EAPLI

Coding Best Practices

What's in a name?

What's in a name?

- Use specific names for variables, for example "value", "equals", "data", ... are not valid names for any case.
- Use meaningful names for variables. Variable name must define the exact explanation of its content.
- Don't start variables with o_, obj_, m_ etc. A variable does not need tags stating that it is a variable.

What's in a name? (ii)

- Decide and use one natural language for naming, e.g. using mixed English and German names will be inconsistent and unreadable.
- Use meaningful names for methods. The name must specify the exact action of the method and for most cases must start with a verb.
  - e.g. createPasswordHash

What's in a name? (iii)

- Classes should be named based on "nouns",
  - e.g., CheckingAccount
- Interfaces should be named based on "adjectives",
  - e.g., comparable
- Variables (e.g., object references) should be named to represent the exact meaning of its content,
  - e.g., initialBalance

What's in a name? (iv)

- Methods should be named based on "verbs" in terms of business operations, e.g.,
  - registerExpense
  - isFulfiled
  - hasOwner
  - canSpend
- Lines of code should read as a natural language sentence, e.g,
  - if (contract.isOverdue()){ … }
  - marysCheckingAccount.registerExpense(newShoes)

Code Smells & Refactorings

Code Smells & Refactorings

- Code Smell
  - Indicator of something potentially "bad" in the code
- Refactoring
  - Changes applied to the source code in order to improve its internal quality without affecting its external behaviour

Some Code Smells => Refactorings

- Comments
  - There's a fine line between comments that illuminate and comments that obscure. Are the comments necessary? Do they explain "why" and not "what"? remember, you're writing comments for people, not machines.
  - Refactorings: introduce method
- Long Method
  - All other things being equal, a shorter method is easier to read, easier to understand, and easier to troubleshoot.
  - Refactorings: introduce method.
- Long Parameter List
  - The more parameters a method has, the more complex it is. Limit the number of parameters you need in a given method, or use an object to combine the parameters.
  - Refactorings: introduce Parameter Object
- Data Class
  - Avoid classes that passively store data. Classes should contain data and methods to operate on that data, too.
  - Refactoring: move method
- Data Clumps
  - If you always see the same data hanging around together, maybe it belongs together. Consider rolling the related data up into a larger class.
  - Refactorings: introduce class

http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf

Smells to Refactorings Quick Reference Guide

industrial logic

Smell	Refactoring
Alternative Classes with Different Interfaces: occurs when the interfaces of two classes are different and yet the classes are quite similar. If you can find the similarities between the two classes, you can often refactor the classes to make them share a common interface [F 85, K 43]	Unify Interfaces with Adapter [K 247] Rename Method [F 273] Move Method [F 142]
Combinatorial Explosion: A subtle form of duplication, this smell exists when numerous pieces of code do the same thing using different combinations of data or behavior. [K 45]	Replace Implicit Language with Interpreter [K 269]
Comments (a.k.a. Deodorant): When you feel like writing a comment, first try "to refactor so that the comment becomes superfluous" [F 87]	Rename Method [F 273] Extract Method [F 110] Introduce Assertion [F 267]
Conditional Complexity: Conditional logic is innocent in its infancy, when it's simple to understand and contained within a few lines of code. Unfortunately, it rarely ages well. You implement several new features and suddenly your conditional logic becomes complicated and expansive. [K 41]	Introduce Null Object [F 260, K 301] Move Embellishment to Decorator [K 144] Replace Conditional Logic with Strategy [K 129] Replace State-Altering Conditionals with State [K 166]
Data Class: Classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes. [F 86]	Move Method [F 142] Encapsulate Field [F 206] Encapsulate Collection [F 208]
Data Clumps: Bunches of data that that hang around together really ought to be made into their own object. A good test is to consider deleting one of the data values: if you did this, would the others make any sense? If they don't, it's a sure sign that you have an object that's dying to be born. [F 81]	Extract Class [F 149] Preserve Whole Object [F 288] Introduce Parameter Object [F 295]
Divergent Change: Occurs when one class is commonly changed in different ways for different reasons. Separating these divergent responsibilities decreases the chance that one change could affect another and lower maintenance costs. [F 79]	Extract Class [F 149]
Duplicated Code: Duplicated code is the most pervasive and pungent smell in software. It tends to be either explicit or subtle. Explicit duplication exists in identical code, while subtle duplication exists in structures or processing steps that are outwardly different, yet	Chain Constructors [K 340] Extract Composite [K 214] Extract Method [F 110] Extract Class [F 149] Form Template Method [F 345, K 205] Introduce Null Object [F 260, K 301] Introduce Polymorphic Creation with Factory Method [K 88]

