Java and Java Script Coding Standard Guidelines

Confidentiality Notice
Copyright (c) 2016 eInfochips. - All rights reserved
This document is authored by eInfochips and is eInfochips intellectual property, including the copyrights in all countries in the world. This document is provided under a license to use only with all other rights, including ownership rights, being retained by eInfochips. This file may not be distributed, copied, or reproduced in any manner, electronic or otherwise, without the express written consent of eInfochips.

CONTENTS

TABLES

1 DOCUMENT DETAILS

1.1 Document History

Version	Author		Reviewer		Approver	
	Name	Date (DD-MMM-YYYY)	Name	Date (DD-MMM-YYYY)	Name	Date (DD-MMM-YYYY)
Baseline 1.0	Himasindhu	30-SEP-2013	*SEPG Manager*	*29-Oct-2013*	*SEPG Manager*	*29-Oct-2013*
Baseline 1.1	Sridhar Pabba	28-May-2014	Bhairavi Shah	29-May-2014	Bijal Chudgar	30-May-2014
Baseline 1.2	Darshik Gajjar	22-Nov-2016	Gayatri Patel	22-Nov-2016	Bijal Chudgar	8-Dec-2016

Version	Description of changes
Baseline 1.0	*Changed from template to guideline*
Baseline 1.1	File Naming Conventions standardized across QMS artifacts (it covers all artifacts that are global, specific to S/W, H/W, ASIC or Aerospace).
Baseline1.2	Added JavaScript Language Rules and others how to use Selector Caching

Table 1: Document History

1.2 Definition, Acronyms and Abbreviations

Definition/Acronym/Abbreviation	Description

Table 2: Definition, Acronyms and Abbreviations

1.3 References

No.	Document	Version	Remarks

Table 3: References

2 PURPOSE

This document describes a collection of standards, conventions and guidelines for writing Java code that is easy to understand, to maintain, and to enhance.

2.1 Important feature of this Document

Existing standards from the industry are used wherever possible
The reason behind each standard is explained so that developers can understand why they should follow it.
These standards are based on proven software-engineering principles that lead to improved development productivity, greater maintainability, and greater scalability.

2.2 Target Audience

Professional Software developers who are involved in:

Writing Java code that is easy to maintain and to enhance
Increasing their productivity

2.3 Why Coding Standard Are Important

Coding standards for Java are important because they lead to greater consistency within code of all developers. Consistency leads to code that is easier to understand, which in turn results in a code, which is easier to develop and maintain. Code that is difficult to understand and maintain runs the risk of being scrapped and rewritten

3 THE PRIME DIRECTIVE

A project requirement may vary from the standards mentioned in this document. When going against the standards, projects should make sure to document it.

3.1 Naming Convention

Use full English descriptors that accurately describe the variable/field/class/interface
For example, use names like firstName, grandTotal, or CorporateCustomer.

Use terminology applicable to the domain
If the users of the system refer to their clients as Customer, then use the term Customer for the class, not client.

Use mixed case to make names readable

Use abbreviations sparingly, but if you do so then use then intelligently and document it
For example, to use a short form for the word “number”, choose one of nbr, no or num.

Avoid long names (<15 characters is a good tradeoff)

Avoid names that are similar or differ only in case

3.2 Documentation

Comments should add to the clarity of code.

Avoid decoration, i.e., do not use banner-like comments
Document why something is being done, not just what.

3.3 Java Comments

Comment Type	Usage	Example
Documentation Starts with /** and ends with */	Used before declarations of interfaces, classes, member functions, and fields to document them.	/** * Customer – a person or * organization */
C style Starts with /* and ends with */	Used to document out lines of code that are no longer applicable. It is helpful in debugging.	/* This code was commented out by Ashish Sarin */
Single line Starts with // and go until the end of the line	Used internally within member functions to document business logic, sections of code, and declarations of temporary variables.	// If the amount is greater // than 10 multiply by 100

Table 4: Java Comments

4 JAVASCRIPT LANGUAGE RULES

4.1 var

Declarations with var: Always
Decision:When you fail to specify var, the variable gets placed in the global context, potentially clobbering existing values. Also, if there's no declaration, it's hard to tell in what scope a variable lives (e.g., it could be in the Document or Window just as easily as in the local scope). So always declare with var.

4.2 Constants

- Use NAMES_LIKE_THIS for constant *values*.

Decision:

Constant values

If a value is intended to be *constant* and *immutable*, it should be given a name in CONSTANT_VALUE_CASE. ALL_CAPS additionally implies @const (that the value is not overwritable).

```
/**
 * Request timeout in milliseconds.
 * @type {number}
 */
goog.example.TIMEOUT_IN_MILLISECONDS = 60;
```

The number of seconds in a minute never changes. It is a constant value. ALL_CAPS also implies @const, so the constant cannot be overwritten.

4.3 Semicolons

Always use semicolons.

Relying on implicit insertion can cause subtle, hard to debug problems. Don't do it. You're better than that.

