Imperative Programming 3

Style

Peter Jeavons

Trinity Term 2019

The Challenges of Building Software

- Software complexity grows fast
  - at least exponentially in number of branches
- Cannot rely only on testing and proofs
  - we need principles, discipline, and processes
- Building software is a team effort
  - think of code as write-once / read-many
  - e.g., code reviewed by peers

Fighting Complexity

- Use OO design principles
- Classes: smallest organizational units
  - the smallest piece of a system with behavior
  - we have applied design principles mainly to classes

$\Rightarrow$ We need more coarse organizational units!

- classes are grouped into subsystems
- subsystems are grouped into larger (sub)systems

$\Rightarrow$ We should apply the same design principles at all these levels!

Big Idea: Chunking

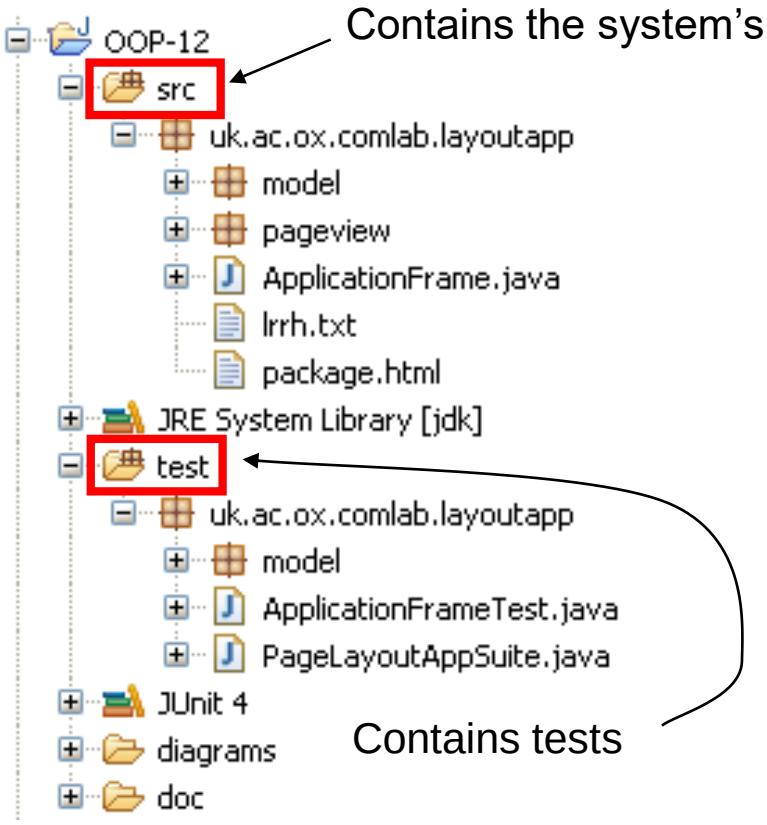
Organizing Classes into Packages

- Package: contains a set of “related” classes
- The meaning of “related” varies:
  - classes that logically form a subsystem
  - classes that are loosely connected
  - classes that are used in many places (AKA utility packages)

  $\Rightarrow$ designer must know what a package represents
- Packages also have interfaces
  - public classes can be used outside the package
  - package classes can be used only inside the package

Code Organization and Directories

- Directories also allow for code organization
  - orthogonal to packages



- Both `src` and `test` directories contain the same packages
  - e.g., `uk.ac.ox.comlab.layoutapp.model`
  - needed for white-box testing
- This separation allows us to compile the system and the tests independently
  - when shipping the system, we do not want to ship the test code
  - when working on tests, we do not want to always recompile system's code

Classes and Files

- Separate files for interface / implementations
  - one single logical compilation unit (class/trait/object) per file
  - exceptions in Scala: companion objects and sealed traits with several sub-classes
- Name file after the name of the class
  - Java already forces you to do all these = good
  - use upper camel case for class names (e.g., `HashMyMap`)
- Within a file, separate methods clearly
- Length of line ≤ 120 characters
  - let 120 be the exception, aim for ≤ 80 in the common case
- Length of class ≤ 2,000 lines
  - this is a rough guideline, aim for fewer lines if you can

Big Idea: Readability

Readability

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability."

John F. Woods

Conventions

- Choose and follow a coding convention
  - indentation, spaces, formatting…
  - e.g., Scala style: http://docs.scala-lang.org/style/overview
  - one convention is better than none or many
    - adds structure and discipline to the code
    - provides common language for code writers and readers
- Use an IDE; heed all warnings
  - retrofitting code to another coding style is very hard

- Consistency is the mother of organization
  - you know what to look for and where ⇒ leads to higher efficiency