Other Best Practices

Other Best Practices

- Always construct objects in valid state.
- Improve code slightly every time you touch it, *i.e.*, refactor
- You need comprehensive tests
- Test code should be of the same quality as production code

Other Best Practices (ii)

To avoid	Best Done
`fooBar.fChar = barFoo.lchar = 'c';`	`fooBar.fChar = 'c';` `barFoo.lchar = 'c';`
`d = (a = b + c) + r;`	`a = b + c;` `d = a + r;`
`if (a == b && c == d)`	`if ((a == b) && (c == d))`

`if (booleanExpression) {`
`  return TRUE;`
`} else {`
`  return FALSE;`
`}`
	`return booleanExpression;`

`if (condition) {`
`  return x;`
`}`
`return y;`
	`return (condition ? x : y);`

Other Best Practices (iii)

- JavaDoc
  - http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html
- Code Quality Tools
  - FindBugs
    - http://findbugs.sourceforge.net/
  - PMD
    - http://pmd.sourceforge.net/
  - CheckStyle
    - http://checkstyle.sourceforge.net/
  - Sonar
    - http://www.sonarsource.org/

Exceptions

Exceptions

- When something exceptional occurs
  - Regular Flow
  - Exception Flow
- Improves source code's writing and reading
- It creates new names bound to business logic rules, *e.g.* InsufficientBalanceException

Exceptions

- Exceptions should answer these three questions:
  - What went wrong?
  - Why did it go wrong?
  - Where did it go wrong?

Java Exceptions

- Exception vs RuntimeException
  - Client code cannot do anything
    - -> Make it an unchecked exception (RunTimeException)
  - Client code will take some useful recovery action based on information in exception
    - -> Make it a checked exception (Exception)
- Dangers of blind "exception handling"
  - Not thinking about what to do with the exception by only using the IDE's automatic treatment for exceptions
  - Silencing the exception in a way that it will not be reported

Exceptions Best Practices

- Create exceptions for really exceptional situations based on meaningful business
  - Avoid technical exceptions
- New exception classes should have meaningfull methods
  - If the exception has no methods, it should not be a new class
- If you are not going to deal with it, do not catch it
- Leverage existing exceptions before creating new ones
- `try-catch-finally` and `try-with-resources` are your friends
- Document your exception classes

Java Coding Conventions

Java Coding Conventions

- Oracle – Code Conventions for Java Programming Language
  - https://horstmann.com/bigj2/style.html
  - 80% of the lifetime cost of a piece of software goes to maintenance.
  - Hardly any software is maintained for its whole life by the original author
  - Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

Java Coding Conventions: Class and Interface Declarations

1. Class/Interface comments
2. Class/Interface statement
3. Class/Interface comment (if needed)
4. Class static variables
5. Instance variables
6. Constructors
7. Methods
    1. Public
    2. Private

Java Coding Conventions: Comments

- Implementation comments
  - /*...*/
  - //
  - Use XXX to comment bogus code that works , FIXME to comment bogus and broken code and TODO for things that need to be done but are not bogus
- Documentation comments
  - /**...*/
  - see http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Java Coding Conventions: Declarations