There are a couple places where missing semicolons are particularly dangerous:

```
// 1.
MyClass.prototype.myMethod = function() {
  return 42;
}  // No semicolon here.

(function() {
  // Some initialization code wrapped in a function to create a scope for
locals.
})();


var x = {
  'i': 1,
  'j': 2
}  // No semicolon here.

// 2.  Trying to do one thing on Internet Explorer and another on Firefox.
// I know you'd never write code like this, but throw me a bone.
[ffVersion, ieVersion][isIE]();


var THINGS_TO_EAT = [apples, oysters, sprayOnCheese]  // No semicolon here.

// 3. conditional execution a la bash
-1 == resultOfOperation() || die();
```

So what happens?

1. JavaScript error - first the function returning 42 is called with the second function as a parameter, then the number 42 is "called" resulting in an error.

2. You will most likely get a 'no such property in undefined' error at runtime as it tries to call `x[ffVersion, ieVersion][isIE]()`.

3. `die` is always called since the array minus 1 is `NaN` which is never equal to anything (not even if `resultOfOperation()` returns `NaN`) and `THINGS_TO_EAT` gets assigned the result of `die()`.

Why?

JavaScript requires statements to end with a semicolon, except when it thinks it can safely infer their existence. In each of these examples, a function declaration or object or array literal is used inside a statement. The closing brackets are not enough to signal the end of the statement. Javascript never ends a statement if the next token is an infix or bracket operator.

This has really surprised people, so make sure your assignments end with semicolons.

Clarification: Semicolons and functions

Semicolons should be included at the end of function expressions, but not at the end of function declarations. The distinction is best illustrated with an example:

```
var foo = function() {
  return true;
};  // semicolon here.

function foo() {
  return true;
}  // no semicolon here.
```

4.4 Function Declarations within Blocks

No do not do this:

```
if (x) {
  function foo() {}
}
```

While most script engines support Function Declarations within blocks it is not part of ECMAScript (see ECMA-262, clause 13 and 14). Worse implementations are inconsistent with each other and with future EcmaScript proposals. ECMAScript only allows for Function Declarations in the root statement list of a script or function. Instead use a variable initialized with a Function Expression to define a function within a block:

```
if (x) {
  var foo = function() {};
}
```

4.5 Standards features

Always preferred over non-standards features

For maximum portability and compatibility, always prefer standards features over non-standards features (e.g., `string.charAt(3)` over `string[3]` and element access with DOM functions instead of using an application-specific shorthand).

4.6 Closures

Yes, but be careful.

The ability to create closures is perhaps the most useful and often overlooked feature of JS. Here is a good description of how closures work.

One thing to keep in mind, however, is that a closure keeps a pointer to its enclosing scope. As a result, attaching a closure to a DOM element can create a circular reference and thus, a memory leak. For example, in the following code:

```
function foo(element, a, b) {
  element.onclick = function() { /* uses a and b */ };
}
```

the function closure keeps a reference to `element`, `a`, and `b` even if it never uses `element`.
Since `element` also keeps a reference to the closure, we have a cycle that won't be cleaned up by garbage collection. In these situations, the code can be structured as follows:

```
function foo(element, a, b) {
  element.onclick = bar(a, b);
}

function bar(a, b) {
  return function() { /* uses a and b */ };
}
```

4.7 eval()

Only for code loaders and REPL (Read–eval–print loop)

`eval()` makes for confusing semantics and is dangerous to use if the string being `eval()`'d contains user input. There's usually a better, clearer, and safer way to write your code, so its use is generally not permitted.

For RPC you can always use JSON and read the result using `JSON.parse()` instead of `eval()`.

Let's assume we have a server that returns something like this:

```
{
  "name": "Alice",
  "id": 31502,
  "email": "looking_glass@example.com"
}
```

```
var userInfo = eval(feed);
var email = userInfo['email'];
```

If the feed was modified to include malicious JavaScript code, then if we use `eval` then that code will be executed.

```
var userInfo = JSON.parse(feed);
var email = userInfo['email'];
```

With `JSON.parse`, invalid JSON (including all executable JavaScript) will cause an exception to be thrown.

4.8 this

Only in object constructors, methods, and in setting up closures

The semantics of `this` can be tricky. At times it refers to the global object (in most places), the scope of the caller (in `eval`), a node in the DOM tree (when attached using an event handler HTML attribute), a newly created object (in a constructor), or some other object (if function was `call()`ed or `apply()`ed).

Because this is so easy to get wrong, limit its use to those places where it is required:

- in constructors
- in methods of objects (including in the creation of closures)

4.9 for-in loop

Only for iterating over keys in an object/map/hash

`for-in` loops are often incorrectly used to loop over the elements in an `Array`. This is however very error prone because it does not loop from `0` to `length - 1` but over all the present keys in the object and its prototype chain. Here are a few cases where it fails:

```
function printArray(arr) {
  for (var key in arr) {
    print(arr[key]);
  }
}

printArray([0,1,2,3]);  // This works.

var a = new Array(10);
printArray(a);  // This is wrong.

a = document.getElementsByTagName('*');
printArray(a);  // This is wrong.

a = [0,1,2,3];
a.buhu = 'wine';
printArray(a);  // This is wrong again.

a = new Array;
a[3] = 3;
printArray(a);  // This is wrong again.
```

Always use normal for loops when using arrays.

```
function printArray(arr) {
  var l = arr.length;
  for (var i = 0; i < l; i++) {
    print(arr[i]);
  }
}
```

4.10 Multiline string literals

No do not do this:

```
var myString = 'A rather long string of English text, an error message \
               actually that just keeps going and going -- an error \
               message to make the Energizer bunny blush (right through \
               those Schwarzenegger shades)! Where was I? Oh yes, \
               you\'ve got an error and all the extraneous whitespace is \
               just gravy.  Have a nice day.';
```

The whitespace at the beginning of each line can't be safely stripped at compile time; whitespace after the slash will result in tricky errors; and while most script engines support this, it is not part of ECMAScript.

Use string concatenation instead:

```
var myString = 'A rather long string of English text, an error message ' +
    'actually that just keeps going and going -- an error ' +
    'message to make the Energizer bunny blush (right through ' +
    'those Schwarzenegger shades)! Where was I? Oh yes, ' +
    'you\'ve got an error and all the extraneous whitespace is ' +
    'just gravy.  Have a nice day.';
```

When a statement is too long to fit on one line, line breaks must occur after an operator.

```
// Bad
var html = '<p>The sum of ' + a + ' and ' + b + ' plus ' + c
    + ' is ' + ( a + b + c );

// Good
var html = '<p>The sum of ' + a + ' and ' + b + ' plus ' + c +
    ' is ' + ( a + b + c );
```

4.11 Chained Method Calls

When a chain of method calls is too long to fit on one line, there must be one call per line, with the first call on a separate line from the object the methods are called on. If the method changes the context, an extra level of indentation must be used.

```
elements
    .addClass( 'foo' )
    .children()
        .html( 'hello' )
    .end()
    .appendTo( 'body' );
```

4.12 Array and Object literals

Yes, Use `Array` and `Object` literals instead of `Array` and `Object` constructors.

Array constructors are error-prone due to their arguments.

```
// Length is 3.
var a1 = new Array(x1, x2, x3);

// Length is 2.
var a2 = new Array(x1, x2);

// If x1 is a number and it is a natural number the length will be x1.
// If x1 is a number but not a natural number this will throw an exception.
// Otherwise the array will have one element with x1 as its value.
var a3 = new Array(x1);

// Length is 0.
var a4 = new Array();
```

Because of this, if someone changes the code to pass 1 argument instead of 2 arguments, the array might not have the expected length.

To avoid these kinds of weird cases, always use the more readable array literal.

```
var a = [x1, x2, x3];
var a2 = [x1, x2];
var a3 = [x1];
var a4 = [];
```

Object constructors don't have the same problems, but for readability and consistency object literals should be used.

```
var o = new Object();

var o2 = new Object();
o2.a = 0;
o2.b = 1;
o2.c = 2;
o2['strange key'] = 3;
```

Should be written as:

```
var o = {};

var o2 = {
  a: 0,
  b: 1,
  c: 2,
```

```
  'strange key': 3
};
```

4.13 Code formatting

Expand for more information.

We follow the C++ formatting rules in spirit, with the following additional clarifications.

Curly Braces

Because of implicit semicolon insertion, always start your curly braces on the same line as whatever they're opening. For example:

```
if (something) {
  // ...
} else {
  // ...
}
```

4.14 Array and Object Initializers

Single-line array and object initializers are allowed when they fit on a line:

```
var arr = [1, 2, 3];  // No space after [ or before ].
var obj = {a: 1, b: 2, c: 3};  // No space after { or before }.
```

Multiline array initializers and object initializers are indented 2 spaces, with the braces on their own line, just like blocks.

```
// Object initializer.
var inset = {
  top: 10,
  right: 20,
  bottom: 15,
  left: 12
};

// Array initializer.
this.rows_ = [
  '"Slartibartfast" <fjordmaster@magrathea.com>',
  '"Zaphod Beeblebrox" <theprez@universe.gov>',
  '"Ford Prefect" <ford@theguide.com>',
  '"Arthur Dent" <has.no.tea@gmail.com>',
  '"Marvin the Paranoid Android" <marv@googlemail.com>',
  'the.mice@magrathea.com'
];

// Used in a method call.
goog.dom.createDom(goog.dom.TagName.DIV, {
  id: 'foo',
  className: 'some-css-class',
  style: 'display:none'
}, 'Hello, world!');
```

Long identifiers or values present problems for aligned initialization lists, so always prefer non-aligned initialization. For example:

```
CORRECT_Object.prototype = {
  a: 0,
  b: 1,
  lengthyName: 2
};
```

Not like this:

```
WRONG_Object.prototype = {
  a           : 0,
  b           : 1,
  lengthyName: 2
};
```

4.15 Object Rules

General rules for object definitions:

- Place the opening bracket on the same line as the object name.
- Use colon plus one space between each property and its value.
- Use quotes around string values, not around numeric values.
- Do not add a comma after the last property-value pair.
- Place the closing bracket on a new line, without leading spaces.
- Always end an object definition with a semicolon.

Example:

```
var person = {
    firstName: "John",
    lastName: "Doe",
    age: 50,
    eyeColor: "blue"
};
```

Short objects can be written compressed, on one line, using spaces only between properties, like this:

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

4.16 Use different namespaces for external code and internal code

"External code" is code that comes from outside your codebase, and is compiled independently. Internal and external names should be kept strictly separate. If you're using an external library that makes things available in `foo.hats.*`, your internal code should not define all its symbols in `foo.hats.*`, because it will break if the other team defines new symbols.

```
foo.require('foo.hats');

/**
 * WRONG -- Do NOT do this.
 * @constructor
 * @extends {foo.hats.RoundHat}
 */
foo.hats.BowlerHat = function() {
};
```

If you need to define new APIs on an external namespace, then you should explicitly export the public API functions, and only those functions. Your internal code should call the internal APIs by their internal names, for consistency and so that the compiler can optimize them better.

```
foo.provide('googleyhats.BowlerHat');

foo.require('foo.hats');
```

```
/**
 * @constructor
 * @extends {foo.hats.RoundHat}
 */
googleyhats.BowlerHat = function() {
  ...
};

goog.exportSymbol('foo.hats.BowlerHat', googleyhats.BowlerHat);
```

4.17 Function Arguments

When possible, all function arguments should be listed on the same line. If doing so would exceed the 80-column limit, the arguments must be line-wrapped in a readable way. To save space, you may wrap as close to 80 as possible, or put each argument on its own line to enhance readability. The indentation may be either four spaces, or aligned to the parenthesis. Below are the most common patterns for argument wrapping:

```
// Four-space, wrap at 80.  Works with very long function names, survives
// renaming without reindenting, low on space.
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
  // ...
};

// Four-space, one argument per line.  Works with long function names,
// survives renaming, and emphasizes each argument.
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne,
    veryDescriptiveArgumentTwo,
    tableModelEventHandlerProxy,
    artichokeDescriptorAdapterIterator) {
  // ...
};

// Parenthesis-aligned indentation, wrap at 80.  Visually groups arguments,
// low on space.
function foo(veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
             tableModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
  // ...
}

// Parenthesis-aligned, one argument per line.  Emphasizes each
// individual argument.
function bar(veryDescriptiveArgumentNumberOne,
             veryDescriptiveArgumentTwo,
             tableModelEventHandlerProxy,
             artichokeDescriptorAdapterIterator) {
  // ...
}
```

When the function call is itself indented, you're free to start the 4-space indent relative to the beginning of the original statement or relative to the beginning of the current function call. The following are all acceptable indentation styles.

```
if (veryLongFunctionNameA(
        veryLongArgumentName) ||
    veryLongFunctionNameB(
    veryLongArgumentName)) {
  veryLongFunctionNameC(veryLongFunctionNameD(
      veryLongFunctioNameE(
          veryLongFunctionNameF)));
```

```
}
```

Passing Anonymous Functions

When declaring an anonymous function in the list of arguments for a function call, the body of the function is indented two spaces from the left edge of the statement, or two spaces from the left edge of the function keyword. This is to make the body of the anonymous function easier to read (i.e. not be all squished up into the right half of the screen).

```
prefix.something.reallyLongFunctionName('whatever', function(a1, a2) {
  if (a1.equals(a2)) {
    someOtherLongFunctionName(a1);
  } else {
    andNowForSomethingCompletelyDifferent(a2.parrot);
  }
});

var names = prefix.something.myExcellentMapFunction(
    verboselyNamedCollectionOfItems,
    function(item) {
      return item.name;
    });
```

4.18 Binary and Ternary Operators

Always put the operator on the preceding line. Otherwise, line breaks and indentation follow the same rules as in other Google style guides. This operator placement was initially agreed upon out of concerns about automatic semicolon insertion. In fact, semicolon insertion cannot happen before a binary operator, but new code should stick to this style for consistency.

```
var x = a ? b : c;  // All on one line if it will fit.

// Indentation +4 is OK.
var y = a ?
    longButSimpleOperandB : longButSimpleOperandC;

// Indenting to the line position of the first operand is also OK.
var z = a ?
        moreComplicatedB :
        moreComplicatedC;
```

This includes the dot operator.

```
var x = foo.bar().
    doSomething().
    doSomethingElse();
```

4.19 Strings

Prefer “over"

For consistency single-quotes (') are preferred to double-quotes ("). This is helpful when creating strings that include HTML:

```
var msg = 'This is some HTML';
```

4.20 Tips and Tricks

JavaScript tidbits

True and False Boolean Expressions

The following are all false in Boolean expressions:

- `null`

- `undefined`
- `''` the empty string
- `0` the number

But be careful, because these are all true:

- `'0'` the string
- `[]` the empty array
- `{}` the empty object

This means that instead of this:

```
while (x != null) {
```

you can write this shorter code (as long as you don't expect x to be 0, or the empty string, or false):

```
while (x) {
```

And if you want to check a string to see if it is null or empty, you could do this:

```
if (y != null && y != '') {
```

But this is shorter and nicer:

```
if (y) {
```

Caution: There are many unintuitive things about boolean expressions. Here are some of them:

- ```
  Boolean('0') == true
  '0' != true
  ```
- ```
  0 != null
  0 == []
  0 == false
  ```
- ```
  Boolean(null) == false
  null != true
  null != false
  ```
- ```
  Boolean(undefined) == false
  undefined != true
  undefined != false
  ```
- ```
  Boolean([]) == true
  [] != true
  [] == false
  ```
- ```
  Boolean({}) == true
  {} != true
  {} != false
  ```

Conditional (Ternary) Operator (?:)

Instead of this:

```
if (val) {
  return foo();
} else {
  return bar();
}
```

you can write this:

```
return val ? foo() : bar();
```

The ternary conditional is also useful when generating HTML:

```
var html = '<input type="checkbox"' +
    (isChecked ? ' checked' : '') +
    (isEnabled ? '' : ' disabled') +
    ' name="foo">';
```

&& and ||

These binary boolean operators are short-circuited, and evaluate to the last evaluated term.

"||" has been called the 'default' operator, because instead of writing this:

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win;
  if (opt_win) {
    win = opt_win;
  } else {
    win = window;
  }
  // ...
}
```

you can write this:

```
/** @param {*=} opt_win */
function foo(opt_win) {
  var win = opt_win || window;
  // ...
}
```

"&&" is also useful for shortening code. For instance, instead of this:

```
if (node) {
  if (node.kids) {
    if (node.kids[index]) {
      foo(node.kids[index]);
    }
  }
}
```

you could do this:

```
if (node && node.kids && node.kids[index]) {
  foo(node.kids[index]);
}
```

or this:

```
var kid = node && node.kids && node.kids[index];
if (kid) {
  foo(kid);
}
```

However, this is going a little too far:

```
node && node.kids && node.kids[index] && foo(node.kids[index]);
```

Iterating over Node Lists

Node lists are often implemented as node iterators with a filter. This means that getting a property like length is O(n), and iterating over the list by re-checking the length will be O(n^2).

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0; i < paragraphs.length; i++) {
  doSomething(paragraphs[i]);
}
```

It is better to do this instead:

```
var paragraphs = document.getElementsByTagName('p');
for (var i = 0, paragraph; paragraph = paragraphs[i]; i++) {
  doSomething(paragraph);
}
```

This works well for all collections and arrays as long as the array does not contain things that are treated as boolean false.

In cases where you are iterating over the childNodes you can also use the firstChild and nextSibling properties.

```
var parentNode = document.getElementById('foo');
for (var child = parentNode.firstChild; child; child = child.nextSibling) {
  doSomething(child);
}
```

4.21 JavaScript Files

JavaScript programs should be stored in and delivered as .js files.

JavaScript code should not be embedded in HTML files unless the code is specific to a single session. Code in HTML adds significantly to pageweight with no opportunity for mitigation by caching and compression.

<script src=*filename*.js> tags should be placed as late in the body as possible. This reduces the effects of delays imposed by script loading on other page components. There is no need to use the language or type attributes. It is the server, not the script tag, that determines the MIME type.

4.22 Line Length

Avoid lines longer than 80 characters. When a statement will not fit on a single line, it may be necessary to break it. Place the break after an operator, ideally after a comma. A break after an operator decreases the likelihood that a copy-paste error will be masked by semicolon insertion. The next line should be indented 8 spaces.

4.23 Variable Declarations

All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied globals. Implied global variables should never be used. Use of global variables should be minimized.

4.24 Avoid Global Variables

Minimize the use of global variables.

This includes all data types, objects, and functions.

Global variables and functions can be overwritten by other scripts.

Use local variables instead, and learn how to use closures.

The `var` statement should be the first statement in the function body.

It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order if possible.

```
var currentEntry, // currently selected table entry
    level,        // indentation level
    size;         // size of table
```

JavaScript does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages. Define all variables at the top of the function.

4.25 Declarations on Top

It is a good coding practice to put all declarations at the top of each script or function.

This will:

Give cleaner code

Provide a single place to look for local variables

Make it easier to avoid unwanted (implied) global variables

Reduce the possibility of unwanted re-declarations

```
// Declare at the beginning
var firstName, lastName, price, discount, fullPrice;

// Use later
firstName = "John";
lastName = "Doe";

price = 19.90;
discount = 0.10;

fullPrice = price * 100 / discount;
```

4.26 Initialize Variables

It is a good coding practice to initialize variables when you declare them.

This will:

Give cleaner code

Provide a single place to initialize variables

Avoid undefined values

4.27 Never Declare Number, String, or Boolean Objects

Always treat numbers, strings, or booleans as primitive values. Not as objects.

Declaring these types as objects, slows down execution speed, and produces nasty side effects:

4.28 Example

```
var x = "John";
var y = new String("John");
(x === y) // is false because x is a string and y is an object.
```

4.29 Function Declarations

All functions should be declared before they are used. Inner functions should follow the `var` statement. This helps make it clear what variables are included in its scope.

There should be no space between the name of a function and the ((left parenthesis) of its parameter list. There should be one space between the) (right parenthesis) and the { (left curly brace) that begins the statement body. The body itself is indented four spaces. The } (right curly brace) is aligned with the line containing the beginning of the declaration of the function.

```
function outer(c, d) {
    var e = c * d;

    function inner(a, b) {
        return (e * a) + b;
    }

    return inner(0, 1);
}
```

This convention works well with JavaScript because in JavaScript, functions and object literals can be placed anywhere that an expression is allowed. It provides the best readability with inline functions and complex structures.

```
    function getElementsByClassName(className) {
        var results = [];
        walkTheDOM(document.body, function (node) {
            var array,                // array of class names
                ncn = node.className; // the node's classname

// If the node has a class name, then split it into a list of simple names.
// If any of them match the requested name, then append the node to the list of
results.

            if (ncn && ncn.split(' ').indexOf(className) >= 0) {
                        results.push(node);
            }
        });
        return results;
    }
```

If a function literal is anonymous, there should be one space between the word `function` and the ((left parenthesis). If the space is omitted, then it can appear that the function's name is `function`, which is an incorrect reading.

```
    div.onclick = function (e) {
        return false;
    };

    that = {
        method: function () {
            return this.datum;
        },
        datum: 0
    };
```

Use of global functions should be minimized.

When a function is to be invoked immediately, the entire invocation expression should be wrapped in parens so that it is clear that the value being produced is the result of the function and not the function itself.

```
var collection = (function () {
    var keys = [], values = [];

    return {
        get: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                return values[at];
            }
        },
        set: function (key, value) {
            var at = keys.indexOf(key);
            if (at < 0) {
                at = keys.length;
            }
            keys[at] = key;
            values[at] = value;
        },
        remove: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                keys.splice(at, 1);
                values.splice(at, 1);
            }
        }
    };
}());
```

Put the opening bracket at the end of the first line.

Do not end a complex statement with a semicolon.

4.30 Names

Names should be formed from the 26 upper and lower case letters (A .. Z, a .. z), the 10 digits (0 .. 9), and _ (underbar). Avoid use of international characters because they may not read well or be understood everywhere. Do not use $ (dollar sign) or \ (backslash) in names.

Do not use _ (underbar) as the first or last character of a name. It is sometimes intended to indicate privacy, but it does not actually provide privacy. If privacy is important, use the forms that provide private members. Avoid conventions that demonstrate a lack of competence.

Most variables and functions should start with a lower case letter.

Constructor functions that must be used with the `new` prefix should start with a capital letter. JavaScript issues neither a compile-time warning nor a run-time warning if a required new is omitted. Bad things can happen if new is not used, so the capitalization convention is the only defence we have.

Global variables should be in all caps. (JavaScript does not have macros or constants, so there isn't much point in using all caps to signify features that JavaScript doesn't have.)

4.31 === and !== Operators.

Use the === and !== operators. The == and != operators do type coercion and should not be used.

5 PERFORMANCE

5.1 Caching selectors for long periods of time.

Reusing selectors is a good thing. For example, you could refactor this:

```
var initMenu = function()
{
    $('.menu').show();
    $('.menu').addClass('enabled');
    $('.menu').click(handleClick);
};
```

Into this:

```
var initMenu = function()
{
    var $menu = $('.menu'); // Cache the selector
    $menu.show();
    $menu.addClass('enabled');
    $menu.click(handleClick);
};
```

6 STANDARDS FOR MEMBER FUNCTIONS

6.1 Naming member functions

Member functions should be named using a full English description, using mixed case with the first letter of any non-initial word capitalized. The first word of the member function should be a verb.

Examples:

openAccount()
printMailingList()
save()
delete()

This results in member functions whose purpose can be determined just by looking at its name.

6.1.1 Naming Accessor Member Functions

6.1.1.1 Getters:

member functions that return the value of a field / attribute / property of an object.

Use prefix “get” to the name of the field / attribute / property if the field in not boolean
Use prefix “is” to the name of the field / attribute / property if the field is Boolean
A viable alternative is to use the prefix ‘has’ or ‘can’ instead of ‘is’ for boolean getters.

Examples
getFirstName()
isPersistent()

6.1.1.2 Setters

member functions that modify the values of a field.

Use prefix ‘set’ to the name of the field.

Examples
setFirstName()

6.1.1.3 Constructors

member functions that perform any necessary initialization when an object is created. Constructors are always given the same name as their class.

Examples
Customer()
SavingsAccount()

6.2 Member Function Visibility

A good design requires minimum coupling between the classes. The general rule is to be as restrictive as possible when setting the visibility of a member function. If member function doesn’t have to be public then make it protected, and if it doesn’t have to be protected than make it private.

6.3 Documenting Member Functions

6.3.1 Member Function Header

Member function documentation should include the following:

What and why the member function does what it does
What member function must be passed as parameters
What a member function returns
Known bugs
Any exception that a member function throws
Visibility decisions (if questionable by other developers)
How a member function changes the object – it is to helps a developer to understand how a member function invocation will affect the target object.
Include a history of any code changes
Examples of how to invoke the member function if appropriate.
Applicable pre conditions and post conditions under which the function will work properly. These are the assumptions made during writing of the function.
All concurrency issues should be addressed.
- Explanation of why keeping a function synchronized must be documented.

When a member function updates a field/attribute/property, of a class that implements the **Runnable** interface, is not synchronized then it should be documented why it is unsynchronized.
If a member function is overloaded or overridden or synchronization changed, it should also be documented.

Note: It's not necessary to document all the factors described above for each and every member function because not all factors are applicable to every member function.

6.3.2 Internal Documentation

Comments within the member functions

Use C style comments to document out lines of unneeded code.
Use single-line comments for business logic.

Internally following should be documented:

Control Structures This includes comparison statements and loops
Why, as well as what, the code does
Local variables
Difficult or complex code
The processing order If there are statements in the code that must be executed in a defined order

6.3.3 Document the closing braces

There are many control structures one inside another

7 TECHNIQUES FOR WRITING CLEAN CODE

Document the code Already discussed above
Paragraph/Indent the code: Any code between the { and } should be properly indented
Paragraph and punctuate multi-line statements
Example

```
Line 1   BankAccount newPersonalAccount = AccountFactory
Line 2                    createBankAccountFor(currentCustomer, startDate,
Line 3                    initialDeposit, branch)
```

Lines 2 & 3 have been indented by one unit (horizontal tab)

Use white space

A few blank lines or spaces can help make the code more readable.

Single blank lines to separate logical groups of code, such as control structures
Two blank lines to separate member function definitions

Specify the order of Operations: Use extra parenthesis to increase the readability of the code using AND OR comparisons. This facilitates in identifying the exact order of operations in the code

Write short, single command lines Code should do one operation per line So only one statement should be there per line

8 STANDARDS FOR FIELDS (ATTRIBUTES / PROPERTIES)

8.1 Naming Fields

Use a Full English Descriptor for Field Names
Fields that are collections, such as arrays or vectors, should be given names that are plural to indicate that they represent multiple values.

Examples
firstName
orderItems

If the name of the field begins with an acronym then the acronym should be completely in lower case

Example
Sql Database

8.2 Naming Components

Use full English descriptor postfixed by the widget type. This makes it easy for a developer to identify the purpose of the components as well as its type.

Example

okButton
customerList
fileMenu
newFileMenuItem

8.3 Naming Constants

In Java, constants, values that do not change, are typically implemented as static final fields of classes. The convention is to use full English words, all in upper case, with underscores between the words

Example
MINIMUM_BALANCE
MAX_VALUE
DEFAULT_START_DATE

8.4 Field Visibility

Fields should not be declared public for reasons of encapsulation. All fields should be declared private and accessor methods should be used to access / modify the field value. This results in less coupling between classes as the protected / public / package access of field can result in direct access of the field from other classes

8.5 Documenting a Field

Document the following:

It's description
Document all applicable invariants of a field are the conditions that are always true about it. By documenting the restrictions on the values of a field one can understand important business rules, making it easier to understand how the code works / how the code is supposed to work
Examples For fields that have complex business rules associated with them one should provide several example values so as to make them easier to understand
Concurrency issues
Visibility decisions If a field is declared anything but private then it should be documented why it has not been declared private.

8.6 Usage of Accesors

Accessors can be used for more than just getting and setting the values of instance fields. Accesors should be used for following purpose also:

Initialize the values of fields Use lazy initialization where fields are initialized by their getter member functions.

Example

```
/**
  * Answer the branch number, which is the leftmost four digits of the full account
  * number. Account numbers are in the format BBBBAAAAAA.
  */
protected int getBranchNumber()
{
        if(branchNumber == 0)
        {
                // The default branch number is 1000, which is the
                // main branch in downtown Bedrock
                setBranchNumber(1000);
        }
        return branchNumber;
}
```

Note:
This approach is advantageous for objects that have fields that aren't regularly accessed
Whenever lazy initialization is used in a getter function the programmer should document what is the type of default value, what the default value as in the example above.

8.6.1 Access constant values

Commonly constant values are declared as static final fields. This approach makes sense for "constants" that are stable.

If the constants can change because of some changes in the business rules as the business matures then it is better to use getter member functions for constants.

By using accesors for constants programmer can decrease the chance of bugs and at the same time increase the maintainability of the system.

8.6.2 Access Collections

The main purpose of accesors is to encapsulate the access to fields so as to reduce the coupling within the code. Collections, such as arrays and vectors, being more complex than single value fields have more than just standard getter and setter member function implemented for them. Because the business rule may require to add and remove to and from collections, accessor member functions need to be included to do so.

Example

Member function type	Naming Convention	Example
Getter for the collection	getCollection()	getOrderItems()
Setter for the collection	setCollection()	setOrderItems()
Insert an object into the collection	insertObject()	insertOrderItems()
Delete an object from the collection	deleteObject()	deleteOrderItems()
Create and add a new object into the collection	newObject()	newOrderItem()

Table 5: Access Collection

Note

The advantage of this approach is that the collection is fully encapsulated, allowing programmer to later replace it with another structure

It is common to that the getter member functions be public and the setter be protected

Always Initialize Static Fields because one can't assume that instances of a class will be created before a static field is accessed

9 STANDARDS FOR LOCAL VARIABLES

9.1 Naming Local Variables

Use full English descriptors with the first letter of any non-initial word in uppercase.

9.1.1 Naming Streams

When there is a single input and/or output stream being opened, used, and then closed within a member function the convention is to use **in** and **out** for the names of these streams, respectively.

9.1.2 Naming Loop Counters

A common way is to use words like **loopCounters** or simply **counter** because it helps facilitate the search for the counters in the program.
i, j, k can also be used as loop counters but the disadvantage is that search for i ,j and k in the code will result in many hits.

9.1.3 Naming Exception Objects

The use of letter **e** for a generic exception

9.2 Declaring and Documenting Local Variables

Declare one local variable per line of code
Document local variable with an endline comment
Declare local variables immediately before their use
Use local variable for one operation only. Whenever a local variable is used for more than one reason, it effectively decreases its cohesion, making it difficult to understand. It also increases the chances of introducing bugs into the code from unexpected side effects of previous values of a local variable from earlier in the code.

Note
Reusing local variables is more efficient because less memory needs to be allocated, but reusing local variables decreases the maintainability of code and makes it more fragile

10 STANDARDS FOR PARAMETERS (ARGUMENTS) TO MEMBER FUNCTIONS

10.1 Naming Parameters

Parameters should be named following the exact same conventions as for local variable

Name parameters the same as their corresponding fields (if any)

Example
If **Account** has an attribute called **balance** and you needed to pass a parameter representing a new value for it the parameter would be called **balance** The field would be referred to as **this.balance** in the code and the parameter would be referred as **balance**

10.2 Documenting Parameters

Parameters to a member function are documented in the header documentation for the member function using the javadoc @param tag. It should describe:
What it should be used for
Any restrictions or preconditions
Examples If it is not completely obvious what a parameter should be, then it should provide one or more examples in the documentation

Note

Use interface as a parameter to the member function then the object itself.

11 STANDARDS FOR CLASSES

11.1 Class Visibility

Use package visibility for classes internal to a component
Use public visibility for the façade of components

11.2 Naming classes

Use full English descriptor starting with the first letter capitalized using mixed case for the rest of the name

11.3 Documenting a Class

The purpose of the class
Known bugs
The development/maintenance history of the class
Document applicable variants
The concurrency strategy Any class that implements the interface **Runnable** should have its concurrency strategy fully described

9.4 Ordering Member Functions and Fields
The order should be:
Constructors
private fields
public member functions
protected member functions
private member functions
finalize()

12 STANDARDS FOR INTERFACES

12.1 Naming Interfaces

Name interfaces using mixed case with the first letter of each word capitalized.
Prefix the letter “I” or “Ifc” to the interface name

12.2 Documenting Interfaces

The Purpose
How it should and shouldn’t be used

13 STANDARDS FOR PACKAGES

Local packages names begin with an identifier that is not all upper case
Global package names begin with the reversed Internet domain name for the organization
Package names should be singular.

13.1 Documenting a Package

The rationale for the package
The classes in the packages

14 STANDARDS FOR COMPILATION UNIT (SOURCE CODE FILE)

14.1 Naming a Compilation Unit

A compilation unit should be given the name of the primary class or interface that is declared within it. Use the same name of the class for the file name, using the same case.

14.2 Beginning Comments

```
/**
  * Classname
  *
  * Version information
  *
  * Date
  *
  * Copyright notice
  */
```

14.3 Declaration

Class/interface documentation comment (/**...*/)	See Documentation standard for class / interfaces
Class or interface statement	
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.
Class (static) variables	First the public class variables, then the protected, then package level (no access modifier), and then the private.
Instance variables	First public, then protected, then package level (no access modifier), and then private.
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.

Table 6: Declaration

14.4 Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

14.5 Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length-generally no more than 70 characters.

14.6 Wrapping Lines

When an expression will 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:

```
someMethod(longExpression1, longExpression2, longExpression3,
            longExpression4, longExpression5);
                        var = someMethod1(longExpression1,
            someMethod2(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 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;
```

14.7 Declaration

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;
```

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

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

Note: The examples above use one space between the type and the identifier. Another acceptable alternative is to use tabs, e.g.:

```
int         level;              // indentation level
int         size;                // size of table
Object      currentEntry;               // currently selected table entry
```

14.8 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.

14.9 Placement

Put declarations only at the beginning of blocks. (A block is any code surrounded by curly braces "{" and "}".) Don't wait to declare variables until their first use; it can confuse the unwary programmer and hamper code portability within the scope.

```
void myMethod() {
int int1 = 0;         // beginning of method block

   if (condition) {
      int int2 = 0;     // beginning of "if" block
      ...
   }
}
```

The one exception to the rule is indexes of for loops, which in Java can be declared in the for statement:

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
myMethod() {
   if (condition) {
      int count = 0;     // AVOID!
      ...
   }
   ...
}
```

14.10 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
Open brace "{" appears 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() {}

    ...
}
```

14.11 Statements

14.11.1 Simple Statements

Each line should contain at most one statement.

Example:

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

14.11.2 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 single statements, when they are part of a control structure, such as a if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

14.11.3 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);

14.11.4 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 {
   statements;
}
```

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

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

14.11.5 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);
```

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).

14.11.6 while Statements

A while statement should have the following form:

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

An empty while statement should have the following form:
while (condition);

14.11.7 do-while Statements

A do-while statement should have the following form:

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

14.11.8 switch Statements

A switch statement should have the following form:

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

case DEF:
   statements;
   break;

case XYZ:
   statements;
   break;

default:
   statements;
   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. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

14.11.9 try-catch Statements

A try-catch statement should have the following format:

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

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

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

14.11.10 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

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 . 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++;
}
printSize("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 space. Examples:

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

15 NAMING CONVENTIONS SUMMARY

Identifier Type	Rules for Naming	Examples
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. 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).	class Raster; class ImageSprite;
Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate; interface Storing;

Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	run();
		runFast();
		getBackground();

Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. 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. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.	int i; char c; float myWidth;

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 999;
		static final int GET_THE_CPU = 1;