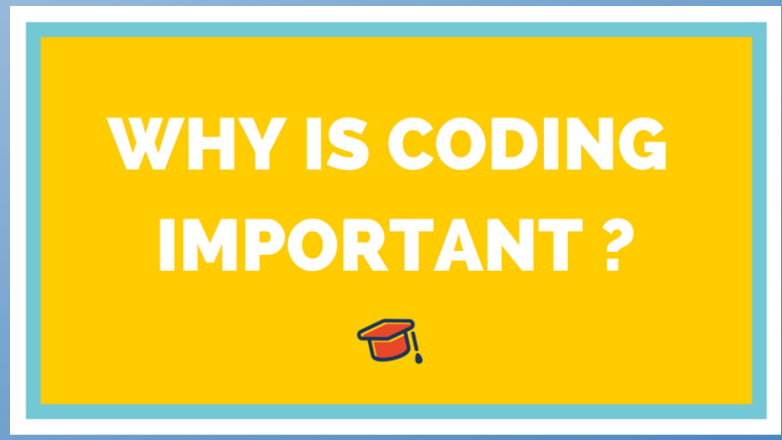




So You Think You Know How to Code

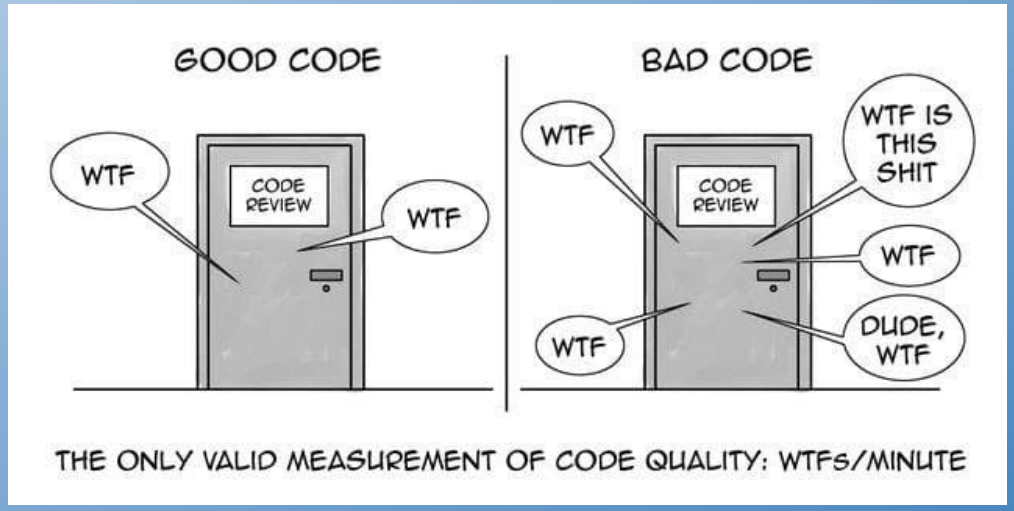
- You already know how to code
  - Or you wouldn't be here
  - You've been coding for a while
- This course should teach you to code better
  - Coding not for the moment
  - Coding for large, long-lived systems
  - Coding that will last
- I want you to use what you learned
  - In your project
  - In your programming assignment
  - In your future work

Coding is Important



- You will spend considerable time writing the code
  - Or it will seem that way
- You will spend a lot more time reading that code
  - Maybe not in this course, but in the real world
  - Also reading code that others wrote
  - For debugging, maintenance, evolution, understanding …
- This lecture tries to provide some general principles
  - Beyond the naming and ordering conventions we previously covered
- And example rules or suggestions for coding in Java and JavaScript
  - Other languages have similar rule sets
  - Find them on-line, or invent them yourselves
  - But make sure they are meaningful for you

Basic Principles



- Write code to be read by humans
  - Names should be meaningful
  - Code shouldn't be overly complex
  - Code should be documented where needed
  - Code should be organized
    - Finding things in the code should be easy
  - Conventions should be followed
    - Naming, ordering, style
  - Code should be easy to understand
    - Simpler is better

Basic Principles

- Take pride in your code
  - You should want to show it off
  - You should want others to read it
  - You should want others to emulate it
  - You should want others to use it

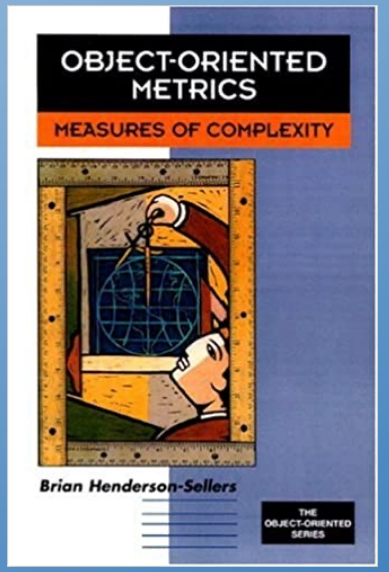
Basic Principles

- Write code meant to be permanent
  - Assume it will be around for 20 years
  - Even if you anticipate doing so, you won't throw it away
- Write code so it can be extended in the future
  - Code rarely shrinks
  - Classes will get additional methods/functions and fields
  - Classes will be used in different ways
  - Methods will get longer and more complex
  - Fields and variables will be added

Basic Principles

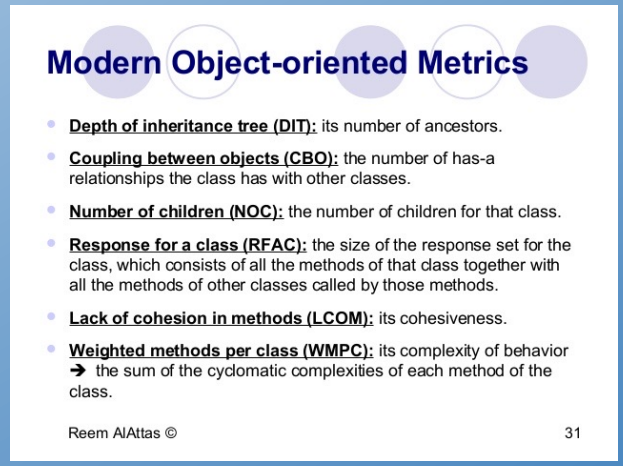
- Write code to be maintained
  - Ease of debugging is more important than ease of writing
    - Code defensively: it really helps
    - Ability to find a location given class/method name
      - This might be all you get from a stack trace or bug report
    - Avoid constructs that are difficult to debug and test
  - Ease of changing the code is more important than ease of writing
    - Easy to add new items
    - Easy to adapt, refactor
    - Localize as much as possible (principle of least privilege)
    - Minimize coupling, maximize cohesion
    - Document as needed

Making this Concrete: Classes



- Files and Classes should be reasonably sized
  - Not too long or too short: think about reading it
    - 200-1000 lines is best
- Keep the number of top-level classes in a package reasonable
  - Number of files in a directory
  - 5-20 would be a good target: think about finding something
  - Use inner classes, separate packages, or subpackages
- Inner classes should be small
  - If it is > 100 lines (1 page/screen), should probably be an outer class
- Make inner classes static, non-extendible classes final
- Classes and files shouldn't have too many public methods
  - Interfaces should be kept simple

Making this Concrete: Methods



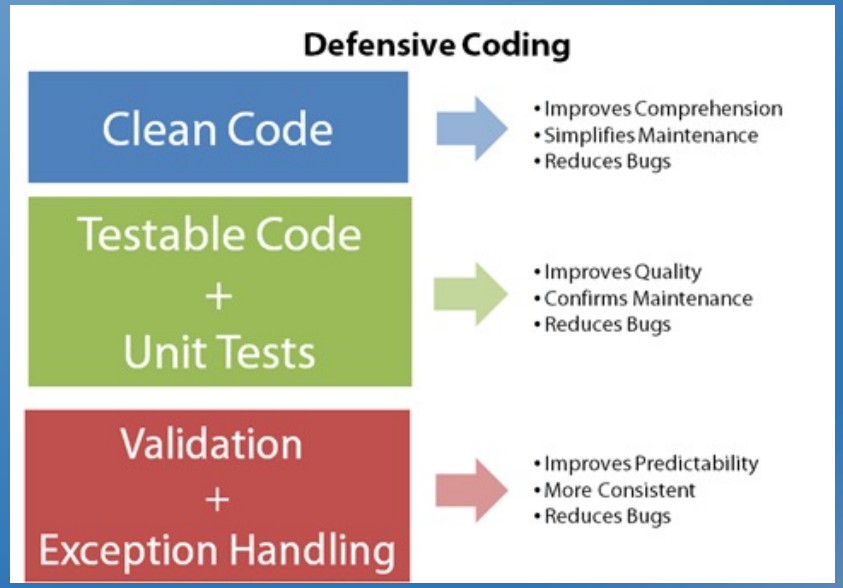
- Methods / Functions should be reasonably sized
  - **Should fit on one page or screen**
  - If more complex, split into multiple methods or use helpers
  - Or encapsulate in an inner class to provide common local variables
- Methods shouldn't have too many parameters
  - Especially accessible methods (public, protected, …)
  - Parameters should be necessary and logical
- Parameter types should be primitive where possible
  - Easier to understand, simpler to use
- Use a consistent parameter order throughout project
  - (width, height) versus (height, width)

