The Java coding standard

Version 1.0, August 2002
S-JCS-2002-08-1-0

Contents

General principles

While it is important to write software than 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.

Adhere to the style of the original

When modifying existing software, your changes should follow the style of the original code. 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.

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 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 it.
Completeness	Provide the minimum functionality that any reasonable user will 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.

Do it right the first time

Apply these rules to any code you write, not just the 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.

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 shy you have done so.

Formatting conventions

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 generally code using a Java development environment, adjust the environment to produce correct indentation.

When writing the code, use four spaces to ensure readability:

```
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 a label easier to notice:

```
void function(int arg)
{
····loop:
········for(int index=0;index<=arg;index++)
············switch(index)
············{
··············case 0:
··················//
··················break loop;
··············default:
··················//
··················break;
············}
····}
}
```

Place the opening brace '{' of each block statement on a line of its own, aligned with the beginning of the statement preceding the block. Place the closing brace '}' of the block on a line of its own, aligned with the opening brace of the same 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 MyClass
{
```

```
...
    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 than everyone complies with this standard.

Our recommendation of four spaces is specific to Cybernetic Intelligence GmbH, although other organizations may use three or even two spaces.

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 a result is a line that exceeds your maximum allowable line length. If two statement expressions are placed on one line:

```
double s=Math.random(); double y=Math.ran
dom();  //  Too long!
```

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

```
double s=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(Ma
th.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 the 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 long line contains one or more commas:

```
double length=Math.sqrt(Math.pow(x,2.0),Math.p
ow(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 preceding 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==((MyClass)obj).field); //  Too long!
    }
      ...
}
```

Then introduce a line break just after 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 ||
                    (obj instanceof MyClass &&
                     this.field==((MyClass)obj).field);
    }
      ...
}
```

Step three

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

Declare class interfaces on a separate line

When a class implements one or more interfaces, place the declaration of implemented interfaces on a separate line and indent it:

```
class MyClass extends MyBaseClass
    implements MyInterface1,MyInterface2
{
     ...
}
```

Declare method exceptions on a separate line

When a method throws one or more exceptions, place the declaration of thrown exceptions on a separate line and indent it:

```
public void method()
    throws IOException
{
     ...
}
```

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 and code blocks clearly.

Use blank lines to separate:

- Each logical section of a method implementation:

```
void handleMessage(Message message)
{
    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
{
    class InnerFoo
    {
    }

    private Bar bar;
    Foo(Bar bar)
    {
        this.bar=bar;
```

```
    }
}
```

- Each class and interface definition in a source file:

```
package com.company.xyz;
interface FooInterface
{
  ...
}

public class Foo implements FooInterface
{
  ...
}
```

Do not use “hard” tabs

Many developers use tab characters to indent and align their source code, without realizing that 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.

Naming conventions

The naming conventions described in this section are based on those used by Sun Microsystems in naming the identifiers that appear in the Java Software Development Kit, with several amendments introduced by Cybernetic Intelligence GmbH as a result of its internal experience with Java programming.

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 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.

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.

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, than 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 an entity may produce new classes, interfaces, methods or variables that are more focused and can be given more meaningful yet simpler names.

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 appndSgntre(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.

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:

  ```
  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:

  ```
  private Document xmlDocument;
  ```

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 names **`theSqlInputStream`**. If both names appear in the same scope, each effectively hides the other when considered from a perspective of a person trying to read and understand the code.

Package names

Use the reversed, lowercase form of your organization's Internet domain name as the root qualified for names of your organization's private packages

Any package specific to your organization should include the lowercase domain name of the originating organization, in reverse order. For example, if a company named Rogue Wave Software, whose Internet domain name is `roguewave.com`, decides to develop and distribute an application server package called `server`, than Rogue Wave would name this 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.

Cybernetic Intelligence GmbH reserves the top-level package qualifier `gem` to be used by GEM core packages.

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

The qualified portion of a public package name should consist of a single, lowercase word that clearly captures the purpose and utility of a 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`.

If a package is intended for a private use by its developer, and not for the public use, prefix the package name with a single underscore.

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 behaviourally 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 a 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 this 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

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.

```
public class PrintStream extends FilterOutputStream
{
  ...
}

public interface ActionListener extends EventListener
{
  ...
}
```

Prefix the names of non-public classes or interfaces with an underscore

When naming a class or interface with a package (i.e. not public) visibility, prefix the class or interface name with an underscore:

```
class _QueueItem
{
  ...
}

public class Queue
{
  ...
}
```

Class names

Use nouns when naming classes

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

```
public class CustomerAccount
{
  ...
}

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

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:

```
public class LineMetrics
{
    public lineMetrics();
    public sbatract int getNumChars();
    public sbatract flost getAscent();
    public sbatract flost getDescent();
    public sbatract flost getLeading();
    public sbatract flost getHeight();
    ...
}
```

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

```
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:

```
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

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 the verb:

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

Prefix names of "true" interfaces with 'I'

An interface may appear in Java code in two situations:

1. It implements an interface from an object-oriented design model.
2. It implements a class from an object-oriented design model, which had to become Java interface to the Java language restrictions (for example, to emulate multiple inheritance).

In the situation 1, prefix the Java interface name with an uppercase letter 'I':

```
public interface ISerializable
{
    public void serialRead(IObjectInput stream);
    public void serialWrite(IObjectOutput stream);
}
```

This allows distinguishing between class and interface names more easily.

For a non-public interface, an interface name is prefixed with 'I' before it is prefixed with an underscore:

```
interface _IPrivateInterface
{
  ...
}
```

Method names

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

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

    public void flush()
    {
      ...
    }

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

```
}
```

Prefix the names of private methods with an underscore

When naming a package or private method, prefix the method name with an underscore:

```
public class MyImage extends Image
{
    public MyImage()
    {
        _initialize();
    }

    private void _initialize()
    {
      ...
    }
}
```

Use verbs with naming methods

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

```
public class Account
{
    private int m_balance;

    public void withdraw(int amount)
    {
        deposit(-1*anmount);
    }

    public void deposit(int amount)
    {
        m_balance+=amount;
    }
}
```

Follow the JavaBeans conventions for naming property accessor methods

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

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

```
public boolean isValid()
{
    return m_isValid;
}
```

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

```
public String getName()
{
    return m_name;
}
```

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

```
public String getAlias(int index)
```

```
{
    return m_aliases[index];
}
```

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

```
public void setValid(boolean isValid)
{
    m_isValid=isValid;
}

public void setName(String name)
{
    m_name=name;
}
```

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

```
public void setAlias(int index,String alias)
{
    m_aliases[index]=alias;
}
```

The Java Development Kit strongly adheres to these conventions. The is/get/set notation is requires for exposing the component properties of a Bean you define to a `BeanInfo` class.

Variable names

Use lowercase for the first word and capitalize only the first letter of every 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.