- one declaration per line
- placement
  - at the beginning of blocks after "{"
  - Exception
    - FOR loops
- duplication
  - do not declare the same variable name in an inner block

```
//Not recommended
int level, size;
```

```
//Recommended
int level; // indentation  level
int size; // size of table Z
```

```
int count;
…
func() {
        int sum = 0;
        if (condition) {
                int count; //Avoid
                …
        }
        …
}
```

Java Coding Conventions: Declarations (ii)

- Initialization
  - Try to initialize variables where they are declared

```
class Sample extends Object {
        int ivar1;
        int ivar2;

        Sample(int i, int j) {
                ivar1 = i;
                ivar2 = j;
        }

        int emptyMethod()  {}
        ...
}
```

Java Coding Conventions: Declarations (iii)

- Class and Interface Declarations
  - No space between method name and "("
  - "{" at the end of the declaration line
  - "}" beginning of line with matching statement indentation
- Note: Typical today's IDE perform this automatically
  - Netbeans : Source > Format

Java Coding Conventions: Statements

- Simple Statements
  - Avoid multiple statements on the same line
    - argv++; argc--; // AVOID!
- Return statements
  - return;
  - return myDisk.size();
  - return (size ? size : defaultSize);

Java Coding Conventions: Statements (ii)

- If
  - Always use "{" and "}"

```
if (condition) {
        statements;
}
```

- if-else
  - Always use "{" and "}"

```
if (condition) {
        statements;
} else {
        statements;
}
```

- if-else-if-else
  - Always use "{" and "}"

```
if (condition) {
        statements;
} else if (condition){
        statements;
} else if (condition) {
        statements;
}
```

Java Coding Conventions: Statements (iv)

- switch statements

```
switch (condition) {
case ABC:
        statements;
        /* falls through */
case DEF:
        statements;
        break;

case XYZ:
        statements;
        break;
default:
        statements;
        break;
}
```

- try-catch statements

```
try {
        statements;
} catch (ExceptionClass e) {
        statements;
}
```

Java Coding Conventions: Blank Lines

- One blank line
  - Between methods
  - Between local variables in methods and first statement
  - Before a block or single-line comments
- Two blank lines between
  - Sections of a source file
  - Class and Interface definitions

Java Coding Conventions: Naming Conventions

- Class Names
  - Pascal Case
  - first letter of each word capitalized
  - Avoid acronyms and abbreviations
    - Exceptions URL, HTML

```
class Raster;
class ImageSprite;
```

- Interfaces
  - Pascal Case
  - first letter of each word capitalized
  - Avoid acronyms and abbreviations
    - Exceptions URL, HTML

```
interface RasterDelegate;
interface Serializable;
```

Java Coding Conventions: Naming Conventions (ii)

- Methods
  - Camel Case
  - first letter is lower case
  - first letter of subsequent word capitalized

```
run();
runFast();
getBackground();

int i;
float myWidth;

int MIN_WIDTH = 4;
int MAX_WIDTH = 999;
int GET_THE_CPU = 1;
```

- Variables
  - Camel Case
  - first letter is lower case
  - first letter of subsequent word capitalized
- Constants
  - Uppercase
  - Separated by "_"

Java Coding Conventions: common Prefixes and Suffixes

- Classes & Interfaces
  - Base/Abstract
    - AbstractVehicle
      - Bicycle
      - Car
    - VehicleBase
      - Bicycle
      - Car
  - Impl
    - Comparator
      - NumericComparatorImpl
- Methods
  - get
  - set
  - find

References

References

- http://www.oracle.com/technetwork/java/codeconv-138413.html
- http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html
- http://codebuild.blogspot.pt/2012/02/15-best-practices-of-variable-method.html
- http://www.omaxy.com/development/developer-guidelines/coding-standards/
- http://docs.oracle.com/javase/tutorial/essential/exceptions/
- http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html
- http://javarevisited.blogspot.pt/2013/03/0-exception-handling-best-practices-in-Java-Programming.html
- http://www.codinghorror.com/blog/2006/05/code-smells.html
- http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf