1.

General Principles

While it is important to write software that performs well, many other issues should concern the professional Java developer. All *good* software performs well. But *great* software, written with style, is predictable, robust, maintainable, supportable, and extensible.

1. *Adhere to the style of the original.*

When modifying existing software, your changes should follow the style of the original code.[1] Do not introduce a new coding style in a modification, and do not attempt to rewrite the old software just to make it match the new style. The use of different styles within a single source file produces code that is more difficult to read and comprehend. Rewriting old code simply to change its style may result in the introduction of costly yet avoidable defects.

2. *Adhere to the Principle of Least Astonishment.*

The *Principle of Least Astonishment* suggests you should avoid doing things that will surprise a user of your software. This implies the means of interaction and the behavior exhibited by your software must be predictable and consistent,[2] and, if not, the documentation must clearly identify and justify any unusual patterns of use or behavior.

To minimize the chances that a user will encounter something surprising in your software, you should emphasize the following characteristics in the design, implementation, and documentation of your Java software:

Simplicity	Build simple classes and simple methods. Determine how much you need to do to meet the expectations of your users.
Clarity	Ensure each class, interface, method, variable, and object has a clear purpose. Explain where, when, why, and how to use each.
Completeness	Provide the minimum functionality that any reasonable user would expect to find and use. Create complete documentation; document all features and functionality.
Consistency	Similar entities should look and behave the same; dissimilar entities should look and behave differently. Create and apply standards whenever possible.
Robustness	Provide predictable documented behavior in response to errors and exceptions. Do not hide errors and do not force clients to detect errors.

3. *Do it right the first time.*

Apply these rules to any code you write, not just code destined for production. More often than not, some piece of prototype or experimental code will make its way into a finished product, so you should anticipate this eventuality. Even if your code never makes it into production, someone else may still have to read it. Anyone who must look at your code will appreciate your professionalism and foresight at having consistently applied these rules from the start.

4. *Document any deviations.*

No standard is perfect and no standard is universally applicable. Sometimes you will find yourself in a situation where you need to deviate from an established standard.

Before you decide to ignore a rule, you should first make sure you understand why the rule exists and what the consequences are if it is not applied. If you decide you must violate a rule, then document why you have done so.

This is the *prime directive.*

---

1 Jim Karabatsos. "When does this document apply?" In "Visual Basic Programming Standards." (GUI Computing Ltd., 22 March 1996). Accessed online at http://www.gui.com.au/jkcoding.htm, Aug 1999.

2 George Brackett. "Class 6: Designing for Communication: Layout, Structure, Navigation for Nets and Webs." In "Course T525: Designing Educational Experiences for Networks and Webs." (Harvard Graduate School of Education, 26 August 1999). Accessed online at http://hgseclass.harvard.edu/t52598/classes/class6/, Aug 1999.

2.

Formatting Conventions

5. Indent nested code.

One way to improve code readability is to group individual statements into block statements and uniformly indent the content of each block to set off its contents from the surrounding code.

If you generate code using a Java development environment, use the indentation style produced by the environment. If you are generating the code by hand, use two spaces to ensure readability without taking up too much space:

```
class MyClass {
··void function(int arg) {
····if (arg < 0) {
······for (int index = 0; index <= arg; index++) {
········// ...
····· }
···· }
·· }
}
```

In addition to indenting the contents of block statements, you should also indent the statements that follow a label to make the label easier to notice:

```
void function(int arg) {
··loop:
····for (int index = 0; index <= arg; index++) {
·····switch (index) {
```

```
        ........case 0:
        ..........//...
        ..........goto loop; // exit the for statement
        ........default:
        ..........//...
        ..........break; // exit the switch statement
        ......}
        ....}
    }
```

Locate the opening brace '{' of each block statement in the last character position of the line that introduced the block. Place the closing brace '}' of a block on a line of its own, aligned with the first character of the line that introduced the block. The following examples illustrate how this rule applies to each of the various Java definition and control constructs.

Class definitions:

```
public class MyClass {
  ...
}
```

Inner class definitions:

```
public class OuterClass {
  ...
  class InnerClass {
    ...
  }
  ...
}
```

Method definitions:

```
void method(int j) {
  ...
}
```

Static blocks:

```
static {
  ...
}
```


For-loop statements:

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

If and else statements:

```
if (j < 0) {
  ...
}
else if (j > 0) {
  ...
}
else {
  ...
}
```

Try, catch, and finally blocks:

```
try {
  ...
}
catch (Exception e) {
  ...
}
finally {
  ...
}
```

Switch statements:

```
switch (value) {
  case 0:
    ...
    break;
  default:
    ...
    break;
}
```

Anonymous inner classes:

```
button.addActionListener(
  new ActionEventListener() {
```

```
        public void actionPerformed() {
          ...
        }
      }
    }
```

While statements:

```
while (++k <= j) {
  ...
}
```

Do-while statements:

```
do {
  ...
} while (++k <= j);
```

If you are managing a development team, do not leave it up to individual developers to choose their own indentation amount and style. Establish a standard indentation policy for the organization and ensure that everyone complies with this standard.

Our recommendation of two spaces appears to be the most common standard, although your organization may prefer three or even four spaces.

6. *Break up long lines.*

While a modern window-based editor can easily handle long source code lines by scrolling horizontally, a printer must truncate, wrap, or print on separate sheets any lines that exceed its maximum printable line width. To ensure your source code is still readable when printed, you should limit your source code line lengths to the maximum width your printing environment supports, typically 80 or 132 characters.

First, do not place multiple statement expressions on a single line if the result is a line that exceeds your maximum allowable line length. If two statement expressions are placed on one line:


```
double x = Math.random(); double y = Math.ran
dom(); // Too Long!
```

Then introduce a new line to place them on separate lines:

```
double x = Math.random();
double y = Math.random();
```

Second, if a line is too long because it contains a complex expression:

```
double length = Math.sqrt(Math.pow(Math.random(),
2.0) + Math.pow(Math.random(), 2.0)); // Too Long!
```

Then subdivide that expression into several smaller subexpressions. Use a separate line to store the result produced by an evaluation of each subexpression into a temporary variable:

```
double xSquared = Math.pow(Math.random(),2.0);
double ySquared = Math.pow(Math.random(),2.0);
double length = Math.sqrt(xSquared+ySquared);
```

Last, if a long line cannot be shortened under the previous two guidelines, then break, wrap, and indent that line using the following rules:

Step one

If the top-level expression on the line contains one or more commas:

```
double length = Math.sqrt(Math.pow(x, 2.0),Math
.pow(y,2.0)); // Too Long!
```

Then introduce a line break after each comma. Align each expression following a comma with the first character of the expression proceeding the comma:

```
double length = Math.sqrt(Math.pow(x,2.0),
                          Math.pow(y,2.0));
```

Step two

If the top-level expression on the line contains no commas:

```
class MyClass {
  private int field;
```

```
  ...
  boolean equals(Object obj) {
    return this == obj || (obj instanceof MyClass
&& this.field == obj.field); // Too Long!
  }
  ...
}
```

Then introduce a line break just before the operator with the lowest precedence or, if more than one operator of equally low precedence exists between each such operator:

```
class MyClass {
  private int field;
  ...
  boolean equals(Object obj) {
    return this == obj
        || (this.obj instanceof MyClass
            && this.field == obj.field);
  }
  ...
}
```

Step three

Reapply steps one and two, as required, until each line created from the original statement expression is less than the maximum allowable length.

7. *Include white space.*

White space is the area on a page devoid of visible characters. Code with too little white space is difficult to read and understand, so use plenty of white space to delineate methods, comments, code blocks, and expressions clearly.

Use a single space to separate:

- A right parenthesis ')' or curly brace '}' and any keyword that immediately follows, a keyword and any left parenthesis '(' or curly brace '{' that immediately follows, or a right parenthesis ')' and any left curly brace '{' that immediately follows:


```
for·(...)·{
  ...
}

while·(...)·{
  ...
}

do·{
  ...
}·while·(...);

switch·(...)·{
  ...
}

if·(...)·{
  ...
}
else·if·(...)·{
  ...
}
else·{
  ...
}

try·{
  ...
}
catch·(...)·{
  ...
}
finally·{
  ...
}
```