```
public class Customer
{
    public Address address;

    public Address setAddress(Address address)
    {
        Address oldAddress=this.address;
        this.address=address;
        return oldAddress;
    }
}
```

Use nouns to name variables

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

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

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=new Customer[MAX_CUSTOMERS];

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

Vector orderItems=new Vector();

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

Establish and use a set of standard names for “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. 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

You may qualify field variables with “this” to distinguish them from local variables

To make distinguishing between local variables and field variables easier, you may qualify field variables using “this”:

```
public class AtomicAdder
{
    public 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;
    }
}
```

Prefix the names of private fields with “m_”

When naming a private field, prefix the method name with “m_”:

```
public class Customer
{
    private Address m_billingAddress;
    private Address m_shippingAddress;
    private Phone m_daytimePhone;
    private Vector m_openOrders;
}
```

Prefix the names of package fields with an underscore

When naming a package field, prefix the method name with an underscore:

```
public class Customer
{
    Address _billingAddress;
    Address _shippingAddress;
    Phone _daytimePhone;
    Vector _openOrders;
}
```

Parameter names

When a constructor or “set” method assigns a parameter to a field, give that parameter the same name as a 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 from 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
{
    public String name;
    private String m_alias;

    puyblic Dude(String name,String alias)
    {
        this.name=name;
        this.m_alias=alias;
    }
}
```

Note, that if a parameter is destined for assignment to a private or package field, then, due to decoration of field names, the parameter name will actually not hide the field name.

Constant names

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 non-final variables:

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

Documentation conventions

Write documentation for those who must use your code and those who must maintain it

Document the public programming interface of your code so others can use it correctly and effectively. Document the private interface and internal implementation details of your code so others can maintain and enhance it.

Always assume someone who is completely unfamiliar with your code will eventually have to read and understand it. In fact, if enough time passes, your own code may become unfamiliar, so this person may even be you!

Keep comments and code in sync

> When the code and the comments disagree, both are probably wrong.
> – *Norm Schryer, Bell Labs.*

When you modify code, make sure you also update any related comments. The code and documentation together form a software product, so treat each with equal importance.

Use the active voice and omit needless words

Comments are a form of prose. Forceful, clear and concise language is especially beneficial for technical documentation. Use it.

Comment types

Java supports three comment types:

- A *documentation comment* that starts with "`/**`" and ends with "`*/`":

```
/**
 *   A documentation comment
 */
```

- A *standard* or *C-style comment* that starts with "`/*`" and ends with "`*/`":

```
/*
 *   A standard comment
 */
```

- A *one-line* or *end-line comment* that begins with "`//`" and continues to the end of the line:

```
//  A one-line comment

class MyClass
{
    int myField;  //  And end-line comment
    ...
}
```

Each comment type serves a particular purpose and you should use each type in a manner consistent with that purpose.

Use documentation comments to describe the programming interface

You may place documentation comments in front of any class, method, constructor or field declaration that appears in your code. These comments provide information the Javadoc utility uses to generate

HTML-formatted class reference or Application Programming Interface (API) documentation. To create this documentation, the Javadoc utility reads these comments as it parses all the declarations that appear in a set of Java source code files. This information is used by Javadoc to produce a corresponding set of HTML pages that describe, by default, all the public and protected classes, interfaces, constructors, methods and fields found in those files.

Javadoc only recognizes documentation comments when they appear immediately before a class, interface, constructor, method or field declaration. Javadoc ignores any documentation comments that appear within the body of a method, so do not use them in that manner. Javadoc allows only one documentation comment per declaration statement, so do not attempt to use more than one comment block per declaration.

The primary purpose for documentation comments is to define a *programming contract* between a *client* and a supplier of the *service*. The documentation associated with the method should describe all aspects of behavior on which a caller of that method can rely and should not attempt to describe implementation details.

The following code illustrates the use of documentation comments to document a class that declared an inner-class, a field, a method and a constructor:

```
/**
 * The <code>Rectangle2D</code> class describes a rectangle
 * defined by location (x,y) and dimensions (w,h).
 */
public abstract class Rectangle2D extends RectangularShape
{

    /**
     * The <code>Double</code> class defines a rectangle
     * specified in double coordinates.
     */
    static class Double extends Rectangle2D
    {
      ...
    }

    /**
     * The bitmask that indicates that a point lies
     * below this Rectangle2D
     */
    static int OUT_BOTTOM;

    /**
     * Adds a Rectangle2D to this Rectangle2D
     */
    public void add(Rectangle2D r)
    {
      ...
    }

    /**
     * This is an abstract class that cannot be
     * instantiated directly
     */
    protected Rectangle2D()
    {
      ...
    }

    ...
}
```

Use standard comments to hide code without removing it

Use standard C-style comments when you wish to hide code temporarily from the compiler without actually removing it from the source file. You may only use this comment type to "comment-out" a section of code that does not have another comment block embedded within it.

To avoid any problems with nested comment blocks, because they look very much like documentation comments, you should not use this type of comment for any purpose other than for temporarily hiding code.

The following code fragment demonstrates how to use this comment type to hide a member function definition:

```
/**
 * ...
 * @deprecated
 */
/*
  I have temporarily removed this method because
  It has been deprecated for some time, and I
  Want to determine whether any other packages
  Are still using it. – J. Kirk on 9 Dec 1997

public void thisOldFunction()
{
    //  There has got to be a better way!
    ...
}
*/
```

Use one-line comments to explain implementation details

Use one or more one-line comments to document:

- The purpose of specific variables or expressions.
- Implementation-level design decisions.
- The source material for complex algorithms.
- Defect fixes or workarounds.
- Code that may benefit from further optimization or elaboration.
- Known problems, limitations, or deficiencies.

Strive to minimize the need for embedded comments by writing code that documents itself. Do not add comments that simply repeat what code does. Add comments only if they add useful information.

```
double totalCost;  //  Used to total invoice
...

//  Apply the discount to all invoices over $1000
if(totalCost>1000.0)
{   //  TODO: Use constant?
    //  The discount is hard-coded because current
    //  customers all use the same discount rate.
    //  We will need to replace this constant with a
    //  variable if we ever get a customer who needs
    //  a different rate, or one that wants to apply
    //  multiple discount rates.

    totalCost*=DISCOUNT;
}
```

Documentation Comments

Describe the programming interface before you write the code

The best time to create API reference documentation is early in the development process. Use documentation or "doc" comments to define the purpose, use, and behavior of each class or interface that forms part of a potential design solution. Write these comments while the purpose and rationale for introducing the new type is still fresh in your mind. Do not think you must wait to complete the implementation of every method before generating documentation - the Javadoc utility can run on Java source files containing classes whose methods are simple stubs with no method bodies. This means you can write documentation comments and run Javadoc in the earliest stages of implementation, before writing any method bodies.

The initial description of a type and its methods and fields should not only provide guidance to the developers who must implement that type, but also form the basis for the final API reference documentation for that type. A developer tasked with implementing a class may choose to elaborate on the original documentation when the implementation details of the public interface become better defined and more apparent.

Document at least public and protected members

Supply documentation comments for all members with public and protected access. This allows for the generation of detailed API documentation. You can also choose to supply documentation comments for all or some members with package or private access, thus providing quality documentation to the developer who must learn and understand your code before implementing an enhancement or bug fix.

Have in mind that a detailed UML design model provides for much better means of understanding the intricacies of code that Javadoc documentation.

Provide a summary description and overview for each package

The Javadoc utility provides a mechanism for including package descriptions in the documentation it generates. Use this capability to provide a summary description and overview for each package you create.

To create a description for a package, you must create a package comment file, named `package.html`, and place that file in the package directory along with the other package source files. Javadoc will automatically look for a filename in this location.

The package comment file contains HTML, not Java source code. The package description must appear within an HTML `<body>` element. Javadoc treats the first sentence or phrase that appears within the `<body>...</body>` tags as the summary description for the package, just as it does when processing normal documentation comments.

You may use any Javadoc tag within the package description, except for the `{@link}` tag. Any `@see` tags that appear in a package description must use fully qualified names.

Provide a summary description and overview for each application or group of packages

The Javadoc utility provides a mechanism for including a package-independent overview description in the documentation it generates. Use this capability to provide an overview description for each application or group of related packages you create.

To create an overview description, you must create an overview comment file, that may be given any name that ends in "`.html`", such as `overview.html`. To include this file in your documentation, you must tell Javadoc where to find the file by using the `-overview` option.

The overview comment file contains HTML, not Java source code. The overview must appear within an HTML `<body>` element. Javadoc treats the first sentence or phrase that appears between the

<body>...</body> tags as the summary description for the application or package group, just as it does when processing normal documentation comments.

You may use any Javadoc tag within the description, except for the {@link} tag. Any @see tags that appear in an overview description must use fully qualified names.

Comment style

The “doc” comment formatting conventions listed in this section closely follow the conventions adopted and published by SUS MICROSYSTEMS.

Use a single consistent format and organization for all documentation comments

A properly formatted documentation comment contains a description followed by one or more Javadoc tags. Format each documentation comment as follows:

- Indent the first line of the comment to align the slash character of the start-comment symbol "/**" with the first character in the line containing the associated definition.
- Begin each subsequent line within an asterisk '*'. Align this asterisk with the first asterisk in the start-comment symbol.
- Use a single space to separate each asterisk from any descriptive text or tags that appear on the same line.
- Insert a blank comment line between the descriptive text and any Javadoc tags that appear in the comment block.
- End each documentation comment block with the asterisk in the end-comment symbol "*/" aligned with the other asterisks in the comment block:

```
/**
 * Descriptive text for this entity.
 * @tag Descriptive text for this tag.
 */
```

The following rules specify additional guidelines for creating high-quality, maintainable documentation comments.

Wrap keywords, identifiers, and constants with <code>...</code> tags

Nest keywords, package names, class names, interface names, method names, field names, parameter names, constant names, and constant values that appear in a documentation comment within HTML <code>...</code> mark-up tags:

```
/**
 * Allocates a <code>l1aq</code> object
 * representing the <code>value</code> argument.
 * ...
 */
public Flag(boolean value) {...}
```

The <code>...</code> tags tell HTML browsers to render the content in a different style than that of normal text, so these elements will stand out.

Wrap code with <pre>...</pre> tags

Nest code that appears in a documentation comment within
HTML <pre> ... < /pre> mark-up tags:

```
/**
 * The following example uses a
 * <code>Class</code> object to print the class
```

```
 * name of an object:
 *
 * <pre>
 * void printClassName(object o)
 * {
 *    System.out.println("The class of "
 *                       + o
 *                       + " is "
 *                       + o.getClass().getName());
 * }
 * </pre>
 */

public final class Class {...}
```

The <pre>...</pre> tags are used to tell HTML browsers to retain the original formatting, including indentation and line-ends, of the "preformatted" element.

Consider marking the first occurrence of an identifier with a {@link} tag

Each package, class, interface, method, and field name that appears within a documentation comment may be converted into a hypertext link by replacing that name with an appropriately coded {@link} tag:

```
/**
 * Allocates a <code>Flag</code> object
 * representing the <code>value</code> argument.
 * Use this form of constructor as an alternative
 * to the {@link #Flag(String)} form.
 */
public Flag(boolean value) {...}
/**
 * Allocates a <code>Flag</code> object
 * representing the value <code>true</code> if
 * the string argument is not <code>null</code>
 * and is equal to the string "true".
 * Use this form of constructor as an alternative
 * to the {@link #Flag(boolean)} form.
 */
public Flag(String s) {...}
```

Do not feel you must create links to every identifier that appears within a comment block. Creating a significant number of these links can quickly clutter a comment block with {@link} tags, making it hard to read and maintain the original source.

Create links only when the documentation associated with the referenced element would truly be of interest or value to the reader. Do not create links for every occurrence of an identifier. If an identifier appears more than once, simply create a link for the first occurrence and mark any subsequent instances using the <code>...</code> markup. Also, some classes and methods are so commonly used by a proficient Java programmer, they do not warrant their own links-again, these identifiers should still be marked using the <code>...</code> markup.

Establish and use a fixed ordering for Javadoc tags

SUN MICROSYSTEMS recommends the following Javadoc tag ordering:

- In classes and interface descriptions:

  ```
  /**
   * Description.
   *
  ```

```
 * @author
 * @version
 *
 * @see
 * @since
 * @deprecated
 */
```

Consider including an `@author` and `@version` tag in every class or interface description. List multiple @author tags in chronological order, with the class or interface creator listed first.

- In method descriptions:

```
/**
 * Description.
 *
 * @param
 * @return
 * @exception
 *
 * @see
 * @since
 * @deprecated
 */
```

Include a `@param` tag for every parameter. List multiple `@param` tags in parameter declaration order. Include a `@return` tag if the method returns any type other than void.