Class Layout

1. Header comment
2. Class data
3. Public methods
4. Protected methods
5. Private methods

```
/**
* Hash table based implementation of the <tt>MyMap</tt>
* interface. This implementation provides all of
* the optional map operations, and permits
* ...
*/
class HashMap[K,V] extends AbstractMap[K, V] {
  private val table: Array[Entry] = ...
  // ...
  this(initCapacity: Int, loadFactor: Float) = {
    this()
    if (initCapacity < 0)
      throw new IllegalArgumentException(...)
    // ...
  }
  def +(kv: (K,V)): AbstractMap[K,V] = {
    // ...
  }
  // ...
  protected def getStats: Statistics = {
    // ...
  }
  // ...
  private def resize(newCapacity: Int) = {
    assert(newCapacity > table.length ||
           table.length == MAXIMUM_CAPACITY)
   // ...
  }
  // ...
}
```

Whitespace

- Whitespace is the key to understandable text
  - spaces, tabs, line breaks, blank lines
- Whitespace provides the basis for grouping
- Indentation suggests logical structure
  - aim for balance: 2-4 spaces is optimal [Miaria et al. 1983]
- Avoid deceit – tell same story to human as to computer

```
arrayLength = 3+4 * 2+7;
// swap left and right elements for whole array
for (i = 0; i < arrayLength; ++i)
    leftElement = left[i];
    left[i] = right[i];
    right[i] = leftElement;
```

parentheses can improve readability

Self-Documenting Code

- Think very deeply about naming schemes!
  - Good naming allows you to apply your intuition

Variable Names

- Keep in mind to read-optimize, not write-optimize

Purpose of Variable	Good Names, Good Descriptors	Bad Names, Poor Descriptors
Running total of checks written to date	*runningTotal, checkTotal*	*written, ct, checks, CHKTTL, x, x1, x2*
Velocity of a bullet train	*velocity, trainVelocity, velocityInMph*	*velt, v, tv, x, x1, x2, train*
Current date	*currentDate, todaysDate*	*cd, current, c, x, x1, x2, date*
Lines per page	*linesPerPage*	*lpp, lines, l, x, x1, x2*

- Use a single natural language for project
- Names to avoid:
  - words that sound similar (`wrap` vs. `rap`)
  - names should differ in at least two characters
  - avoid numerals (`file1`, `file2`, …)
  - avoid commonly misspelled words (`acummulate`, `independant`, `reciept`, `calender`)

Variable Names (2)

- Loop indexes:
  - customary `i`, `j`, `k`, BUT use meaningful name when outside the loop
- Computed-value qualifiers
  - frequently used: Total, Sum, Average, Max, Min, etc.
  - place at end of variable name: usersTotal, consumptionAverage...
- Use common opposites
  - begin/end, min/max, next/previous, source/target, source/destination, up/down
- Status variables
  - Avoid using *flag* in the name
  - Use explicit variable names instead

```
if (flag) ...                          BAD
if (statusFlag & 0x0F) ...             BAD
if (dataReady)…                        OK
if (characterType & PRINTABLE_CHAR)    OK
```

- Boolean variables
  - good names: done, error, found, success, ok, ... (can only take on true/false)
  - bad names: status, sourceFile, ...
  - using the "is" prefix protects you from bad names (isStatus... ?)
  - favour using positive boolean names (negatives are cumbersome to negate)

Comments

> “Don’t comment bad code – rewrite it.”
>
> *Kernighan & Plauger*
>
> *Elements of programming style (1978)*

- Natural language is less precise than code
  - don't rely on comments for correct reading
  - use comments simply to make reading faster
  - acts like headings in a book / table of contents
- Good comments clarify code's intent – do not repeat the code itself
- Bad comments can do more harm than good
  - disagreement between comments and code is nasty
- Code reviews should also check comments
- Strive for balance
  - too many comments can reduce understandability
- Random trivia: LOC count does not include comments

Types of Comments

- Summary of code
  - OK
- Describe code intent
  - Good
- Repeat the code
  - Useless
- Explain the code
  - Usually indicates confused code → you are better off fixing the code
- Tag the code
  - use standard markers (e.g., FIXME, TODO, “fixes BUG #4554”)

Comment *What*, not *How*

```
// check each character in "inputString" until a dollar
// sign is found or all characters have been checked
done = false
maxLen = inputString.length
i = 0
while (!done && i < maxLen) {
  if (inputString(i) == '$') {
    done = true
  } else {
    i = i + 1
  }
}
```

```
// find the command-word terminator
```

```
// if account flag is zero
if (accountFlag == 0) {
  ...
}
// if establishing a new account
if (accountFlag == 0) {
  ...
}
// if establishing a new account
if (accountType == accountType.NewAccount) {
  ...
}
```

Warn the Reader

- Prepare code reader for what is to follow
  - comment should precede the code it refers to
- Document any "workarounds”

```
/* The SL811 has a hardware flaw when hub devices send out
SE0 between packets. It has been found in a TI chipset and
Cypress hub chipset. It causes the SL811 to hang
The workaround is to issue the preamble again.
*/
if (cmd & SL11H_HCTLMASK_PREAMBLE) {
    SL811Write (hci, SL11H_PIDEPREG_B, 0xc0);
    SL811Write (hci, SL11H_HOSTCTLREG_B, 0x1)    // send preamble
}
```

Recap

"Programs must be written for people to read, and only incidentally for machines to execute."

Abelson and Sussman

- Layout & comments done well can help a lot
  - good layout illuminates the logical structure of the program
- Don't rely on comments to make obscure code clearer
  - comments should say things the code cannot say
- Your code and comments are your messenger
  - whoever reads them should want to thank you

Big Idea: Documentation

Importance of Documentation

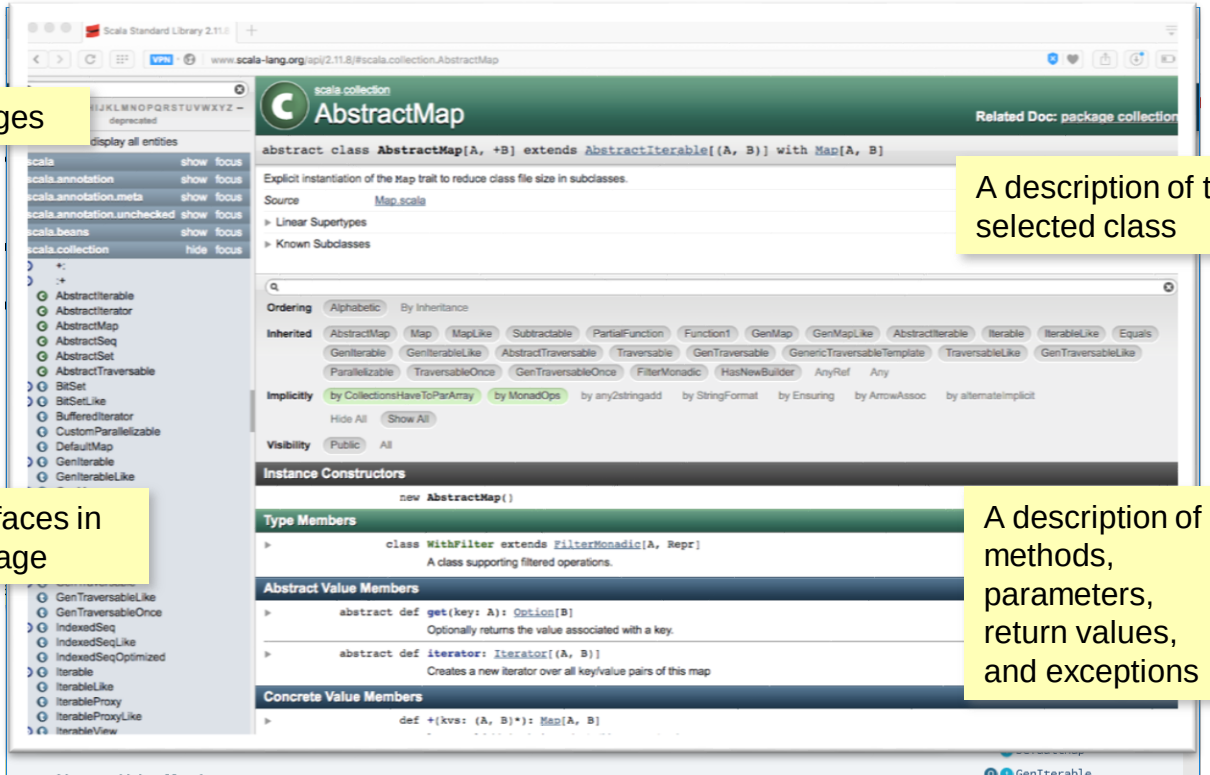
- Many people think that documentation…
  - …is of secondary importance
  - …prevents them from getting “real work” done
- They are very wrong!
  - You will need documentation!
    - you will be puzzled with your own code
  - Your colleagues will need documentation!
    - you need to be on the same page with your teammates
  - Your customers will need documentation!
    - they want to know how to use your system
- Write documentation as you code
  - writing a two-sentence description of a class/field/method before actually producing the code will help you organize your thoughts
  - makes you verbalize and thus *better understand* your ideas

Explicit Documentation

- Package-level documentation
  - What is the purpose of a package?
  - What subsystem does the package realize?
  - What are the main classes of the package?
- Class-level documentation
  - What is the purpose of a class?
  - Does the class apply a well-known pattern?
  - What other classes does the class collaborate with?
  - What comprises the state of the class?
  - What invariants hold of different class fields?
- Method-level documentation
  - What are method’s pre- and postconditions?
  - What are the possible errors and how are they handled?

Generating Documentation

- Documentation automators (Javadoc, Scaladoc, Doxygen, ...)
- The `javadoc` or `scaladoc` command-line tool
  - …processes the text between `/**` and `*/` and
  - …generates HTML documentation
  - tags `@param`, `@return`, and `@throws` have special meaning



Summary

Some challenges and three big ideas for improving your style:

- Chunking
  - organise code into packages, directories, files
- Readability
  - code is write-once read-many (by psychopaths)
  - use layout, names, comments, to help the reader
- Documentation
  - is needed for real software, and can be automated