- Any binary operator, except for the "." qualification operator and the expression that proceeds and the expression that follows the operator:

```
double length = Math.sqrt(x * x + y * y);
double xNorm = length > 0.0 ? (x / length) : x;
```

Use blank lines to separate:

- Each logical section of a method implementation:

```
void handleMessage(Message messsage) {

  DataInput content = message.getDataInput();
  int messageType = content.readInt();

  switch (messageType) {

    case WARNING:
      ... do some stuff here ...
      break;

    case ERROR:
      ... do some stuff here ...
      break;

    default:
      ... do some stuff here ...
      break;
  }
}
```

- Each member of a class and/or interface definition:

```
public class Foo {

  /**
   * Defines an inner class.
   */
  class InnerFoo {
    ...
  }
```



```
  /**
   * The Bar associated with this Foo.
   */
  private Bar bar;

  /**
   * Construct a Foo with the specified Bar.
   */
  Foo(Bar bar) {
    this.bar = bar;
  }
}
```

- Each class and interface definition in a source file:

```
/*
 * ... file description ...
 */

package com.company.xyz;

/**
 * ... interface description ...
 */
interface FooInterface {
  ...
}

/**
 * ... class description ...
 */
public class Foo implements FooInterface {
  ...
}
```

8. *Do not use "hard" tabs.*

Many developers use tab characters to indent and align their source code, without realizing the interpretation of tab characters varies across environments. Code that appears to possess the correct formatting when viewed in the original editing environment can appear unformatted and virtually

unreadable when transported to an environment that interprets tabs differently.

To avoid this problem, always use spaces instead of tabs to indent and align source code. You may do this simply by using the space bar instead of the tab key or by configuring your editor to replace tabs with spaces. Some editors also provide a "smart" indentation capability. You will need to disable this feature if it uses tab characters.

Your organization should set a common indentation size and apply it consistently to all its Java code as outlined in Rule #5.

3.

Naming Conventions

THE FOLLOWING naming conventions are identical to those used by SUN MICROSYSTEMS in naming the identifiers that appear in the *Java Software Development Kit.*[3,4]

9. *Use meaningful names.*

When you name a class, variable, method, or constant, use a name that is, and will remain, meaningful to those programmers who must eventually read your code. Use meaningful words to create names. Avoid using a single character or generic names that do little to define the purpose of the entities they name.

The purpose for the variable "a" and the constant "65" in the following code is unclear:

```
if (a < 65) { // What property does 'a' describe?
  y = 65 - a; // What is being calculated here?
}
else {
  y = 0;
}
```

The code is much easier to understand when meaningful names are used:

```
if (age < RETIREMENT_AGE) {
  yearsToRetirement = RETIREMENT_AGE - age;
}
else {
  yearsToRetirement = 0;
}
```

The only exception to this rule concerns temporary variables whose context provides sufficient information to determine

their purpose, such as a variable used as a counter or index within a loop:

```
for (int i = 0; i < numberOfStudents; ++i) {
  enrollStudent(i);
}
```

Some variable meanings and use scenarios occur frequently enough to be standardized.

(See Rule #28 for more information.)

10. Use familiar names.

Use words that exist in the terminology of the target domain. If your users refer to their clients as customers, then use the name `Customer` for the class, not `Client`. Many developers will make the mistake of creating new or generic terms for concepts when satisfactory terms already exist in the target industry or domain.

11. Question excessively long names.

The name given an object must adequately describe its purpose. If a class, interface, variable, or method has an overly long name, then that entity is probably trying to accomplish too much.

Instead of simply giving the entity a new name that conveys less meaning, first reconsider its design or purpose. A refactoring of the entity may produce new classes, interfaces, methods, or variables that are more focused and can be given more meaningful yet simpler names.

12. Join the vowel generation.

Abbreviations reduce the readability of your code and introduce ambiguity if more than one meaningful name reduces to the same abbreviation.

Do not attempt to shorten names by removing vowels. This practice reduces the readability of your code and introduces ambiguity if more than one meaningful name reduces to the same consonants.

The casual reader can understand the names in this definition:

```
public Message appendSignature(Message message,
                               String signature) {
  ...
}
```

While the shortened forms are more difficult to read:

```
public Msg appndSgntr(Msg msg,
                      String sgntr) {
  ...
}
```

If you remove vowels simply to shorten a long name, then you need to question whether the original name is appropriate. See Rule #11.

13. Capitalize only the first letter in acronyms.

This style helps to eliminate confusion in names where uppercase letters act as word separators, and it is especially important if one acronym immediately follows another:

```
setDSTOffset()          setDstOffset()
loadXMLDocument()       loadXmlDocument()
```

This rule does not apply to:

- Acronyms that appear within the name of a constant as these names only contain capital letters (see Rule #31):

```
static final String XML_DOCUMENT = "text/XML";
```

- Acronyms that appear at the beginning of a method, variable or parameter name, as these names should always start with a lowercase letter (see Rules #22 and #25):

```
private Document xmlDocument;
```

14. Do not use names that differ only in case.

The Java compiler can distinguish between names that differ only in case, but a human reader may fail to notice the difference.

For example, a variable named **`theSQLInputStream`** should not appear in the same scope as a variable named **`theSqlInputStream`**. If both names appear in the same scope, each effectively hides the other when considered from the perspective of a person trying to read and understand the code.[5]

Package Names

15. Use the reversed, lowercase form of your organization's Internet domain name as the root qualifier for your package names.

Any package distributed to other organizations should include the lowercase domain name of the originating organization, in reverse order.[6] For example, if a company named ROGUE WAVE SOFTWARE, whose Internet domain name is roguewave.com, decides to distribute an application server package called `server`, then Rogue Wave would name that package `com.roguewave.server`.

SUN MICROSYSTEMS has placed restrictions on the use of the package qualifier names `java` and `javax`. The `java` package qualifier may only be used by Java vendors to provide conforming implementations of the standard Java class libraries. SUN MICROSYSTEMS reserves the name `javax` for use in qualifying its own Java extension packages.

16. Use a single, lowercase word as the root name of each package.

The qualified portion of a package name should consist of a single, lowercase word that clearly captures the purpose and utility of the package. A package name may consist of a meaningful abbreviation. Examples of acceptable abbreviations are the standard Java packages of `java.io` and `java.net`.

17. Use the same name for a new version of a package, but only if that new version is still binary compatible *with the previous version, otherwise, use a new name.*

The intent of this rule is to ensure that two Java classes with identical qualified names will be binary and behaviorally compatible with each other.

The Java execution model binds the clients of a class to implementation of that class at run time. This means unless you adopt this convention, you have no way to ensure that your application is using the same version of the software you had used and tested with when you built the application.

If you produce a new version of a package that is not binary or behaviorally compatible, you should change the name of the package. This renaming may be accomplished in a variety of ways, but the safest and easiest technique is simply to

add a version number to the package name and then increment that version number each time an incompatible change is made:

```
com.roguewave.server.v1
com.roguewave.server.v2
:
```

The one drawback to this approach is the dependency between a client of a package and a specific implementation of that package is hard-coded into the client code. A package client can only be bound to a new version of that package by modifying the client code.

Type Names

18. Capitalize the first letter of each word that appears in a class or interface name.

The capitalization provides a visual cue for separating the individual words within each name. The leading capital letter provides a mechanism for differentiating between class or interface names and variable names (see Rule #25):

```
public class PrintStream
  extends FilterOutputStream {
  ...
}
public interface ActionListener
  extends EventListener {
  ...
}
```

Class Names.

19. Use nouns when naming classes.

Classes define objects, or *things*, which are identified by nouns:


```
class CustomerAccount {
  ...
}
public abstract class KeyAdapter
  implements KeyListener {
  ...
}
```

20. Pluralize the names of classes that group related attributes, static services, or constants.

Give classes that group related attributes, static services, or constants a name that corresponds to the plural form of the attribute, service, or constant type defined by the class.

The `java.awt.font.LineMetrics` class is an example of a class that defines an object that manages a group of related attributes:

```
/**
 * The <code>LineMetrics</code> class gives
 * access to the metrics needed to layout
 * characters along a line and to layout of
 * a set of lines.
 */
public class LineMetrics {
  public LineMetrics()
  public abstract int getNumChars();
  public abstract float getAscent();
  public abstract float getDescent();
  public abstract float getLeading();
  public abstract float getHeight();
  ...
}
```

The `java.beans.Beans` class is an example of a class that defines a group of related static services:

```
/**
 * The <code>Beans</code> class provides some
 * general purpose beans control methods.
 */
```

```
public class Beans {
  public static Object instantiate(...) {...}
  public static Object getInstanceOf(...) {...}
  public static boolean isInstanceOf(...) {...}
  public static boolean isDesignTime() {...}
  public static boolean isGuiAvailable() {...}
  public static void setDesignTime(...) {...}
  public static void setGuiAvailable(...) {...}
  ...
}
```

The `java.sql.Types` class is an example of a class that defines a group of related static constants:

```
/**
 * The <code>Types</code> class defines constants
 * that are used to identify SQL types.
 */
public class Types {
  public final static int BIT = -7;
  public final static int TINYINT = -6;
  public final static int SMALLINT = 5;
  public final static int INTEGER = 4;
  public final static int BIGINT = -5;
  public final static int FLOAT = 6;
  public final static int REAL = 7;
  public final static int DOUBLE = 8;
  public final static int NUMERIC = 2;
  public final static int DECIMAL = 3;
  public final static int CHAR = 1;
  ...
}
```

Interface Names

21. Use nouns or adjectives when naming interfaces.

An *interface* provides a declaration of the services provided by an object, or it provides a description of the capabilities of an object.

Use nouns to name interfaces that act as service declarations:


```
public interface ActionListener {
  public void actionPerformed(ActionEvent e);
}
```

Use adjectives to name interfaces that act as descriptions of capabilities. Most interfaces that describe capabilities use an adjective created by tacking an "able" or "ible" suffix onto the end of a verb:

```
public interface Runnable {
  public void run();
}

public interface Accessible {
  public Context getContext();
}
```

Method Names

22. Use lowercase for the first word and capitalize only the first letter of each subsequent word that appears in a method name.

The capitalization provides a visual cue for separating the individual words within each name. The leading lowercase letter provides a mechanism for differentiating between a method invocation and a constructor invocation:

```
class MyImage extends Image {
  public MyImage() {
    ...
  }

  public void flush() {
    ...
  }

  public Image getScaledInstance() {
    ...
  }
}
```

23. Use verbs when naming methods.

Methods and operations commonly define *actions*, which are verbs.

```
class Account {
  private int balance;
  ...
  public void withdraw(int amount) {
    deposit(-1 * amount);
  }

  public void deposit(int amount) {
    this.balance += amount;
  }
}
```

24. Follow the JavaBeans™ conventions for naming property accessor methods.

The JavaBeans™ specification[7] establishes standard naming conventions[8] for methods that give access to the properties of a JavaBean implementation. You should apply these conventions when naming methods in any class, regardless of whether it implements a Bean.

A JavaBean exposes `boolean` properties using methods that begin with "`is`":

```
boolean isValid() {
  return this.isValid;
}
```

A JavaBean gives read access to other property types using methods that begin with "`get`":

```
String getName() {
  return this.name;
}
```

The accessor method for reading an indexed property takes an `int` index argument:


```
String getAlias(int index) {
  return this.aliases[index];
}
```

A JavaBean gives write access to `boolean` and other types of properties using methods that begin with "`set`":

```
void setValid(boolean isValid) {
  this.isValid = isValid;
}

void setName(String name) {
  this.name = name;
}
```

The accessor method for setting an indexed property takes an `int` index argument:

```
void setAlias(int index, String alias) {
  this.aliases[index] = alias;
}
```

The *Java Development Kit* strongly adheres to these conventions. The is/get/set notation is required for exposing the component properties of a Bean unless you define a `BeanInfo` class.[9,10]

Variable Names

25. Use lowercase for the first word and capitalize only the first letter of each subsequent word that appears in a variable name.

The capitalization provides a visual cue for separating the individual words within each name. The leading lowercase letter provides a mechanism for differentiating between variable names and class names (see Rule #18):

```
class Customer {
  ...
  private Address address;
```

```
  private Phone daytimePhone;
  ...
  public Address setAddress(Address address) {
    Address oldAddress = this.address;
    this.address = address;
    return oldAddress;
  }
  ...
  public void setDaytimePhone(Phone daytimePhone);
    ...
  }
  ...
}
```

26. *Use nouns to name variables.*

Variables refer to objects, or *things*, which are identified by nouns:

```
class Customer {
  ...
  private Address     billingAddress;
  private Address     shippingAddress;
  private Phone       daytimePhone;
  private Vector      openOrders;
  ...
}
```

27. *Pluralize the names of collection references.*

Give fields and variables that refer to collections of objects a name that corresponds to the plural form of the object type contained in the collection. This enables a reader of your code to distinguish between variables representing multiple values from those representing single values:

```
Customer[] customers =
  newCustomer[MAX_CUSTOMERS];

void addCustomer(int index, Customer customer) {
  this.customers[index] = customer;
}
```



```
Vector orderItems = new Vector();

void addOrderItem(OrderItem orderItem) {
  this.orderItems.addElement(orderItem);
}
```

28. *Establish and use a set of standard names for trivial "throwaway" variables.*

You should use full descriptive names for most variables, but many variable types that appear frequently within Java code have common "shorthand" names, which you may choose to use instead.[11,12] The following table lists a few examples:

Character	c, d, e
Coordinate	x, y, z
Exception	e
Graphics	g
Object	o
Stream	in, out, inOut
String	s

Field Names

29. *Qualify field variables with "`this`" to distinguish them from local variables.*

To make distinguishing between local variables and field variables easier, always qualify field variables using "`this`":

```
public class AtomicAdder {

  private int count;

  public AtomicAdder(int count) {
    this.count = count;
  }
```

```
  public synchronized int fetchAndAdd(int value) {
    int temp = this.count;
    this.count += value;
    return temp;
  }

  public synchronized int addAndFetch(int value) {
    return this.count += value;
  }
}
```

Parameter Names

30. When a constructor or "set" method assigns a parameter to a field, give that parameter the same name as the field.

While hiding the names of instance variables with local variables is generally poor style, in this case some benefits exist. Using the same name relieves you of the responsibility for coming up with a name that *is* different. Using the same name also provides a subtle clue to the reader that the parameter value is destined for assignment to the field of the same name.

```
class Dude {

  private String name;

  public Dude(String name) {
    this.name = name;
  }

  public setName(String name) {
    this.name = name;
  }
}
```


Constant Names

31. Use uppercase letters for each word and separate each pair of words with an underscore when naming constants.

The capitalization of constant names distinguishes them from other nonfinal variables:

```
class Byte {
  public static final byte MAX_VALUE = 255;
  public static final byte MIN_VALUE = 0;
  public static final Class TYPE = Byte.class;
}
```

---

3 Sun Microsystems. *Java™ Code Conventions.* (Palo Alto, California: Sun Microsystems Inc., 20 April 1999). Accessed online at ftp://ftp.javasoft.com/docs/codeconv/CodeConventions.pdf, Aug 1999.

4 Sun Microsystems. *Java™ 2 Platform, Standard Edition, v1.2.2 API Specification.* (Sun Microsystems Inc., 1999). Accessed online at http://java.sun.com/products/jdk/1.2/docs/api/index.html, Aug 1999.

5 Jonathan Nagler. "Coding Style and Good Computing Practices." *The Political Methodologist,* Vol. 6, No. 2 (Spring 1995). Accessed online at http://wizard.ucr.edu/~nagler/coding_style.html, Aug 1999.

6 James Gosling et al. *The Java™ Language Specification.* (Reading, Massachusetts: Addison–Wesley, 1996), pp. 125–126.

7 To be called "Java Beans" or "Beans" for the remainder of this book.

8 Sun Microsystems. *JavaBeans™ API Specification.*, ed. Graham Hamilton, (Mountain View, California: Sun Microsystems Inc., 1997), pp. 54–57. Accessed online at http://www.javasoft.com/beans/docs/beans.101.pdf, Aug 1999.

9 Ibid., pp. 56–57.

10 Patrick Chan, and Rosanna Lee. *The Java™ Class Libraries, Volume 2: java.applet, java.awt, java.beans., 2nd Edition.* (Reading, Massachusetts: Addison–Wesley, 1998), pg. 132.

11 James Gosling et al. *The Java™ Language Specification.*

12 Sun Microsystems. *Java™ Code Conventions.*