Include an @exception tag for every *checked exception* listed in a throws clause. Include an `@exception` tag for every *unchecked exception* that a user may reasonably expect to catch. List multiple `@exception` tags in alphabetical order of the exception class names.

- In field descriptions:

```
/**
 * Description.
 *
 * @see
 * @since
 * @deprecated
 */
```

Sort multiple `@see` tags according to their "distance" from the current location, in terms of document navigation and name qualification. Order each group of overloaded methods according to the number of parameters each accepts, starting with the method that has the least number of parameters:

```
/**
 * ...
 * @see #field
 * @see #Constructor()
 * @see #Constructor{Type...)
 * @see #method()
 * @see #method(Type...)
 * @see Class
 * @see Class#field
 * @see Class#Constructor()
 * @see Class#Constructor(Type...)
 * @see Class#method()
```

```
* @see Class#method(Type...)
* @see package
* @see package.Class
* @see package.Class#field
* @see package.Class#Constructor()
* @see package.Class#Constructor(Type...}
* @see package.Class#method(}
* @see package.Class#method(Type...)
* @see <a href="URL#label">label</a>
* @see "String"
* ...
*/
```

Write in the third person narrative form

When describing the purpose and behavior of classes, interfaces, and methods, use third-person pronouns - such as "they" and "it" - and third-person verb forms - such as "sets" and "gets" - instead of second-person forms-such as "set" and "get."

Some of the third-person verb forms that commonly appear in API documentation include

adds	**deallocates**	**removes**
allocates	**destroys**	**returns**
computes	**gets**	**sets**
constructs	**provides**	**tests**
converts	**reads**	**writes**

Write summary descriptions that stand alone

The Javadoc utility uses the first sentence or phrase in a documentation comment as a summary description of the class, interface, method, or field that immediately follows the comment block. To locate the end of the summary description, Javadoc starts at the beginning of the comment block and searches for a period that is followed by a space, tab, or end-of-line, or a Javadoc tag, whichever comes first.

Because this text provides a summary description for some entity, it must present a clear, simple, and concise description of that entity. Do not rely on other sentences in the comment block to provide additional context or elaboration.

Consider the following example:

```
/**
 * Use this function sparingly!
 * Applies the Foo-Bar algorithm to this node.
 */
 public void doFooBar() {...}
```

Processing this code with Javadoc produces the following summary description for the doFooBar method:

> Use this function sparingly!

Reordering the comment block produces a superior summary sentence:

```
/**
 * Applies the Foo-Bar algorithm to this nods.
 * Use this function sparingly!
 */
public void doFooBar() {...}
```

If the entity is an overloaded method, the summary description must differentiate that method from the other forms of the same method:

```
/**
 * Allocates a <code>Flag</code> object
 * representing the <code>value</code> argument.
 * ...
 */
public Flag(boolean value) {...}

/**
 * Allocates a <code>Flag</code> object
 * representing the value true if the
 * string argument is not null and is equal
 * to the string "true".
 * ...
 */
public Flag(String s) {...}
```

Omit the subject in summary descriptions of actions or services

A summary description does not require a subject because the subject can be determined from the context in which the description appears.

The following descriptions incorrectly provide a redundant identification of the subject:

```
/**
 * This method applies the Foo-Bar
 * algorithm to this node.
 * ...
 */
public void doFooBar{} {...}

/**
 * The <code>doFooBar</code> method applies the
 * Foo-Bar algorithm to this node.
 * ...
 */
public void doFooBar{} {...}
```

The following description correctly omits the subject:

```
/**
 * Applies the Foo-Bar algorithm to this nods.
 * ...
 */
public void doFooBar{} {...}
```

Omit the subject and the verb in summary descriptions of things

A summary description for a class, interface, or field that represents a "thing" does not require an explicit subject or verb, as the description needs only to identify an object. A subject is unnecessary because it can be determined from the context. A verb is unnecessary because it simply states the subject "is," "exists as," or "represents" some object.

The following example illustrates a description that contains an unnecessary subject and verb:

```
/**
 * A thread group represents a set of threads.
 * ...
 */
```

```
public class ThreadGroup {...}
```

Drop the subject and verb to obtain the correct form of summary description:

```
/**
 * A set of threads.
 * ...
 */
public class ThreadGroup {...}
```

Use "this" rather than "the" when referring to instances of the current class

When describing the purpose or behavior of a method, use "this" instead of "the" to refer to an object that is an instance of the class defining the method:

```
/**
 * Returns a <code>String</code> representing the
 * value of the <code>Flag</code> object.
 * ...
 */
public String toString() {...}

/**
 * Returns a <code>String</code> representing the
 * value of this <code>Flag</code> object.
 * ...
 */
public String toString() {...}
```

Do not add parentheses to a method or constructor name unless you want to specify a particular signature

A method or constructor reference should not include any parentheses unless the reference identifies an overloaded method or constructor and you wish to refer to a single form of the overloaded operation.

Do not add an empty pair of parenthesis "`()`" to indicate a name refers to a method. This practice causes confusion if the name is associated with an overloaded method and one of the overloaded forms of that method takes no arguments.

Consider the following pair of overloaded defined by the `java.lang.String` class:

```
public class String
{
 ...
 public String toLowerCase() {...}
 public String toLowerCase(Locale locale) {...}
 ...
}
```

If you use the identifier "`toLowerCase()`" to refer to any or all of the `toLowerCase` methods, then you will likely confuse those who read your documentation. Most likely, your users will think you meant the first form of the method, not either or all forms of it. Use parentheses only when you want to specify the exact signature of a method or constructor:

`toLowerCase`	Refers to either or both forms of the method.
`toLowerCase()`	Refers only to the first form of the method.
`toLowerCase(Locale)`	Refers only to the second form of the method.

Comment Content

Provide a summary description for each class, interface, fell and method

Every class, interface, field, and method should be preceded by a documentation comment that contains at least one sentence that acts as a summary description of that entity.

Fully describe the signature of each method

The documentation for each method should always include a description for each parameter, each checked exception, any relevant unchecked exceptions, and any return value.

Include examples

One of the easiest ways to explain and understand how to use software is by giving specific examples.

Try to include a simple example in each nontrivial class and method description. Use the HTML `<pre>...</pre>` tags to maintain the formatting of each example:

```
/**
 * If you are formatting multiple numbers, it is more
 * efficient to get the format just once so the system
 * does not have to fetch the information about the
 * local language and country conventions multiple
 * times:
 * <pre>
 * DateFormat df=Daterormat.getDateInstance();
 * for(int i=0;i<a.length;++i)
 * {
 *     output.println(df.format(myDate[i])+";");
 * }
 * </pre>
 * To format a number for a different Locale,
 * specify the locale in the call to
 * <code>getDatelnstance</code>:
 * <pre>
 * DateFormat df;
 * df=DateFormat.getDateInstance(Locale.US);
 * </pre>
 * ...
 */
public abstract class DateFormat extends Format
```

Document preconditions, postconditions, and invariant conditions

A *precondition* is a condition that must hold true before a method starts if this method is to behave properly. A typical precondition may limit the range of acceptable values for a method argument.

A *postcondition* is a condition that must hold true following the completion of a method if this method has behaved properly. A typical postcondition describes the state of an object that should result from an invocation of the method given an initial state and the invocation parameters.

An *invariant* is a condition that must always hold true for an object. A typical invariant might restrict the integer field `vacationDays` to a value between `0` and `25`.

As preconditions, postconditions, and invariants are the assumptions under which you use and interact with a class, documenting them is important, especially if these conditions are too costly to verify using run-time assertions.

Document known defects and deficiencies

Identify and describe any outstanding problems associated with a class or method. Indicate any replacements or workarounds that exist. If possible, indicate when the problem might be resolved.

While no one likes to publicize problems in his or her code, your colleagues and customers will appreciate the information. This information will give them the chance to implement a workaround or to isolate the problem to minimize the impact of future changes.

Document synchronization semantics

The presence of the synchronized modifier in the signature of a method normally reveals whether that method serializes calling threads to protect the state of an object. A user can determine whether a method is synchronized by looking at the documentation generated by Javadoc because Javadoc will include the modifier as part of the signature of each method that is declared as synchronized.

Java also provides a second synchronization mechanism that applies to a block of code instead of an entire method. Methods that use this second mechanism may, in fact, be thread-safe, but the signature of these methods will not indicate this. In this situation, you must indicate these are internally synchronized methods within the documentation for each such method.

Internal Comments

Add internal comments only if they will aid others in understanding your code

Avoid the temptation to insert comments that provide useless or irrelevant information:

```
public int occurrences0f(Object item)
{
    // This turned out to be much simpler
    // than I expected. Let's Go Mets!!
    return (find(item)!=null)?1:0;
}
```

Add comments only when they provide information that will help others understand how the code works:

```
public int occurrences0f(Object item)
{
    // This works because no duplicates are allowed:
    return (find(item)!=null)?1:0;
}
```

If an internal comment does not add any value, it is best to let the code speak far itself.

Describe why the code is doing what it does, not what the code is doing

Good code is self-documenting. Another developer should be able to look at well-written code and determine what it does.

For example, a quick examination of the following code reveals that the program appears to give a 5 percent discount when an invoice totals over a thousand dollars:

```
if(this.invoiceTotal>1000.0)
    this.invoiceTotal=this.invoiceTotal*0.95;
```

The following comment provides little additional information:

```
// Apply a 5% discount to all invoices
// over a thousand dollars:
```

```
if(this.invoiceTotal>1000.0)
    this.invoiceTotal=this.invoiceTotal*0.95;
```

After reading this code, a reasonable developer may still want to know:

- Why is the discount 5 percent?
- Who determined the discount and dollar amount?
- When or why would these amounts change?

Identify and explain any domain-specific knowledge that is required to understand the code:

```
// This term corrects for the effects of Jupiter,
// Venus, and the flattening of the earth:
sigma+=(cl*Angle.sin(al)
       +c2*Angle.sin(Angle.minus(L1,F))
       +c3*Angle.sin(a2));
```

Avoid the use of end-line comments

End-line comments, one-line comments appended to a line of working code, should be used with care. They can easily interfere with the visual structure of code. Modifications to a commented line of code may push the comment far enough to the right that line wrapping or horizontal scrolling must be employed before the comment can be seen within an editor. Some programmers try to improve the appearance of end-line comments by aligning them so they are left justified, only to find themselves constantly realigning the comments each time the code is modified. This is a waste of time.

Place one-line comments on a separate line immediately preceding the code to which they refer. The only exception to this rule involves local variable declarations whose descriptions are short enough that an end-line comment can describe them without producing an unacceptably long line of code.

Explain local variable declarations with an end-line comment

If the description of a local variable is quite short, consider placing the description in a comment on the same line as the declaration:

```
int cur = 0; // Index of current pattern element
int prev = 0; // Index of previous pattern element
```

Do not worry about positioning the comment so it appears aligned with other end-line comments.

Establish and use a set of keywords to flag unresolved issues

Establish a set of keywords for use in creating special comments that you and other developers can use to signal unresolved issues, which must eventually be dealt with before the code is considered complete. These comments should include a date and the initials of the person who raised the issue. The keywords should be chosen to minimize the chances that the same text may appear elsewhere within the code. In the following example, a pair of colons decorates the word "UNRESOLVED" to increase the likelihood that it is unique:

```
// :UNRESOLVED: EBF, 11 July 1999
// This still does not handle the case where
// the input overflows the internal buffer!!
while(everMorelnput) ...
```

Label closing braces in highly nested control structures

While you should generally avoid creating deeply nested control structures, you can improve the readability of such code by adding end-line comments to the closing braces of each structure:

```
for (i...)
{
    for (j...)
```

```
    {
        while(...)
        {
            if(...)
            {
                switch(...)
                {
                 ...
                } // end switch
            } // end if
        } // end while
    } // end for j
} // end for i
```

Add a “fall-through” comment between two ease labels, if no break statement separates those labels

When the code following a switch statement's case label does not include a break but, instead, "falls through" into the code associated with the next label, add a comment to indicate this was your intent. Other developers will either incorrectly assume a break occurs, or wonder whether you simply forgot to code one:

```
switch(command)
{
    case FAST-FORWARD:
        isFastForward=true;
        // Fall through
    case PLAY:
    case FORWARD:
        isForward=true;
        break;
    case FAST_REWIND:
        isFastRewind=true;
        // Fall through
    case REWIND:
        isRewind=true;
        break;
}
```

Note that two adjacent labels do not require an intervening comment.

Label empty statements

When a control structure, such as while or for loop, has an empty statement by design, add a comment to indicate this was your intent.

```
// Strip leading spaces
while((c=reader.read(1)==SPACE)
    ; // empty
```

Programming conventions

Consider declaring classes representing fundamental data types as final

Simple classes representing fundamental data types, such as a `complexNumber` class in an engineering package, find widespread use within their target domain. As such, efficiency can become an issue of some importance. Declaring a class as final allows its methods to be invoked more efficiently.

Of course, declaring your class as final will prohibit its use as a superclass. Nevertheless, there is seldom any reason to extend a class that implements a fundamental data type. In most such cases, object composition is a more appropriate mechanism for reuse.

Build concrete types from native types and other concrete types

Each nonnative, nonconcrete type that appears in the interface of a concrete type introduces a new, potentially volatile dependency, effectively exposing every client of that concrete type to volatility in these other types.

Minimize the number of dependencies that a concrete class has on nonnative, nonconcrete types. A concrete type that is defined purely in terms of native types provides better isolation and stability than a concrete type built from other concrete types. This is especially important for classes that implement fundamental data types, as dependencies on these low-level classes tend to proliferate throughout an application.

Consider the public interface of the java.util.Bitset class:

```
public final class Bitset ...
{
    public Bitset() {...}
    public BitSet(int) {...}
    public void set(int) {...}
    public void clear(int) {...}
    public boolean get(int) {...}
    public void and(SitSet) {...}
    public void or(BitSet) {...}
    public void xor(SitSet) {...}
    public int hashcode() {...}
    public int size() {...}
    public boolean equals(Object) {...}
    public Object clone() {...}
    public String tostring(} {...}
}
```

This class uses five data types in its interface: `Bitset`, which is a self-reference, the primitive types `int` and `boolean`, and the Java native types object and string. Because the `Bitset` interface uses stable, native Java types, little possibility exists that changes outside this class will affect this class or its clients.

Define small classes and small methods

Smaller classes and methods are easier to design, code, test, document, read, understand, and use. Because smaller classes generally have fewer methods and represent simpler concepts, their interfaces tend to exhibit better cohesion.

Try to limit the interface of each class to the bare minimum number of methods required to provide the necessary functionality. Avoid the temptation to add "convenience" forms of a method when only one general-purpose form will suffice.

All sorts of informal guidelines exist for establishing the maximum size of a class or method---use your best judgment. If a class or method seems too big, then consider refactoring that class or method into additional classes or methods.

Define subclasses so they may be used anywhere their superclasses may be used

A subclass that changes or restricts the behavior of its ancestor class by overriding something is a specialization o£ that class, and its instances may have limited substitutability for the instances of its ancestor class. A specialization may not always be used anywhere the parent class could be used.

A subclass that is behaviorally compatible with its ancestor class is a subtype and its instances are fully substitutable for instances of its ancestor class. A subclass that implements a subtype does not override anything in its ancestor class; it only extends the services provided by that class. A subtype has the same attributes and associations as its supertype.

The following design principle addresses the question of substitutability:

> **The Liskov Substitution Principle**
> Methods that use references to base classes must be able to use objects of derived classes without knowing it.

According to this principle, the ability to substitute a derived class object for a superclass object is characteristic of good design. Such designs offer more stability and reliability when compared with designs that fail to uphold this principle. When a design adheres to this principle, it generally indicates the designer did a good job identifying the base abstractions and generalizing their interfaces.

Any design that requires code changes to handle the introduction of a newly derived class is a bad design. Whenever a derived class violates the existing contract between its superclasses and their clients, it forces changes in the existing code. When a method accepts a superclass instance, yet uses the derived type of this instance to control its behavior, changes will be required for the introduction of each new derived class. Changes of this kind violate the Open-Closed Principle and are something to avoid.

> **The Open-Closed Principle**
> Software entities (Classes, Modules, Functions, and so forth) should be open for extension, but closed for modification.

Consider the following example:

```
class Shape
{
    ...
    public Shape getNext{) { return this.next; }
    public int getDepth() { return this.depth; }
    ...
}

class Circle extends Shape {...}
class Rectangle extends Shape {...}
class Canvas
{
    public void drawShapes(ShapeList list)
    {
        Shape shape=list.getNextShape();
        // Use null to detect end of list
        while(shape!=null)
        {
            drawShape(shape);
            shape=1ist.getNextShape();
        }
```

```
    }

    public void drawShape(Shape shape)
    {
        // Use derived type to call relevant method
        if(shape instanceof Circle)
            drawCircle((Circle)shape);
        else if(shape instanceof Rectangle)
            drawRectangle((Rectangle)shape);
    }

    public void drawCircle(Circle circle) {...}
    public void drawRectangle(Rectangle circle} {...}

    class ShapeList
    {
        protected Shape first;
        protected int entries;
        public int getEntries() ( this.entries; }
        public Shape getNextShape()
        {
            Shape temp=this.first;
            if(null!=this.first)
            {
                this.first=temp.getNext();
                this.entries--;
            }
            // Returns null when empty
            return temp;
        }
    }
}
```

These classes make up a simple drawing package where `Shapes` are stored in `ShapeLists` and passed to a Canvas for rendering. The `drawShapes` method of a canvas object reads one `Shape` object at a time from a list and dispatches each to the appropriate draw method. The process continues until the `getNextShape` method returns a null value, indicating the fist has been fully traversed.

Now, suppose our design calls for a `DepthFilteredShapeList` class that can filter out any `Shapes` whose depth value falls outside a specified target range:

```
class DepthFilteredShapeList extends ShapeList
{
    protected int min;
    protected int max;
    public Shape getNextShape()
    {
        Shape temp = this.first;
        if(null!=this.first)
        {
            this.first=temp.next;
            this.entries--;
            int depth=temp.getDepth();
            // Is the shape in range?
            if(this.min>depth && depth>this.max)
            {
                // No - return null instead
                temp = null;
            }
        }
       // Returns null when filtered!
```

```
        return temp;
    }
}
```

This implementation has a problem, however. The `getNextShape` method not only returns a null value once the end of the list is reached, but also returns a null value for each shape that is filtered. The canvas class does not expect this behavior from a `ShapeList` and will incorrectly stop rendering when it tries to read a `Shape` subject to filtering.

This example violates the Liskov Substitution Principle. In this case, we can satisfy the principle by changing the `getNextShape` method so it will continue to traverse the shape list until it finds an unfiltered shape or until it reaches the end of the list.

The Liskov Substitution Principle also applies to methods. A method designed to recognize particular derivations of a superclass may not know how to handle a new derivation. The `drawShape` method in the canvas class illustrates this problem. This method interrogates each incoming shape to determine its type in to dispatch it to the appropriate drawing routine. Developers would have to change the `Canvas` class and the `drawShape` method each time they wanted to add a new subclass of the `Shape` class.

This problem is solved by adding a `drawSelf` method to the `Shape` subclasses and replacing the shape-specific methods on the canvas with a set of primitive drawing operations that `Shapes` can use to draw themselves. Each subclass of shape would override the `drawSelf` method to call the canvas drawing operations necessary to produce that particular shape. The `drawShapes` method on the canvas would no longer call `drawShape` to dispatch shapes to canvas routines, but would, instead, call the `drawSelf` method on each Shape subclass:

```
class Shape
{
    ...
    public abstract void drawSelf(Canvas canvas);
    ...
}

class Circle extends Shape
{
    ...
    public void drawSelf(Canvas canvas) {...}
    ...
}

class Canvas
{
    public void drawShapes(ShapeList list)
    {
        Shape shape=list.getNext8hape();
        // Use null to detect end of list
        while(shape!=null)
        {
            // Tell the shape to draw itself
            shape.drawself(this);
            shape=list.getNextShape();
        }
    }

    // Define the operations the shapes will use
    public void drawLine(int xl,int yl,int x2,int y2) {...}
    public void drawCircle(int x,int y,int radius} {...}
    ...
}
```

Make all fields private

> Out of sight, out of mind. - *Anonymous*

Doing so ensures the consistency of the member data because only the owning class may make changes to it. Access all member data through object methods. This minimizes coupling between objects, which enhances program maintainability.

The only possible exception from this rule can be made for fields that:

- Have no constraints on their values, and
- Are intended by frequent use

Use polymorphism instead of instanceof

Do not use instanceof to choose behavior depending upon an object's type. This forces you to modify the choice selection code every time the set of choice object types changes, leading to brittle code.

Instead, implement object-specific behavior in methods derived from a base class. This enables a client to interact with the base class abstraction without knowledge of the derived classes, allowing new classes to be introduced without the client's knowledge.

Type safety

Wrap general-purpose classes that operate on java.1ang.Object to provide static type checking

Ensure type safety by wrapping a general class that barters in object types with one that casts objects to a specific type. The following code shows how to wrap a general-purpose queue to create a type-specific one:

```
public class Queue
{
    public void enqueue(Object object) {...}
    public Object dequeue() {...}
}

public class OrderQueue
{
    private Queue m_queue;
    public OrderQueue()
    {
        m_queue = new Queue();
    }
    public void enqueue(Order order)
    {
        m_queue.enqueue(order):
    }
    public Order dequeue()
    {
        return (Order)m_queue.dequeue();
    }
}
```

Encapsulate enumerations as classes

Encapsulate enumerations as classes to provide type-safe comparisons of enumerator values:

```
public class Color
{
    private static int count=0;
```

```
    public static final Color RED=new Color(count++);
    public static final Color GREEN=new Color(count++);
    public static final Color BLUE=new Color(count++);
    private int value;

    private Color(int value)
    {
        this.value=value;
    }

    public boolean equals(Color other)
    {
        return this.value==other.value;
    }

    public static int Color.count()
    {
        return count;
    }
}

Color aColor=Color.RED;
if(anotherColor.equals(aColor))
{
    ...
}
```

Statements and Expressions

Replace repeated nontrivial expressions with equivalent methods

Write code once and only once. Factor out common functionality and repackage as a method or a class. This makes code easier to learn and understand. Changes are localized, so maintenance is easier and testing effort is reduced.

Use block statements instead of expression statements in control flow constructs

The Java block statement provides a mechanism for treating any number of statements as a single compound statement. A block statement may be used anywhere a regular statement may be used, including the expression statement bodies of a Java control construct.

While the language enables you to use simple, nonblock statements as the body of these constructs, you should always consider using a block statement in these situations.

Block statements reduce the ambiguity that often arises when control constructs are nested, and they provide a mechanism for organizing the code for improved readability.

The following code fragment is confusing because the indentation makes it appear as though the else clause is associated with the first `if` statement, while the compiler will associate it with the second `if` statement. The Java language specification refers to this as the "dangling else problem." The use of block statements eliminates this problem:

```
if(x>=0)
    if(x>0)
        positiveX();
else // Oops! Actually matches most recent if!
    negativeX();

if(x>=0)
```

```
{
    if(x>0)
        positivex();
}
else
{
    negativeX(); // This is what we really wanted!
}
```

In the following example, the code on the top is more difficult to modify than the code on the bottom. This is because you cannot add another statement without changing the existing code structure. Because the code on the bottom already uses block statements, modifications are easier to make:

```
for(int i=n;i>=0;i--)
    for (int j = n;j>=0;j--)
        f(i,j);
        // g(i,j) Cannot add here!

for(int i=n;i>=0;i--)
{
    for(int j=n;j>=0;j--)
    {
        f(i,j),
        g(i,j); // Can add here!
    }
}
```

If a control statement has a single, trivial statement as its body, you may put the entire statement on a single line, but only if it improves readability. Treat this case as the exception rather than the norm.

Clarify the order of operations with parentheses

The order of operations in a mathematical expression is not always obvious. Even if you are certain as to the order, you can safely assume others will not be so sure.

```
// Extraneous but useful parentheses.
int width=((buffer*offset)/pixelWidth)+gap;
```

Always code a break statement in the last case of a switch statement

The following switch statement was coded with the assumption that no other cases would follow the Y case, so no `break` statement was required:

```
switch(...)
{
    case X:
        ...
        break;
    case Y:
        ...
}
```

What if a new case is needed, however, and the person adding this case decides simply to add it after the last existing case, but fails to notice this case did not have a break statement? This person may inadvertently introduce a hard-to-detect "fall-through" error, as shown here:

```
switch(...)
{
    case X:
        ...
        break;
    case Y:
```

```
        ...
        // Oops! Unintended fall-through!
    case Z:
        ...
}
```

To prevent future problems, you should always code a break statement for the last case in the switch statement, even if it is the default case:

```
switch(...)
{
    case X:
        ...
        break;
    case Y:
        ...
        break; // OK! No more fall-through!
    case Z:
        ...
        break;
    default:
        ...
        // Complain about value!
        break;
}
```

Do not forget to add a "fall-through" comment in those casts that really do "fail-through."

Use equals(), not ==, to test for equality of objects.

Many C++ programmers make this mistake when dealing with Java dates and strings:

```
Date today=new Date();
while(date!=today) ...

String name;
...
if(name=="Bob")
    hiBob();
```

In Java, the "`!=`" and "`==`" operators compare object identities, not object values. You must use the `equals` method to compare the actual strings:

```
Date today=new Date();
while(!date.equals(today)) ...

String name;
if("Bob".equals(name))
    hiBob();
```

Note, unlike the expression `name.equals("Bob")`, the expression `"Bob".equals(name)` does not throw an exception if name is null.

