

# Java Coding Standards & Conventions

## 1 Introduction

- 1.1 Terminology notes
- 1.2 Guide notes
- 2 Source file basics
  - 2.1 File name
  - 2.2 File encoding: UTF-8
  - 2.3 Special characters
    - 2.3.1 Whitespace characters
    - 2.3.2 Special escape sequences
    - 2.3.3 Non-ASCII characters
- 3 Source file structure
  - 3.1 License or copyright information, if present
  - 3.2 Package statement
  - 3.3 Import statements
    - 3.3.1 No wildcard imports
    - 3.3.2 No line-wrapping
    - 3.3.3 Ordering and spacing
  - 3.4 Class declaration
    - 3.4.1 Exactly one top-level class declaration
    - 3.4.2 Class member ordering
      - 3.4.2.1 Overloads: never split
- 4 Formatting
  - 4.1 Braces
    - 4.1.1 Braces are used where optional
    - 4.1.2 Nonempty blocks: K & R style
    - 4.1.3 Empty blocks: may be concise
  - 4.2 Block indentation: +2 spaces
  - 4.3 One statement per line
  - 4.4 Column limit: 100
  - 4.5 Line-wrapping
    - 4.5.1 Where to break
    - 4.5.2 Indent continuation lines at least +4 spaces
  - 4.6 Whitespace
    - 4.6.1 Vertical Whitespace
    - 4.6.2 Horizontal whitespace
    - 4.6.3 Horizontal alignment: never required
  - 4.7 Grouping parentheses: recommended
  - 4.8 Specific constructs
    - 4.8.1 Enum classes
    - 4.8.2 Variable declarations
      - 4.8.2.1 One variable per declaration
      - 4.8.2.2 Declared when needed, initialized as soon as possible
    - 4.8.3 Arrays
      - 4.8.3.1 Array initializers: can be "block-like"
      - 4.8.3.2 No C-style array declarations
    - 4.8.4 Switch statements
      - 4.8.4.1 Indentation
      - 4.8.4.2 Fall-through: commented
      - 4.8.4.3 The default case is present
    - 4.8.5 Annotations
    - 4.8.6 Comments
      - 4.8.6.1 Block comment style
    - 4.8.7 Modifiers
    - 4.8.8 Numeric Literals
- 5 Naming
  - 5.1 Rules common to all identifiers
  - 5.2 Rules by identifier type
    - 5.2.1 Package names
    - 5.2.2 Class names
    - 5.2.3 Method names
    - 5.2.4 Constant names
    - 5.2.5 Non-constant field names
    - 5.2.6 Parameter names
    - 5.2.7 Local variable names
    - 5.2.8 Type variable names
  - 5.3 Camel case: defined
- 6 Programming Practices
  - 6.1 @Override: always used
  - 6.2 Exceptions

- 6.2.1 Caught exceptions: not ignored
  - 6.2.2 Don't Catch Generic Exceptions
  - 6.3 Static members: qualified using class
- 6.4 Finalizers: not used
- 6.5 Write Short Methods
- Limit Variable Scope
- 7 Javadoc
  - 7.1 Formatting
    - 7.1.1 General form
    - 7.1.2 Paragraphs
    - 7.1.3 At-clauses
  - 7.2 The summary fragment
  - 7.3 Where Javadoc is used
    - 7.3.1 Exception: self-explanatory methods
    - 7.3.2 Exception: overrides



Java coding standards follow the Google Java Style Guide (<https://google-styleguide.googlecode.com/svn/trunk/javaguide.html#s1-introduction>). This page is a copy of the style guide taken 12/1/14 with a number of modifications that are specific to the CrowdTwist team.

# 1 Introduction

This document serves as the **complete** definition of CrowdTwist's coding standards for source code in the Java™ Programming Language. Like other programming style guides, the issues covered span not only aesthetic issues of formatting, but other types of conventions or coding standards as well. However, this document focuses primarily on the **hard-and-fast rules** that we follow universally, and avoids giving *advice* that isn't clearly enforceable (whether by human or tool).

## 1.1 Terminology notes ↔

In this document, unless otherwise clarified:

1. The term *class* is used inclusively to mean an "ordinary" class, enum class, interface or annotation type (`@interface`).
2. The term *comment* always refers to *implementation* comments. We do not use the phrase "documentation comments", instead using the common term "Javadoc."

Other "terminology notes" will appear occasionally throughout the document.

## 1.2 Guide notes ↔

Example code in this document is **non-normative**. That is they may not illustrate the *only* stylish way to represent the code. Optional formatting choices made in examples should not be enforced as rules.

# 2 Source file basics ↔

## 2.1 File name ↔

The source file name consists of the case-sensitive name of the top-level class it contains, plus the `.java` extension.

## 2.2 File encoding: UTF-8 ↔

Source files are encoded in **UTF-8**.

## 2.3 Special characters ↔

### 2.3.1 Whitespace characters ↔

Aside from the line terminator sequence, the **ASCII horizontal space character (0x20)** is the only whitespace character that appears anywhere in a source file. This implies that:

1. All other whitespace characters in string and character literals are escaped.
2. Tab characters are **not** used for indentation.

### 2.3.2 Special escape sequences ↗

For any character that has a special escape sequence (`\b`, `\t`, `\n`, `\f`, `\r`, `\"`, `\'` and `\\`), that sequence is used rather than the corresponding octal (e.g. `\012`) or Unicode (e.g. `\u000a`) escape.

### 2.3.3 Non-ASCII characters ↗

For the remaining non-ASCII characters, either the actual Unicode character (e.g. `μ`) or the equivalent Unicode escape (e.g. `\u221e`) is used, depending only on which makes the code **easier to read and understand**.

**Tip:** In the Unicode escape case, and occasionally even when actual Unicode characters are used, an explanatory comment can be very helpful.

Examples:

Example	Discussion
<code>String unitAbbrev = "s";</code>	Best: perfectly clear even without a comment.
<code>String unitAbbrev = "\u03bcs"; // "s"</code>	Allowed, but there's no reason to do this.
<code>String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"</code>	Allowed, but awkward and prone to mistakes.
<code>String unitAbbrev = "\u03bcs";</code>	Poor: the reader has no idea what this is.
<code>return '\uffff' + content; // byte order mark</code>	Good: use escapes for non-printable characters, and comment if necessary.

**Tip:** Never make your code less readable simply out of fear that some programs might not handle non-ASCII characters properly. If that should happen, those programs are **broken** and they must be **fixed**.

## 3 Source file structure ↗

A source file consists of, **in order**:

1. License or copyright information, if present
2. Package statement
3. Import statements
4. Exactly one top-level class

**Exactly one blank line** separates each section that is present.

### 3.1 License or copyright information, if present ↗

If license or copyright information belongs in a file, it belongs here.

### 3.2 Package statement ↗

The package statement is **not line-wrapped**. The column limit (Section 4.4, [Column limit: 80 or 100](#)) does not apply to package statements.

### 3.3 Import statements ↗

#### 3.3.1 No wildcard imports ↗

**Wildcard imports**, static or otherwise, **are not used**.

#### 3.3.2 No line-wrapping ↗

Import statements are **not line-wrapped**. The column limit (Section 4.4, [Column limit: 80 or 100](#)) does not apply to import statements.

### 3.3.3 Ordering and spacing ↗

Import statements are divided into the following groups, in this order, with each group separated by a single blank line:

1. All static imports in a single group
2. `com.google` imports (only if this source file is in the `com.google` package space)
3. Third-party imports, one group per top-level package, in ASCII sort order
  - for example: `android`, `com`, `junit`, `org`, `sun`
4. `java` imports
5. `javax` imports

Within a group there are no blank lines, and the imported names appear in ASCII sort order. (**Note:** this is not the same as the import *statements* being in ASCII sort order; the presence of semicolons warps the result.)

## 3.4 Class declaration ↗

### 3.4.1 Exactly one top-level class declaration ↗

Each top-level class resides in a source file of its own.

### 3.4.2 Class member ordering ↗

The ordering of the members of a class can have a great effect on learnability, but there is no single correct recipe for how to do it. Different classes may order their members differently.

What is important is that each class order its members in **some logical order**, which its maintainer could explain if asked. For example, new methods are not just habitually added to the end of the class, as that would yield "chronological by date added" ordering, which is not a logical ordering.

Group members of same scope together in increasing order, specifically: *static*, *private*, *protected*, *public*.

#### 3.4.2.1 Overloads: never split ↗

When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no intervening members.

## 4 Formatting ↗

**Terminology Note:** *block-like construct* refers to the body of a class, method or constructor. Note that, by Section 4.8.3.1 on [array initializers](#), any array initializer *may* optionally be treated as if it were a block-like construct.

### 4.1 Braces ↗

#### 4.1.1 Braces are used where optional ↗

Braces are used with `if`, `else`, `for`, `do` and `while` statements, even when the body is empty or contains only a single statement.

#### 4.1.2 Nonempty blocks: K & R style ↗

Braces follow the Kernighan and Ritchie style ("[Egyptian brackets](#)") for *nonempty* blocks and block-like constructs:

- No line break before the opening brace.
- Line break after the opening brace.
- Line break before the closing brace.
- Line break after the closing brace *if* that brace terminates a statement or the body of a method, constructor or *named* class. For example, there is *no* line break after the brace if it is followed by `else` or a comma.

Example:

```
return new MyClass() { @Override public void method() { if (condition()) { try {
something(); } catch (ProblemException e) { recover(); } } } };
```

A few exceptions for enum classes are given in Section 4.8.1, [Enum classes](#).

### 4.1.3 Empty blocks: may be concise ↪

An empty block or block-like construct *may* be closed immediately after it is opened, with no characters or line break in between (`{}`), **unless** it is part of a *multi-block statement* (one that directly contains multiple blocks: `if/else-if/else` or `try/catch/finally`).

Example:

```
void doNothing() {}
```

## 4.2 Block indentation: +2 spaces ↪

Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block. (See the example in Section 4.1.2, [Nonempty blocks: K & R Style](#).)

## 4.3 One statement per line ↪

Each statement is followed by a line-break.

## 4.4 Column limit: 100 ↪

There is a column limit of 100 characters. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in Section 4.5, [Line-wrapping](#).

Exceptions:

1. Lines where obeying the column limit is not possible (for example, a long URL in Javadoc, or a long JSNI method reference).
2. `package` and `import` statements (see Sections 3.2 [Package statement](#) and 3.3 [Import statements](#)).
3. Command lines in a comment that may be cut-and-pasted into a shell.

## 4.5 Line-wrapping ↪

**Terminology Note:** When code that might otherwise legally occupy a single line is divided into multiple lines, typically to avoid overflowing the column limit, this activity is called *line-wrapping*.

There is no comprehensive, deterministic formula showing *exactly* how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

**Tip:** Extracting a method or local variable may solve the problem without the need to line-wrap.

### 4.5.1 Where to break ↪

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**. Also:

1. When a line is broken at a *non-assignment* operator the break comes *before* the symbol.
  - This also applies to the following "operator-like" symbols: the dot separator (`.`), the ampersand in type bounds (`<T extends Foo & Bar>`), and the pipe in catch blocks (`catch (FooException | BarException e)`).
2. When a line is broken at an *assignment* operator the break typically comes *after* the symbol, but either way is acceptable.
  - This also applies to the "assignment-operator-like" colon in an enhanced `for` ("foreach") statement.

3. A method or constructor name stays attached to the open parenthesis ( ( ) that follows it.
4. A comma ( , ) stays attached to the token that precedes it.

## 4.5.2 Indent continuation lines at least +4 spaces ↪

When line-wrapping, each line after the first (each *continuation line*) is indented at least +4 from the original line.

When there are multiple continuation lines, indentation may be varied beyond +4 as desired. In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.

Section 4.6.3 on [Horizontal alignment](#) addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

## 4.6 Whitespace ↪

### 4.6.1 Vertical Whitespace ↪

A single blank line appears:

1. *Between* consecutive members (or initializers) of a class: fields, constructors, methods, nested classes, static initializers, instance initializers.
  - **Exception:** A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create *logical groupings* of fields.
2. Within method bodies, as needed to create *logical groupings* of statements.
3. *Optionally* before the first member or after the last member of the class (neither encouraged nor discouraged).
4. As required by other sections of this document (such as Section 3.3, [Import statements](#)).

Multiple consecutive blank lines are permitted, but never required (or encouraged).

### 4.6.2 Horizontal whitespace ↪

Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places **only**.

1. Separating any reserved word, such as `if`, `for` or `catch`, from an open parenthesis ( ( ) that follows it on that line
2. Separating any reserved word, such as `else` or `catch`, from a closing curly brace ( ) that precedes it on that line
3. Before any open curly brace ( { ), with two exceptions:
  - `@SomeAnnotation({a, b})` (no space is used)
  - `String[][] x = {{ "foo" }};` (no space is required between { { , by item 8 below)
4. On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:
  - the ampersand in a conjunctive type bound: `<T extends Foo & Bar>`
  - the pipe for a catch block that handles multiple exceptions: `catch (FooException | BarException e)`
  - the colon ( : ) in an enhanced `for` ("foreach") statement
5. After `, :` or the closing parenthesis ( ) of a cast
6. On both sides of the double slash ( // ) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.
7. Between the type and variable of a declaration: `List<String> list`

**Note:** This rule never requires or forbids additional space at the start or end of a line, only *interior* space.

### 4.6.3 Horizontal alignment: never required ↪

**Terminology Note:** *Horizontal alignment* is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but is **never required**. It is not even required to *maintain* horizontal alignment in places where it was already used.

Here is an example without alignment, then using alignment:

```
private int x; // this is fine private Color color; // this too private int x; //
permitted, but future edits private Color color; // may leave it unaligned
```

**Tip:** Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing formatting mangled, and that is **allowed**. More often it prompts the coder (perhaps you) to adjust

whitespace on nearby lines as well, possibly triggering a cascading series of reformatings. That one-line change now has a "blast radius." This can at worst result in pointless busywork, but at best it still corrupts version history information, slows down reviewers and exacerbates merge conflicts.



The following form of alignment, specifically around class members, is not allowed:

```
public class MyClass {
    public static String AVeryLongStringThatIsNotImportant = "Hello World";
    private int someInt          = 1;
    protected double[] vals      = {1.0f, 2.0f};
}
```

## 4.7 Grouping parentheses: recommended ↺

Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is *not* reasonable to assume that every reader has the entire Java operator precedence table memorized.

## 4.8 Specific constructs ↺

### 4.8.1 Enum classes ↺

After each comma that follows an enum constant, a line-break is optional.

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer (see Section 4.8.3.1 on [array initializers](#)).

```
private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

Since enum classes *are* classes, all other rules for formatting classes apply.

### 4.8.2 Variable declarations ↺

#### 4.8.2.1 One variable per declaration ↺

Every variable declaration (field or local) declares only one variable: declarations such as `int a, b;` are not used.

#### 4.8.2.2 Declared when needed, initialized as soon as possible ↺

Local variables are **not** habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

### 4.8.3 Arrays ↺

#### 4.8.3.1 Array initializers: can be "block-like" ↺

Any array initializer may *optionally* be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list):

```
new int[] { new int[] { 0, 1, 2, 3 0, } 1, 2, new int[] { 3, 0, 1, } 2, 3 } new int[]
{0, 1, 2, 3}
```

#### 4.8.3.2 No C-style array declarations ↪

The square brackets form a part of the *type*, not the variable: `String[] args`, not `String args[]`.

#### 4.8.4 Switch statements ↪

**Terminology Note:** Inside the braces of a *switch block* are one or more *statement groups*. Each statement group consists of one or more *switch labels* (either `case FOO:` or `default:`), followed by one or more statements.

##### 4.8.4.1 Indentation ↪

As with any other block, the contents of a switch block are indented +2.

After a switch label, a newline appears, and the indentation level is increased +2, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

##### 4.8.4.2 Fall-through: commented ↪

Within a switch block, each statement group either terminates abruptly (with a `break`, `continue`, `return` or thrown exception), or is marked with a comment to indicate that execution will or *might* continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically `// fall through`). This special comment is not required in the last statement group of the switch block. Example:

```
switch (input) { case 1: case 2: prepareOneOrTwo(); // fall through case 3:
handleOneTwoOrThree(); break; default: handleLargeNumber(input); }
```

##### 4.8.4.3 The default case is present ↪

Each switch statement includes a `default` statement group, even if it contains no code.

#### 4.8.5 Annotations ↪

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (Section 4.5, [Line-wrapping](#)), so the indentation level is not increased. Example:

```
@Override @Nullable public String getNameIfPresent() { ... }
```

**Exception:** A *single* parameterless annotation *may* instead appear together with the first line of the signature, for example:

```
@Override public int hashCode() { ... }
```

Annotations applying to a field also appear immediately after the documentation block, but in this case, *multiple* annotations (possibly parameterized) may be listed on the same line; for example:



```
@Partial @Mock DataLoader loader;
```

There are no specific rules for formatting parameter and local variable annotations.

## 4.8.6 Comments ↗

### 4.8.6.1 Block comment style ↗

Block comments are indented at the same level as the surrounding code. They may be in `/* ... */` style or `// ...` style. For multi-line `/* ... */` comments, subsequent lines must start with `*` aligned with the `*` on the previous line.

```
/* * This is // And so /* Or you can * okay. // is this. * even do this. */ */
```

Comments are not enclosed in boxes drawn with asterisks or other characters.

**Tip:** When writing multi-line comments, use the `/* ... */` style if you want automatic code formatters to re-wrap the lines when necessary (paragraph-style). Most formatters don't re-wrap lines in `// ...` style comment blocks.



For multi-line comments, do not include documentation on the first and last line.

**Bad:**

```
/* This variable
 * is a good
variable */
```

**Good:**

```
/*
 * This variable is a
 * good variable
 */
```

## 4.8.7 Modifiers ↗

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

```
public protected private abstract static final transient volatile synchronized native
strictfp
```

## 4.8.8 Numeric Literals ↗

long-valued integer literals use an uppercase `L` suffix, never lowercase (to avoid confusion with the digit 1). For example, `3000000000L` rather than `3000000000l`.

## 5 Naming ↪

### 5.1 Rules common to all identifiers ↪

Identifiers use only ASCII letters and digits, and in two cases noted below, underscores. Thus each valid identifier name is matched by the regular expression `\w+`.

Special prefixes or suffixes, like those seen in the examples `name_`, `mName`, `s_name` and `kName`, are **not** used.

### 5.2 Rules by identifier type ↪

#### 5.2.1 Package names ↪

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, `com.example.deepspace`, not `com.example.deepSpace` or `com.example.deep_space`.

Use the plural for packages with homogeneous content and the singular for packages with heterogeneous content. For example, you're creating an event processing infrastructure which contains different types of event classes (i.e. `ReadEmailEvent`, `LoggedInEvent`, etc) as well as classes for processing events (i.e. `EventFactory`, `EventProcessor`, etc). The event type classes are homogenous instances, where each instance is a type of event, and therefore should use the plural form for the package name (i.e. `com.crowdtwist.events`). The event processing classes are heterogeneous instances and therefore should use the singular form for the package name (i.e. `com.crowdtwist.event`).

#### 5.2.2 Class names ↪

Class names are written in `UpperCamelCase`.

Class names are typically nouns or noun phrases. For example, `Character` or `ImmutableList`. Interface names may also be nouns or noun phrases (for example, `List`), but may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

There are no specific rules or even well-established conventions for naming annotation types.

*Test* classes are named starting with the name of the class they are testing, and ending with `Test`. For example, `HashTest` or `HashIntegrationTest`.

#### 5.2.3 Method names ↪

Method names are written in `lowerCamelCase`.

Method names are typically verbs or verb phrases. For example, `sendMessage` or `stop`.

Underscores may appear in JUnit *test* method names to separate logical components of the name. One typical pattern is `test<MethodUnderTest>_<state>`, for example `testPop_emptyStack`. There is no One Correct Way to name test methods.

#### 5.2.4 Constant names ↪

Constant names use `CONSTANT_CASE`: all uppercase letters, with words separated by underscores. But what *is* a constant, exactly?

Every constant is a static final field, but not all static final fields are constants. Before choosing constant case, consider whether the field really *feels like* a constant. For example, if any of that instance's observable state can change, it is almost certainly not a constant. Merely *intending* to never mutate the object is generally not enough. Examples:

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES =
    ImmutableList.of("Ed", "Ann");
static final Joiner COMMA_JOINER = Joiner.on(',');
// because Joiner is immutable
static final SomeMutableType[] EMPTY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }
// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection =
    new HashSet<String>();
static final ImmutableSet<SomeMutableType> mutableElements =
    ImmutableSet.of(mutable);
static final Logger logger =
    Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these",
    "can", "change"};
```

These names are typically nouns or noun phrases.

## 5.2.5 Non-constant field names ↗

Non-constant field names (static or otherwise) are written in [lowerCamelCase](#).

These names are typically nouns or noun phrases. For example, `computedValues` or `index`.

## 5.2.6 Parameter names ↗

Parameter names are written in [lowerCamelCase](#).

One-character parameter names should be avoided.

## 5.2.7 Local variable names ↗

Local variable names are written in [lowerCamelCase](#), and can be abbreviated more liberally than other types of names.

However, one-character names should be avoided, except for temporary and looping variables.

Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

## 5.2.8 Type variable names ↗

Each type variable is named in one of two styles:

- A single capital letter, optionally followed by a single numeral (such as `E`, `T`, `X`, `T2`)
- A name in the form used for classes (see Section 5.2.2, [Class names](#)), followed by the capital letter `T` (examples: `RequestT`, `FooBarT`).

## 5.3 Camel case: defined ↗

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve predictability, the following (nearly) deterministic scheme is followed.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
  - *Recommended:* if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel case *per se*; it defies any convention, so this recommendation does not apply.
3. Now lowercase *everything* (including acronyms), then uppercase only the first character of:
  - ... each word, to yield *upper camel case*, or
  - ... each word except the first, to yield *lower camel case*
4. Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded. Examples:

Prose form	Correct	Incorrect
------------	---------	-----------



You should not do this. In almost all cases it is inappropriate to catch generic `Exception` or `Throwable`, preferably not `Throwable`, because it includes `Error` exceptions as well. It is very dangerous. It means that Exceptions you never expected (including `RuntimeException`s like `ClassCastException`) end up getting caught in application-level error handling. It obscures the failure handling properties of your code. It means if someone adds a new type of `Exception` in the code you're calling, the compiler won't help you realize you need to handle that error differently. And in most cases you shouldn't be handling different types of exception the same way, anyway.

There are rare exceptions to this rule: certain test code and top-level code where you want to catch all kinds of errors (to prevent them from showing up in a UI, or to keep a batch job running). In that case you may catch generic `Exception` (or `Throwable`) and handle the error appropriately. You should think very carefully before doing this, though, and put in comments explaining why it is safe in this place.

Alternatives to catching generic `Exception`:

- Catch each exception separately as separate catch blocks after a single try. This can be awkward but is still preferable to catching all Exceptions. Beware repeating too much code in the catch blocks.
- Refactor your code to have more fine-grained error handling, with multiple try blocks. Split up the IO from the parsing, handle errors separately in each case.
- Rethrow the exception. Many times you don't need to catch the exception at this level anyway, just let the method throw it.

Remember: exceptions are your friend! When the compiler complains you're not catching an exception, don't scowl. Smile: the compiler just made it easier for you to catch runtime problems in your code.

## 6.3 Static members: qualified using class

When a reference to a static class member must be qualified, it is qualified with that class's name, not with a reference or expression of that class's type.

```
Foo aFoo = ...; Foo.aStaticMethod(); // good aFoo.aStaticMethod(); // bad somethingThatYieldsAFoo().aStaticMethod(); // very bad
```

## 6.4 Finalizers: not used

It is **extremely rare** to override `Object.finalize`.

**Tip:** Don't do it. If you absolutely must, first read and understand *Effective Java* Item 7, "Avoid Finalizers," very carefully, and *then* don't do it.

## 6.5 Write Short Methods

To the extent that it is feasible, methods should be kept small and focused. It is, however, recognized that long methods are sometimes appropriate, so no hard limit is placed on method length. If a method exceeds 100 lines or so, think about whether it can be broken up without harming the structure of the program.

## Limit Variable Scope

The scope of local variables should be kept to a minimum. By doing so, you increase the readability and maintainability of your code and reduce the likelihood of error. Each variable should be declared in the innermost block that encloses all uses of the variable.

Local variables should be declared at the point they are first used. Nearly every local variable declaration should contain an initializer. If you don't yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do.

One exception to this rule concerns try-catch statements. If a variable is initialized with the return value of a method that throws a checked exception, it must be initialized inside a try block. If the value must be used outside of the try block, then it must be declared before the try block, where it cannot yet be sensibly initialized:

```
// Instantiate class cl, which represents some sort of Set
Set s = null;
try {
    s = (Set) cl.newInstance();
} catch(IllegalAccessException e) {
    throw new IllegalArgumentException(cl + " not accessible");
} catch(InstantiationException e) {
    throw new IllegalArgumentException(cl + " not instantiable");
}

// Exercise the set
s.addAll(Arrays.asList(args));
```

But even this case can be avoided by encapsulating the try-catch block in a method:

```
Set createSet(Class cl) {
    // Instantiate class cl, which represents some sort of Set
    try {
        return (Set) cl.newInstance();
    } catch(IllegalAccessException e) {
        throw new IllegalArgumentException(cl + " not accessible");
    } catch(InstantiationException e) {
        throw new IllegalArgumentException(cl + " not instantiable");
    }
}

...

// Exercise the set
Set s = createSet(cl);
s.addAll(Arrays.asList(args));
```

Loop variables should be declared in the for statement itself unless there is a compelling reason to do otherwise:

```
for (int i = 0; i < n; i++) {
    doSomething(i);
}
```

and

```
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomethingElse(i.next());
}
```

## 7 Javadoc

### 7.1 Formatting

#### 7.1.1 General form

The *basic* formatting of Javadoc blocks is as seen in this example:

```
/** * Multiple lines of Javadoc text are written here, * wrapped normally... */ public
int method(String p1) { ... }
```

... or in this single-line example:

```
/** An especially short bit of Javadoc. */
```

The basic form is always acceptable. The single-line form may be substituted when there are no at-clauses present, and the entirety of the Javadoc block (including comment markers) can fit on a single line.

### 7.1.2 Paragraphs ↗

One blank line—that is, a line containing only the aligned leading asterisk (\*)—appears between paragraphs, and before the group of "at-clauses" if present. Each paragraph but the first has <p> immediately before the first word, with no space after.

### 7.1.3 At-clauses ↗

Any of the standard "at-clauses" that are used appear in the order @param, @return, @throws, @deprecated, and these four types never appear with an empty description. When an at-clause doesn't fit on a single line, continuation lines are indented four (or more) spaces from the position of the @.

## 7.2 The summary fragment ↗

The Javadoc for each class and member begins with a brief **summary fragment**. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.

This is a fragment—a noun phrase or verb phrase, not a complete sentence. It does **not** begin with A {`@code Foo`} is a..., or This method returns..., nor does it form a complete imperative sentence like Save the record.. However, the fragment is capitalized and punctuated as if it were a complete sentence.

**Tip:** A common mistake is to write simple Javadoc in the form `/** @return the customer ID */`. This is incorrect, and should be changed to `/** Returns the customer ID. */`.

## 7.3 Where Javadoc is used ↗

At the *minimum*, Javadoc is present for every `public` class, and every `public` or `protected` member of such a class, with a few exceptions noted below.

Other classes and members still have Javadoc *as needed*. Whenever an implementation comment would be used to define the overall purpose or behavior of a class, method or field, that comment is written as Javadoc instead. (It's more uniform, and more tool-friendly.)

### 7.3.1 Exception: self-explanatory methods ↗

Javadoc is optional for "simple, obvious" methods like `getFoo`, in cases where there *really and truly* is nothing else worthwhile to say but "Returns the foo".

**Important:** it is not appropriate to cite this exception to justify omitting relevant information that a typical reader might need to know. For example, for a method named `getCanonicalName`, don't omit its documentation (with the rationale that it would say only `/** Returns the canonical name. */`) if a typical reader may have no idea what the term "canonical name" means!

### 7.3.2 Exception: overrides ↗

Javadoc is not always present on a method that overrides a supertype method.