Making this Concrete: Logging

- Use logging to track execution throughout coding
  - You'll eventually have to debug
  - Have the system capable of explaining what is happening
  - Logging finds bugs that you don't otherwise see
  - Add logging calls as you code as you write it
- Logging libraries exist
  - Java has one built in
  - Apache Commons Logging
  - Slf4j is another widely used one for Java
  - Easy to write your own as well
  - Be wary of bugs introduced in logging

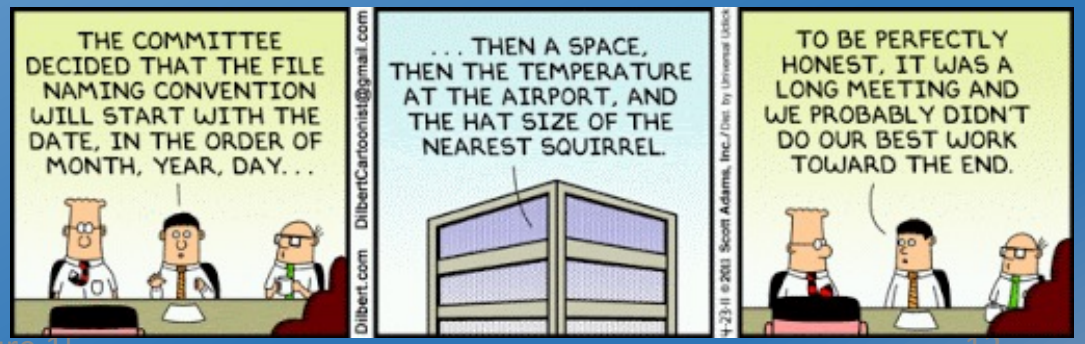
Making this Concrete: Defensive Coding

- Add defensive checks liberally as you write the code
- Fields should be private (prevent others from changing)
  - Keeps change effects local
- Public methods shouldn't trust their arguments
  - Check that parameters have reasonable values
  - Before using them
- Check output is reasonable and expected
  - When testing, debugging, using the system
  - After calling external methods
  - Catch errors early
  - Always look at log output
- Make all assumptions explicit
  - Either in the code or in comments
  - Or both



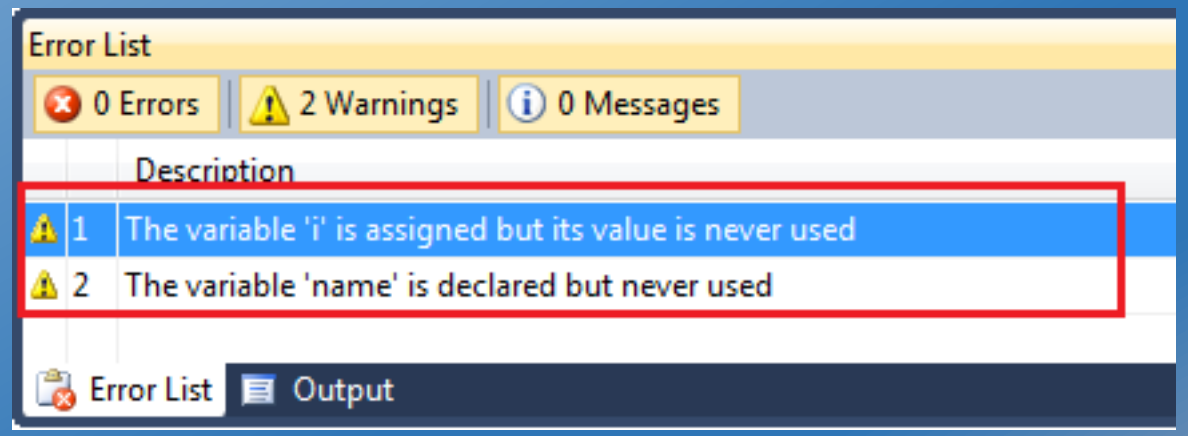
Making this Concrete: Conventions

- Use a consistent and complete set of conventions
  - Standard conventions (e.g., Sun, Google)
    - Quite extensive, but still don't cover everything
    - Might be overly restrictive for your purposes
  - Use naming conventions to your advantage
    - Class name should identify the package and where to find it
    - Looking at an identifier, you should be able to find its definition
    - Avoid the possibility of name conflicts (internal and external)
  - Use file ordering conventions to your advantage
    - Know where in a file to look for things (fields, private methods, …)
    - Formatting should be consistent throughout the project
    - Split files into sections with blocks

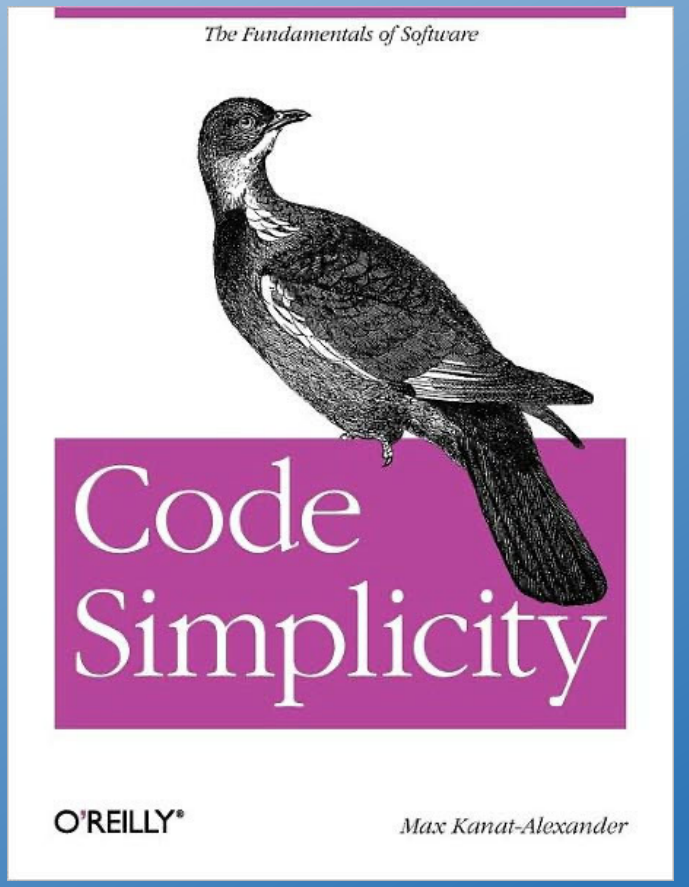


Making this Concrete: Compilation

- Choose a reasonable set of compiler warnings
  - IDE-based compilers make this easy
  - Warnings are there for a reason – better code
- Ensure code compiles without any warnings
  - You can use @SuppressWarnings if needed
  - But you should justify its use



Making this Concrete: Simplicity

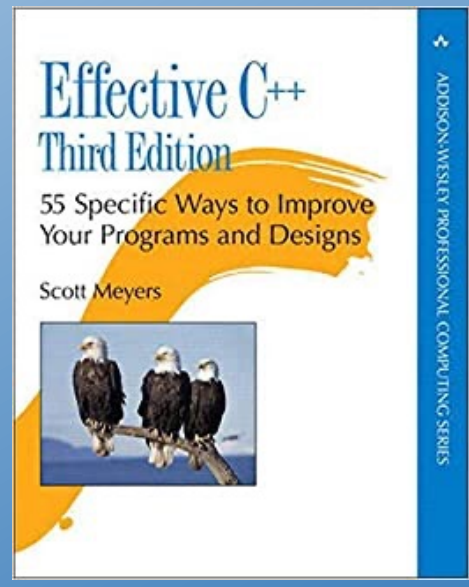


- Keep the code simple
  - Efficiency probably isn't the primary concern
  - Nor is conciseness or minimizing initial typing
  - Target ease of reading, understanding & debugging
- Comment anything that might not be obvious
  - To a programmer unfamiliar with the details
  - To yourself 5 years hence
- Ensure code can be understood from the comments
  - And the method and variable names
  - Without having to read unnecessary details
  - Be able to quickly find the relevant portion of a function or method

Use a Style Checker

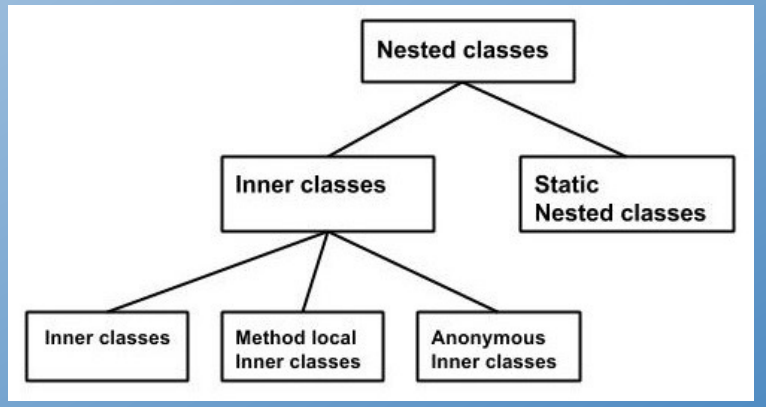
- For Java: use checkstyle
  - Either start with Sun or Google standards
    - Change these to meet your needs and conventions
  - Or go through the various checks and set up or ignore each
    - Justify your decision
  - Should have no checkstyle warnings when done
    - Use @SuppressWarnings when needed and justified
- For other languages
  - C/C++: lint (original style checker)
  - Dart: language enforces some style conventions
  - TypeScript/JavaScript: ESLint
  - Checkers exist for most languages

Making this Concrete: Language



- Every language has its quirks
  - Some aspects are for ease of coding, not ease of reading
  - Other aspects might be hard to understand
    - What happens at run time needs to be obvious to the reader
  - Other aspects might make debugging difficult
  - Other aspects might make evolution difficult
- I generally develop a set of rules for using a language
  - That emphasizes readability, maintenance, debugging, evolution
  - Features not to use in the language
  - Features to use that might otherwise be overlooked
  - Based on experience
- You can generally find language usage guidelines
  - For most languages, either in books or on the web

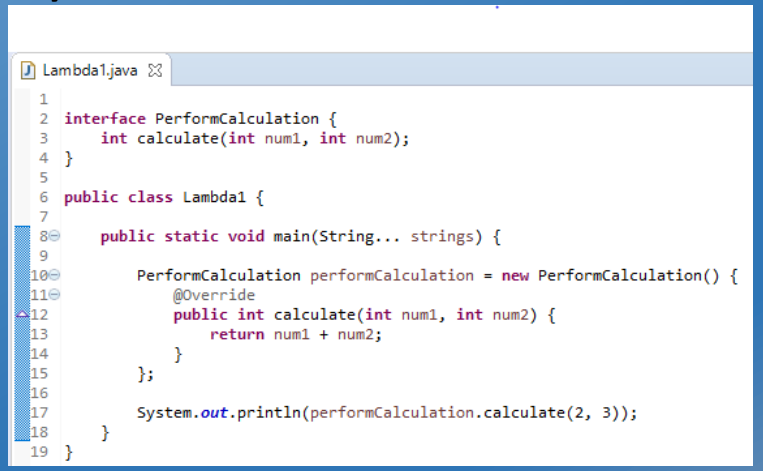
Java Guidelines: Inner classes



- Use inner classes
- Use inner classes to hide implementation details
  - Inner classes should not be exposed directly
  - Inner interfaces in a common interface might be exposed
  - Can enclose an algorithm with its own set of global variables
  - But can implement global interfaces
- Inner classes are likely to become outer classes
  - Name and code accordingly
- Avoid nesting inner classes
- Inner classes should be static if possible
- Inner classes should be private

Java Guidelines: Anonymous Classes

- Do not use anonymous or method-local classes
  - Class outername$10 tells you nothing while debugging
    - Finding the code is difficult
  - Anonymous classes will often grow and become cumbersome
    - Will need to become inner or outer classes eventually
    - Why not make it an inner class initially
    - This gives a reasonable name, file location
    - Makes it easier to extend
  - Understanding variable references is difficult
    - Changing variables can have unintended effects

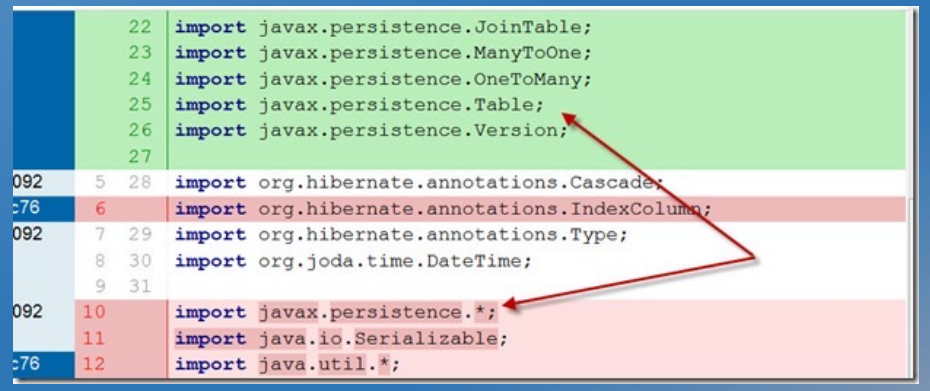


Java Guidelines: Hierarchies

- Embed simple hierarchies as inner classes
  - Inside the exposed (abstract) (root) class of the hierarchy
  - Assumes hierarchy itself is not exposed
    - This is an implementation detail
  - Inner classes must be static in this case
- Only expose a hierarchy if necessary
  - Prefer interface-based hierarchies
  - Allows hierarchy to evolve and change
  - Only the root should be a visible external type
- Provide a visitor for hierarchical structures

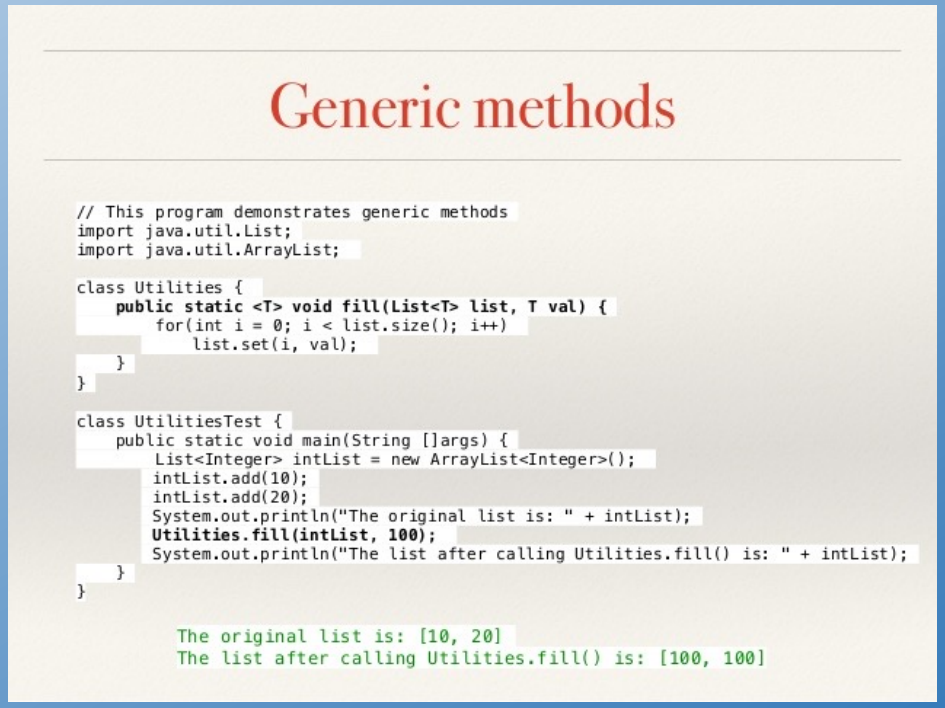
Java Guidelines: Imports

- Don’t use on-demand imports
  - IDEs will add necessary imports, fix up import lists, etc.
- Use static imports only when names will be unambiguous
  - With your code, external libraries, other imports, etc.
- Use imports to access inner components of an interface
  - When those components are named appropriately
  - To avoid name conflicts
  - Otherwise use Outer.Inner notation

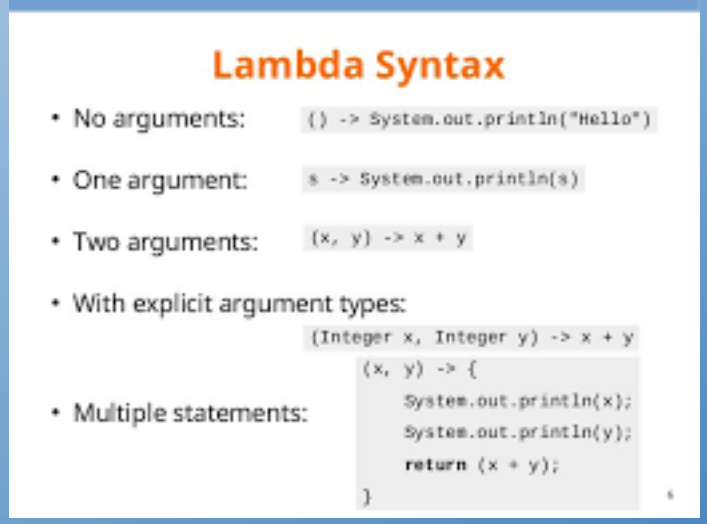


Java Guidelines: Generics

- Get as strong typing as you can
- Use generics for all collections, etc.
- Don’t use Object
  - As a method parameter
  - Unless you really mean an object with multiple simultaneous types
- Learn how to create your own generic classes
  - Template rather than cast
- Learn how to create your own generic methods
  - Where the output type and maybe input types depends on parameters
  - But ensure these are easy to understand and use



Java Guidelines: Lambdas



- Use lambdas sparingly if at all
  - Never use untyped lambdas
  - Difficult to debug – where is lambda$5 in the code?
  - Environment is unclear at run time
  - Problems similar to anonymous classes (and more)
- Implicit typing can be complex and confusing
  - Make sure it is all explicit
  - Prevents future problems
- Be wary of method references
  - Impossible to debug

Annotations in Java -- Definitions

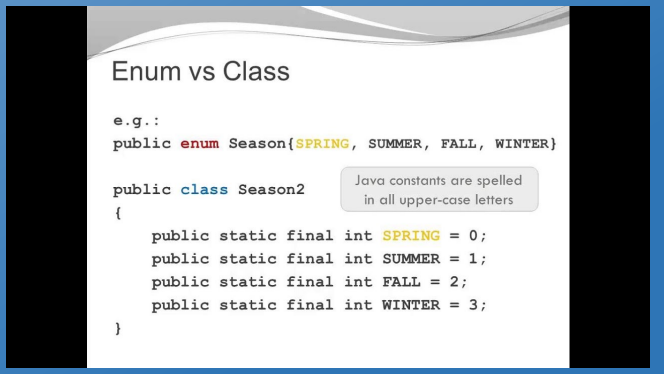
- Understand the use of annotations in the language
  - Built-in Annotations (e.g., @Override, @SuppressWarnings)
  - Checking annotations (e.g., @Nullable, @NonNull)
  - User defined annotations
- Compile-time annotation Processing
  - Augment or convert the code
  - But not in the way you would expect
- Dynamic annotation processing
  - Effectively augment the code using reflection at run time
  - Used by Spring and other frameworks
  - This means what you read is not what you execute

Java Guidelines: Annotations

- Use built-in annotations (@Overrides)
- Use checking annotations if available and **enforced**
  - Otherwise, they can be misleading
- Don’t use an annotation processor or dynamic annotations
  - Seem convenient, but more of a hinderance than a help in the long run
  - Difficult to understand what is going on
    - Unless you are intimately familiar with the annotation library
    - And you won’t be in 5 years
  - Dynamics have no source for debugging, extending
  - Can violate principle of least privilege (@Getter)
  - Don’t work well with static analysis tools
  - Note these are part of the Spring & other frameworks

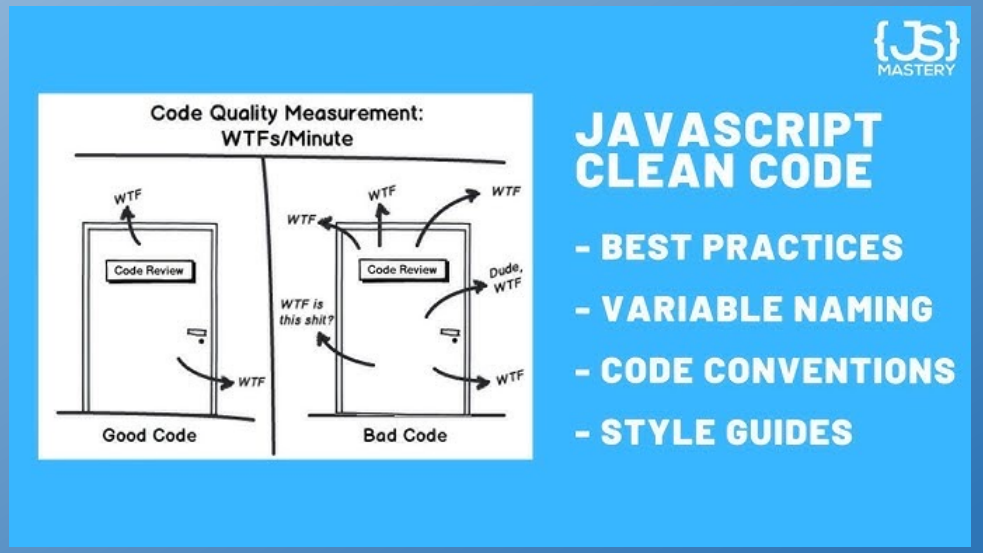
Java Guidelines: Other

- Use enumerations rather than integer constants
- Avoid using records (implicitly make things public)
- Only this ‘this.’ where necessary
- Initialize in the constructor
  - Not in non-static field declarations
  - Initialize all fields
- Don’t depend on default initializations
- Don’t get ”cute”
  - if (x*y == 0) { } versus if (x == 0 || y == 0) { }



