Type-Safe PHP

A compile time approach

Java Code Conventions

Rev	Date	Author	Description	State
0.1	21.10.2012	rstoll	Initial version	done
0.2	31.10.2012	rstoll	Licence info added and revised	done
0.3	17.11.2012	rstoll	Revised citation uses now citavi placeholders	done
0.4	18.11.2012	rstoll	Removed output parameter – not supported anyway	done
0.5	04.02.2013	rstoll	Revised return	done
0.6	18.02.2013	rstoll	public class variable begin now with lower case	done
0.7	22.03.2013	rstoll	Included a table of contents	done
0.8	27.09.2013	rstoll	Added info about license header and author tag	done
0.9	29.09.2013	rstoll	Included Sun's code convention	done
1.0	21.03.2014	rstoll	Included licence notice	done

© Copyright 2014 Robert Stoll <rstoll@tutteli.ch>

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

Adapted with permission from JAVA CODE CONVENTIONS. Copyright 1995-1999 Sun Microsysytems, Inc. All rights reserved.

Contents

1 Introduction ... 5

1.1 Why Have Code Conventions ... 5

1.2 Acknowledgments ... 5

2 File Names ... 6

2.1 File Suffixes ... 6

2.2 Common File Names ... 6

3 File Organisation ... 6

3.1 Java Source Files ... 7

3.1.1 Beginning Comments ... 7

3.1.2 Package and Import Statements ... 8

3.1.3 Class and Interface Declarations ... 8

4 Indentation ... 9

4.1 Line Length ... 9

4.2 Wrapping Lines ... 9

5 Comments ... 12

5.1 Implementation Comment Formats ... 13

5.2 Documentation Comments ... 13

6 Declarations ... 14

6.1 Number per Line ... 14

6.2 Placement ... 14

6.3 Initialization ... 14

6.4 Class and Interface Declarations ... 15

7 Statements ... 15

7.1 Simple Statements ... 15

7.2 Compound Statements ... 16

7.3 return Statements ... 16

7.4 if, if-else, if-else-if-else Statements ... 16

7.5 for Statements ... 17

7.6 while Statements ... 17

7.7 do-while Statements ... 17

7.8 switch Statements ... 18

7.9 try-catch Statements 18

8 White Space 19

8.1 Blank Lines 19

8.2 Blank Spaces 19

9 Naming Conventions 20

10 Programming Practices 22

10.1 Providing Access to Instance and Class Variables 22

10.2 Referring to Class Variables and Methods 22

10.3 Constants 22

10.4 Variable Assignments 22

10.5 Miscellaneous Practices 23

10.5.1 Parentheses 23

10.5.2 Returning Values 23

10.5.3 Expressions before '?' in the Conditional Operator 24

10.5.4 Special Comments 25

10.5.5 Overloaded constructors 25

10.5.6 Method length 25

10.5.7 Number of parameters 25

10.5.8 File length 25

11 Code Examples 26

11.1 Java Source File Example 26

12 List of References 26

1 Introduction

This document reflects the coding standards for Java used in the project 'Type-Safe PHP: A compile time approach'. This convention shall be followed for every code written in Java.

The conventions are based on Sun Microsystems' *Java Code Conventions* (1997). Additional conventions are heavily based on input from *Clean Code: A handbook of agile software craftsmanship* (Martin and Coplien 2009).

✎ [1] marks sections with changes compared to Sun's convention and sections marked with ⊕ [1] are additional conventions.

Sun's chapter 'Introduction' is covered in the following two sub-chapters.

1.1 Why Have Code Conventions

Code conventions are important to programmers for a number of reasons:

- 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.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

1.2 Sun Microsystems' Acknowledgments

Major contributions are from Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, and Scott Hommel.

[1] Mark 2012

2 File Names

This section lists commonly used file suffixes and names.

2.1 File Suffixes

The following file suffixes are used:

File Type	Suffix
Java source	.java
Java bytecode	.class

2.2 Common File Names

Frequently used file names include:

File Name	Use
LICENSE	Contains the Apache 2.0 license information
README.md	Short explanation about the purpose of the corresponding component

3 File Organisation

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

"It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than one thing, and should be converted into many smaller functions, each of which does one thing" (Martin and Coplien 2009: 302). We adopt these thoughts to classes and consider if we injure the 'Single Responsibility Principle' (Martin and Coplien 2009: 138) each time we see sections which are separated using lines. For instance

```
class Costumer
{
    ...
    // Customer Address methods ----------
    ...
    // Customer methods ------------------
    ...
}
```

Files longer than 1500 lines are cumbersome and should be avoided.

