

Software Construction Practices

Why Standards Matter

- code is read far more often than it is written
- a standard isn't about taste
 - it is about making a codebase predictable enough that you don't have to re-learning how to read it for every new file
- tools can enforce some of this automatically
 - e.g. Eclipse auto-format
 - e.g. linters

Naming Conventions

- official Java conventions
 - `ClassName` (Pascal case or upper camel case)
 - `methodName / variableName` (camel case)
 - `CONSTANT_NAME` (screaming snake case)
- other conventions – historical or in other languages
 - `method_name / variable_name` (snake case)
 - leading underscore for private instance variables and to avoid variable shadowing
 - trailing underscore to distinguish instance variables, avoid keyword clashes, avoid variable shadowing
- a name should describe *what*, not *how*
- avoid –
 - abbreviations that aren't obvious
 - single-letter names outside of trivial loop counters
 - exception is allowable if there's a strong well-known convention
 - names that lie about what they hold

Style and Formatting

- a style guide is a written, agreed-on set of choices
 - formatting style – indentation, brace placement, line length, blank-line spacing
 - naming conventions
 - programming practices
- e.g. Google's Java Style Guide
 - <https://google.github.io/styleguide/javaguide.html>
- pick a style and apply it consistently across a whole file or project
 - agree on a style *before* writing code, not after
 - specific style choices generally matter far less than consistency

Documentation – Beyond Javadoc Syntax

- what to document
 - a method's contract – what it promises, what it assumes, what it doesn't handle
- what *not* to document
 - what is already obvious from a well-chosen name
 - simply restating the code line by line
 - a comment that just repeats code is worse than no comment – it can go stale and become misleading
- what style to document in
 - Javadoc comments

Defensive Programming

- *defensive programming* refers to writing code that behaves predictably even when given unexpected input
- core concepts
 - *assume nothing* – treat all external data, user actions, and API responses as untrustworthy or potentially hostile
 - *fail safely* – design the system to handle errors gracefully instead of crashing
 - *fail fast* – check for invalid conditions immediately at the start of a function and stop execution before deeper damage occurs

Defensive Programming

- check the result before returning it
 - check postconditions – is the result in the right range, are invariants maintained
- add an explicit branch for the case you don't expect to happen
 - add a final else to if/else if to throw an `IllegalArgumentException` or log rather than ignoring
- copy mutable data on the way in and on the way out
 - if a constructor or setter takes a mutable object (like a `List`), copy it into a new instance before storing – don't store the reference
 - when returning an internal collection from a getter, return a copy or wrap it in a readonly view (e.g. `Collections.unmodifiableList(...)`) – don't return the live reference

Defensive Programming

- validate input harder at a trust boundary, lighter internally
 - check preconditions and more thoroughly check parameters to public methods
 - skip redundant re-validation for parameters to private methods
- replace a raw type with a type that can't be wrong
 - use enum instead of a `String` or `int` constants when a value has a fixed set of legal options
- never leave a catch block empty
 - at a minimum, log that something occurred
 - avoid a generic catch (`Exception e`) – catch the specific exception type(s) expected

Defensive Programming

- add a bound to anything that loops or waits on external input
 - give a network call a timeout parameter instead of letting it wait indefinitely
 - add a max iteration count or depth to a loop or recursive call whose bound is determined by external/untrusted input