JavaScript/TypeScript Guidelines

- Avoid nested function definitions
  - Especially those > 1 line (a simple call)
  - Preferred: use async/await
    - Code looks sequential, understandable
  - Alternative: use futures
    - But arguments are odd & not as clear to read
  - Alternative: use separate functions
- Avoid lambdas (use named functions)
- Use separate routings with Express
  - One for each component that handles URLs
  - Rather than a master router with all routing in it
- Google JavaScript style guide (overkill)



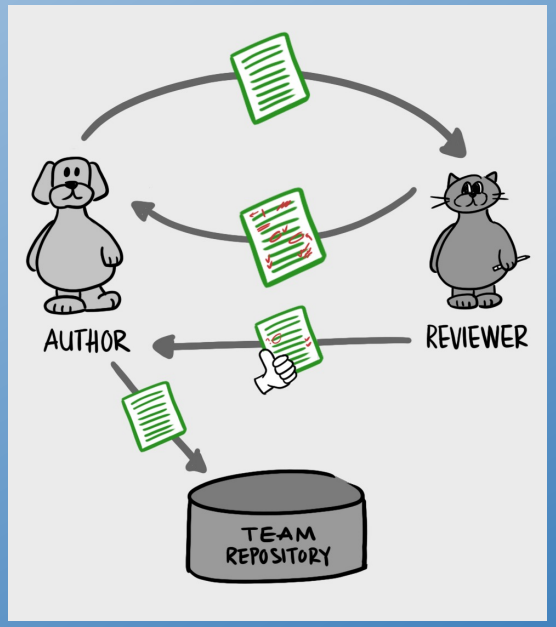
Dart Guidelines

- Avoid nested function definitions
- Avoid lambdas
- Avoid complex nested widget definitions
  - Define the components first, then the widget
- Separate the system into logical directories
  - Treat directories as packages

Code Reviews

- Looking at code off-line
- Have several uses
  - Code style checking for project consistency
  - As a debugging tool
  - As a testing tool
- Take a piece of code (method, class, …)
  - Have a panel of reviewers
    - Can be individuals working separately or a group meeting
  - Pass the code out to the panel (possibly in advance)
  - Panel goes over the code line-by-line
- Goal is to find and eliminate all potential problems
  - Make sure the code conforms to various guidelines
  - Find potential bugs
  - Ask about what-if questions to ensure al possibilities considered

Code Reviews



- Look For
  - Style violations
  - Language usage violations
  - How readable and understandable is the code
- Simulating the execution to see if it works
  - Find possible inputs and conditions that might cause problems
  - Ask what-if questions
- You can do code reviews on your own
  - Read over your own code critically (like proofreading)
    - This is a good idea in general
    - Read it as you write; read it after it is written; read it as you type it
  - Not as effective as having others do it

EXERCISE

- Code base (handout)
- Go through the code and note what you think should be changed
  - To make it follow conventions
  - To make it more readable
  - To make it the way you might code it
  - To make it easier to maintain/evolve/debug/…
  - To make it better (you can always improve your code)
  - You should be able to find at least one thing to correct, possibly several
- Note what changes you would make
  - And then we will go over them as a class
- We'll do this again for debugging next time (different code)
- We'll do this with your project code later on

HOMEWORK

- Added feature due Thursday; Video of your program due next Tuesday
- Explicitly state coding standards for you project (as a team)
  - Add this to your GitHub repo (coding styles should be there already)
  - Ensure the code meets these standards
  - Set up eslint, checkstyle or similar tool for your project
- Explicitly state the coding standards
  - For the language you use for your assignment
  - Can be the same as above if the same language
- Make sure your programming assignment conforms
  - To these standards
  - And meets the other coding criteria mentioned today
  - Clean it up if not
- Optional
  - Pair up and do a code review of your programming assignment

PROJECT

- Ensure that each person can go off and code their portion of the system on their own.
- Project status reports in class next Tuesday (after or interspersed with program videos)
- Project meeting?