3.1 Java Source Files

Each Java source file contains a single `public class` or `interface`. When `private class`es and `interface`s are associated with a `public class`, you can put them in the same source file as the `public class`. The `public class` should be the first class or interface in the file.

Enums are treated like classes. Therefore a `public enum` has to be put in a single source file.

Java source files have the following ordering:

- Beginning comments (see chapter 3.1.1 'Beginning Comments')
- Package and Import statements; for example:

```
import java.applet.Applet;
import java.awt.*;
import java.net.*;
```

- Class and interface declarations (see chapter 0 'Class and Interface Declarations')

3.1.1 Beginning Comments

Source files should not include the version info. This is tracked by the version control system - ~~CVS, SVN~~, Git (we use Git)

Each file contains the following license notice at the top which refers to the LICENSE-file, which shall be found in the root of the component. This notice suffice and there is no need to include the whole license as stated in the LICENSE-file unless the source is copied from another source, then the full acknowledgment has to be stated (no copyright infringement)

```
/*
 * This file is part of the TSPHP project published under the Apache License 2.0
 * For the full copyright and license information, please have a look at LICENSE in the
 * root folder or visit the project's website http://tsphp.ch/wiki/display/TSPHP/License
 */
```

Use the following style for XML, HTML –files etc.

```
<!--
  This file is part of the TSPHP project published under the Apache License 2.0
  For the full copyright and license information, please have a look at LICENSE in the
  root folder or visit the project's website http://tsphp.ch/wiki/display/TSPHP/License
  -->
```

Description of the class or interface belongs to the corresponding JavaDoc and not into the source file comment (header of the file)

3.1.2 *Package and Import Statements*

The first line of most Java source files is a `package` statement. After that, `import` statements can follow. For example:

```
package ch.tutteli.tsphp.typechecker;
import ch.tutteli.tsphp.common.ISymbol;
```

3.1.3 *Class and Interface Declarations*

The following table describes the parts of a class or interface declaration, in the order that they should appear.

	Part of Class/Interface Declaration	**Notes**
1	Class/interface documentation comment (/**...*/)	Chapter 5.2 'Documentation Comments' for information on what should be in this comment.
2	class or interface statement	
3	Class/interface implementation comment (/*...*/), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment. ✎ Don't clutter the classes with examples though
4	Class (`static`) variables	First the `public` class variables, then the `protected`, and then the `private`. ✎ Class (`static`) variables should be either `private` or `protected`. Access modifier `public` is only allowed for data-structure-like classes - for instance DTOs (Fowler 2012).
5	Instance variables	First `public`, then the `protected`, and then the `private`. ✎ Instance variables should be either `private` or `protected`. Access modifier `public` is only allowed for data-structure-like classes - for instance DTOs (Fowler 2012).
6	Constructors	
7	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.
8	Inner classes/interfaces	

4 Indentation

Four spaces should be used as the unit of indentation.

✎ Use only spaces and not tabs

4.1 Line Length

✎ Avoid lines longer than 120 characters.

4.2 Wrapping Lines

When an expression does not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

```
function(longExpression1, longExpression2, longExpression3,
         longExpression4, longExpression5);

var = function1(longExpression1,
                function2(longExpression2,
                          longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6; // PREFER
longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longname6; // AVOID
```

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}
//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}
```

Line wrapping for `if` statements are a clear smell of 'Encapsulate Conditionals' (Martin and Coplien 2009: 301) which give the advice to replace the conditional by a method. Consider the following example:

```
public String getCode(String message) {
    Date now = new Date();
    if (forceAvailability ||
          (
            (availableFrom == null || availableFrom >= now)
            && (availableUntil == null || availableUntil <= now)
          )) {
      return code;
    }
    throw new NotAvailableException("code is not available anymore");
}
```

…continued on the next page

This conditional should be encapsulated and could be refactored as follows:

```
public String getCode(String message) {
    Date now = new Date();
    if (isAvailable()) {
      return code;
    }
    throw new NotAvailableException("code is not available anymore");
}

private boolean isAvailable() {
    return forceAvailability ||
          (
            (availableFrom == null || availableFrom >= now)
            && (availableUntil == null || availableUntil <= now)
          );
}
```

However, if you really need to have line wrapping in if statements (due to reasonable performance arguments) then follow the following rules.

`if` statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();            //MAKE THIS LINE EASY TO MISS
}
//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;
alpha = (aLongBooleanExpression) ? beta
                                 : gamma;
alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

5 Comments

First of all,

Comments should be used as additional explanation when the code cannot say it for itself.

and now read on

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /*...*/, and //. Documentation comments (known as "doc comments") are Java-only, and are delimited by /**...*/. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are mean for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective, to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or nonobvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters. Comments should never include special characters such as form-feed and backspace.

- Keep in mind that methods should only do one thing and classes should have only one responsibility (Martin and Coplien 2009). Comments which purpose is separation are smells and injure those principles.

- Remove obsolete comments. "A comment that has gotten old, irrelevant, and incorrect is obsolete. Comments get old quickly. It is best not to write a comment that will become obsolete. If you find an obsolete comment, it is best to update it or get rid of it as quickly as possible. Obsolete comments tend to migrate away from the code they once described. They become floating islands of irrelevance and misdirection in the code" (Martin and Coplien 2009: 286).
- Remove redundant comments. "A comment is redundant if it describes something that adequately describes itself" (Martin and Coplien 2009: 286). Have a look at the following examples:

```
int i = 0; // initialise i
i++; // increment i
```

  Also Javadoc that adds no value to a function signature (because it says the same or even less) should be removed:

```
/**
 * Return the name of the product
 * @return The name of the product
 */
public String getName() {
    return name;
}
```

5.1 Implementation Comment Formats

- A programmer should be familiar with the different styles and thus the four types described by Sun are not stated here. Read Sun's convention if you want to know more about the types.

Please consider the following two rules:

- Do not use trailing comments. Use 'Block Comments' instead.
- Remove commented-out code. "That code sits there and rots, getting less and less relevant with every passing day. It calls functions that no longer exist. It uses variables whose names have changed. It follows conventions that are long obsolete. It pollutes the modules that contain it and distracts the people who try to read it. Commented-out code is an abomination" (Martin and Coplien 2009: 287).

5.2 Documentation Comments

- Use JavaDoc for documentation. A programmer should be familiar with JavaDoc, if not, then please read Sun's convention chapter 5.2
- Do not add an author tag (@author) into the documentation comment of a class or interface (or elsewhere).

6 Declarations

6.1 Number per Line

One declaration per line is recommended since it encourages commenting. In other words,

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

is preferred over

```
int level, size;
```

In absolutely no case should variables and functions be declared on the same line. Example:

```
long dbaddr, getDbaddr(); // WRONG!
```

Do not put different types on the same line. Example:

```
int foo,  fooarray[]; //WRONG!
```

In most cases a good chosen name for the variable makes a comment needless. Consider to change the name if you feel to comment what a variable is doing.

Tabs are not recommended. The effort does not really pay off due to the fact that many IDEs will reposition it if the user uses an auto format function.

6.2 Placement

Don't read Sun's convention for this particular point. TSPHP's convention is exactly the contrariwise. Place your variables as late and as close to the relevant code as possible. Methods should be short. If the code discomfits you because the declaration of the variable is too far away you should consider refactoring the method.

6.3 Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first (a buffer is a good example of an exception)

6.4 Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

- No space between a method name and the parenthesis "(" starting its parameter list
- An open brace "{" appears at a new line for class-, enum- and interface-declarations. Otherwise at the end of the same line as the declaration statement.
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object {
    int ivar1;
    int ivar2;
    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }
    int emptyMethod() {}
    ...
}
```

- Methods are separated by a blank line

7 Statements

7.1 Simple Statements

Each line should contain at most one statement. Example:

```
argv++; argc--;        // AVOID!
```

Do not use the comma operator to group multiple statements unless it is for an obvious reason. Example:

```
if (err) {
    Format.print(System.out, "error"), exit(1); //VERY WRONG!
}
```

7.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces `{ statements }`. See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even singletons, when they are part of a control structure, such as an `if-else` or `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

7.3 return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

7.4 if, if-else, if-else-if-else Statements

The `if-else` class of statements should have the following form:

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

Note: if statements always use braces {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

7.5 for Statements

A for statement should have the following form:

```
for (initialization; condition; update) {
    statements;
}
```

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update){}
```

An empty `for` statement should be closed with an open and closing brace “{}”

When using the comma operator in the initialization or update clause of a `for` statement, avoid the complexity of using more than three variables. If needed, use separate statements before the `for` loop (for the initialization clause) or at the end of the loop (for the update clause).

7.6 while Statements

A while statement should have the following form:

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

An empty while statement should have the following form:

```
while (condition){}
```

An empty `while` statement should be closed with an open and closing brace “{}”

7.7 do-while Statements

A do-while statement should have the following form:

```
do {
    statements;
} while (condition);
```

7.8 switch Statements

✎ Intend the case (and default) statements:

```
switch(condition){
    case A:
        ...
        // falls through
    case B:
        ...
        break;
    default:
        ...
        break;
}
```

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the // falls through comment.

Every switch statement should include a `default` case unless the condition is an `enum` type. In this case the `default` statement should be omitted because clever IDEs will give a hint that one has forgotten a case.

✎ The `default` case should always be at the end of a switch statement.

✎ Consider the following smell. "Often you find the same switch statement scattered about a program in different places. If you add a new clause to the switch, you have to find all these switch, statements and change them. The object- oriented notion of polymorphism gives you an elegant way to deal with this problem" (Fowler and Beck 1999: 68).

7.9 try-catch Statements

A try-catch statement should have the following format:

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

8 White Space

8.1 Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:
- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:
- Between methods
- Between the local variables in a method and its first statement
- Before a block or single-line comment
- Between logical sections inside a method to improve readability

8.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

  ```
  while (true) {
  ...
  }
  ```

  Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except . (dot) should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

  ```
  a += c + d;
  a = (a + b) / (c * d);
  while (d++ = s++) {
      n++;
  }
  prints("size is " + foo + "\n");
  ```

- The expressions in a `for` statement should be separated by blank spaces. Example:

  ```
  for (expr1; expr2; expr3){}
  ```

- Casts should be followed by a blank. Examples:

  ```
  myMethod((byte) aNum, (Object) x);
  myFunc((int) (cp + 5), ((int) (i + 3)) + 1);
  ```

9 Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier—for example, whether it's a constant, package, or class—which can be helpful in understanding the code.

Identifier Type	Rules for Naming	Examples

Interfaces	Interface names should be nouns, in mixed case with the first letter of each internal word capitalised (also known as PascalCase). Try to keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).
	⊕ Interface names should be prefixed with an I
		`interface IScope{}`
		`interface ISymbol{}`
		`interface ITypeSymbol{}`

`abstract` classes	`abstract` class names should be capitalised like interface names.
	⊕ `abstract` classes should be prefixed with an A
		`class AScope{}`
		`class AScopedSymbol{}`
		`class AScopedTypeSymbol{}`

Classes	Class names should be capitalised like interface names.
	⊕ Sub classes should contain the name of the parent class. For Instance:
	`class BusinessCar extends ACar`
	`{`
	`}`
		`class NamespaceScope{}`
		`class MethodSymbol{}`
		`class ClassTypeSymbol{}`

⊕ Type parameter	Generic type parameter names begin with an uppercased "T" following an appropriate name as with every variable. The name itself should also begin with an uppercase letter.
	The name should reflect/signify if it has a constraint. For an example
	`class Basket<TArticle extends IArticle>`
	`{`
	`}`
		`class Pair<TKey,TValue>`
		`{`
		`}`

⊕ Enums	Enums should be prefixed with an E.	`enum EDeliveryType{}`
		`enum EGiftType{}`
		`enum ESecurityLevel{}`

Identifier Type	Rules for Naming	Examples

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized (also known as camelCase)
	Method parameters which have the same name as class (static) or instance variables should be prefixed. For instance with "a" or "an" or "the" to avoid bugs as the following one:
	`private String name;`
	`// bad - avoid`
	`public void setName(String name) {`
	`    /* would only assign name to the`
	`     * parameter itself`
	`     */`
	`    name = name;`
	`}`
	
	`//good – bug avoided`
	`public void setName(String aName) {`
	`    name = aName;`
	`}`
		`void run();`
		`void runFast();`
		`String getBackground();`

Variables	Variable names should be in camelCase as well. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic— that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables.
	Prefer Variable longer and meaningful names over short and vague names
		`//prefer`
		`int totalErrors;`
		`//over`
		`int total; //of what?`

Constants	Constants should be all uppercase with words separated by _ (under-scores)	`int MIN_WIDTH = 4;`
		`int MAX_WIDTH = 999;`
		`int GET_THE_CPU = 1;`

Please also have a look into the Test Philosophy to see the naming guidelines for tests.

10 Programming Practices

10.1 Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Often, instance variables don't need to be explicitly set or gotten—often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behaviour. In other words, if you would have used a struct instead of a class (if Java supported struct), then it's appropriate to make the class's instance variables public.

10.2 Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead. For example:

```
classMethod();              //OK
AClass.classMethod();       //OK
anObject.classMethod();     //AVOID!
```

10.3 Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

10.4 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

Do not use the assignment operator in a place where it can be easily confused with the equality operator. Example:

```
if (c++ = d++) {        // AVOID! Java disallows
    ...
}
```

should be written as

```
if ((c++ = d++) != 0) {
    ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler, and besides, it rarely actually helps. Example:

```
d = (a = b + c) + r;        // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

10.5 Miscellaneous Practices

10.5.1 Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others—you shouldn't assume that other programmers know precedence as well as you do.

```
int i = flag << 2 * position;      // AVOID!
int i = flag << (2 * position);    // RIGHT
```

10.5.2 Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

should instead be written as

```
return booleanExpression;
```

Similarly,

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

should be written as (✎ as long as it stays readable)

```
return (condition ? x : y);
```

- Methods should only have one return statement. Very small methods may constitute an exception as long as it is easy to spot the different return statements. As an example, methods which literally split the normal case from error cases constitute such an exception. For instance:

```
private TSPHPAst getAstOrErrorNode(CharStream input) {
    try {
        return getAst(input);
    } catch (RecognitionException ex) {
        return new TSPHPErrorNode(ex);
    }
}
```

or

```
private TSPHPAst getAstOrErrorNode(CharStream input) {
    try {
        return getAst(input);
    } catch (RecognitionException ex) {
        return new TSPHPErrorNode(ex);
    } catch (Exception ex) {
        return new TSPHPFatalErrorNode(ex);
    }
}
```

However, this exception does not apply, if more than one level of abstraction is used. For instance:

```
private TSPHPAst getAstOrErrorNode(CharStream input) {
    if (input.getCharPositionInLine != 0) {
        return getAst(input);
    } else {
        // a
        // few
        // statements
        if (anotherCondition) {
            // second level of abstraction
            return result;
        }
    }
    throw new XyException();
}
```

10.5.3 *Expressions before '?' in the Conditional Operator*

- If an expression containing more than one binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

```
(x >= 0 && x <= 10) ? x : -x
```

10.5.4 *Special Comments*

Use `XXX` in a comment to flag something that is bogus but works. Use `FIXME` to flag something that is bogus and broken.

- Use `TODO` in a comment to indicate an open task. For instance a code review, a security review etc.
- To each `XXX`, `FIXME` or `TODO` belong the member code of the author, the issue number and title of the corresponding issue in the issue tracking system. For instance:

```
// TODO rstoll TSPHP-249 performance review parsing
```

10.5.5 *Overloaded constructors*

"When constructors are overloaded, use static factory methods with names that describe the arguments. For example, `Complex fulcrumPoint = Complex.FromRealNumber(23.0);` is generally better than `Complex fulcrumPoint = new Complex(23.0);` Consider enforcing their use by making the corresponding constructors private" (Martin and Coplien 2009: 25).

10.5.6 *Method length*

Methods longer than 50 lines should be considered as a smell and therefore needs a check if a refactoring is necessary. Maximum length of a method is 80 lines.

10.5.7 *Number of parameters*

Methods with more than 2 parameters should be considered as a smell and therefore needs a check if a refactoring is necessary. Maximum of parameters per method is 4.
Constructors of DTOs constitute an exception.

10.5.8 *File length*

Files longer than 1000 lines should be considered as a smell and therefore needs a check if a refactoring is necessary. Does the class/interface in the file injure the 'Single Responsibility Principle' (Martin and Coplien 2009: 138)?

Maximum length of a file is 1500 lines.

11 Code Examples

11.1 Java Source File Example

Visit GitHub and have a look at the source code. For instance:

https://github.com/tsphp/tsphp-typechecker/blob/dev/src/ch/tsphp/typechecker/DefinitionPhaseController.java

12 List of References

Fowler, M. (2012) *P of EAA: Data Transfer Object.* [online] available from <http://martinfowler.com/eaaCatalog/dataTransferObject.html>[21 Oct 2012].

Fowler, M. and Beck, K. (1999) *Refactoring: Improving the design of existing code.* Reading, MA: Addison-Wesley

Mark, J. (2012) *Silk icons.* [online] available from <http://famfamfam.com/lab/icons/silk/famfamfam_silk_icons_v013.zip>[21 Oct 2012].

Martin, R. C. and Coplien, J. O. (2009) *Clean code: A handbook of agile software craftsmanship.* Upper Saddle River, NJ: Prentice Hall

Sun Microsystems (1997) *Java Code Conventions.* [online] available from <http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>[20 Oct 2012].