Construction

Always construct objects in a valid state.

Never allow an invalid object to be constructed. If an object must be constructed in an invalid state and then must undergo further initialization before it becomes valid, use a static method that coordinates the multistage construction. The construction method should construct the object so that when the method

completes, the new object is in a state. Hide any constructors that do not construct instances by making them protected or private.

Do not call nonfinal methods from within a constructor

Subclasses may override nonfinal methods and Java will dispatch a call to such a method according to the actual type of the constructed object - before executing the derived class constructors. This means when the constructor invokes the derived method, the derived class may be in an invalid state. To prevent this, call only final methods from the constructor.

Use nested constructors to eliminate redundant code

To avoid writing redundant constructor code, call lower-level constructors from higher-level constructors.

This code implements the same low-level initialization in two different places:

```
class Account
{
    String _name;
    double _balance;
    final static double _DEFAULT_BALANCE=0.0d;

    Account(String name,double balance)
    {
        this._name=name;
        this._balance=balance;
    }

    Account(String name)
    {
        this._name=name;
        this._balance=_DEFAULT_BALANCE;
    }
    ...
}
```

This code implements the low-level initialization in one place only:

```
class Account
{
    String _name;
    double _balance;
    final static double _DEFAULT_BALANCE=0.0d;

    Account(String name,double balance)
    {
        this._name=name;
        this._balance=balance;
    }

    Account(String name)
    {
        this(_name,_DEFAULT_BALANCE);
    }
    ...
}
```

This approach is also helpful if you are using assertions, as it typically reduces the number of places a given constructor argument appears, thus reducing the number of places the validity of that argument is checked.

Exception Handling

Use unchecked run-time exceptions to report serious unexpected errors that may indicate an error in the program's logic

Catching and handling run-time exceptions is possible, however, they are usually of such a severe nature that program termination is imminent, Run-time exceptions are usually thrown because of programming errors, such as failed assertions, using an out-of-bound index, dividing by zero, or referencing a null pointer.

Use checked exceptions to report errors that may occur, however rarely, under normal program operation.

Checked exceptions indicate a serious problem that should not occur under normal conditions. The caller must catch this exception. Depending upon the application, a program may be able to recover from a checked exception; that is, it doesn't indicate a fundamental flaw in the program's logic.

Use return codes to report expected state changes

For expected state changes, use a return code, sentinel, or special method that returns a status. This makes code more readable and the flow of control straightforward. For example, in the course of reading from a file, it is expected the end of the file will be reached at some point.

Only convert exceptions to add information.

Retain all exception information; never discard lower-level explanations:

```
try
{
    for(int i=v.size();--i>= 0;)
    {
        ostream.println(v.elementAt(i));
    }
}
catch(ArrayOutOfBounds e)
{
    // should never get here
    throw new UnexpectedExceptionError(e);
}
```

Do not silently absorb a run-time or error exception

Breaking this rule makes code hard to debug because information is lost:

```
try
{
    for(int i=v.size(};--i>=0;)
    {
        ostream.println(v.elementAt(i));
    }
}
catch(ArrayOutOfBounds e)
{
    // Oops! We should never get here...
    // ... but if we do, nobody will ever know!
}
```

Even if you have coded a catch block simply to catch an exception you do not expect to occur, at least print a "stack trace." You never know when something "impossible" might occur within your software:

```
try
```

```
{
    for(int i=v.size(};--i>=0;)
    {
        ostream.println(v.elementAt(i));
    }
}
catch(ArrayOutOfBounds e)
{
    // Oops! We should never get here...
    // ... but print a stack trace just in case
    e.printStackTrace();
}
```

Use a finally block to release resources

Once a try-block is entered, any corresponding finally block is guaranteed to be executed. This makes the finally block a good place to release any resources acquired prior to entering or within the try-block. In this first example, if an exception or return occurs following the creation of the output stream, the function will exit without closing and flushing the stream:

```
public void logSomeStuff()
{
    OutputStream log=new FileOutputStream("log");
    // could get exception here!
    log.close();
}
```

In this example, we use a finally block to ensure the stream is always closed when the thread of execution exits the try-block. This is done even if the thread exits the block because an exception has been thrown or a return statement was executed:

```
OutputStream log=null;
try
{
    log=new FileOutputStream("log");
}
finally
{
    if(log!=null)
        log.clcse{);
}
```

Assertions

Program by contract

Consider each method a contract between the caller and the callee. The contract states the caller must abide by the "pre-conditions" of a method and the method, in turn, must return results that satisfy the "postconditions" associated with that method,

Abiding by the preconditions of a method usually means passing parameters as the method expects them; it may also mean calling a set of methods in the correct order. To abide by the postconditions of the method, the method must correctly complete the work it was called upon to perform and it must leave the object in a consistent state.

Check preconditions and postconditions by assertion in any appropriate public methods. Check preconditions at the beginning of a method, before any other code is executed, and check postconditions at the end of a method before the method returns.

Derived class methods that override base class methods must preserve the pre- and postconditions of the base class method. To ensure this, you may use the template method design pattern.

Make each public method final and create a parallel nonfinal protected method that implements the body of the function. The public final method tests preconditions, calls the associated protected method, and then tests postconditions. A deriving class may override public behavior in the base class by overriding the nonfinal protected method associated with each public final method:

```
class LinkedList
{
    public final synchronized void prepend(Object object)
    {
        // Test pre-condition
        if(Assert.ENABLED)
            Assert.isTrue(object!=null);
        doPrepend(object);
        // Test post-condition
        if(Assert.ENABLED)
            Assert.isTrue(first()==object);
    }
    protected void doPrepend(Object object)
    {
        Node node=new Node(object);
        if(this.head==null)
            this.head=node;
        else
        {
            node.next=this.head;
            this.head=node;
        }
    }
}
```

Use dead code elimination to implement assertions

An assertion is an expression you, the programmer, insist must hold true for a piece of code to operate correctly. Assertions are used in code to ensure basic coding assumptions are not violated. If an assertion evaluates to false, the code is flawed.

Use assertions liberally throughout code to rest the basic premises upon which the code was built. Assertions take time to execute, though, and we usually want to remove them from the released code. For this, we can take advantage of dead code elimination.

Dead code elimination occurs when a Java compiler eliminates unreachable code. For example, when a compiler sees the following piece of code, it knows the variable FALSE will always evaluate to false, as it is a static final variable. This allows the compiler to eliminate the block of code following the if expression, as the compiler knows it can never evaluate to true:

```
class DeadCode
{
    public static final boolean FALSE=false;
    public void example()
    {
        if(FALSE)
        {
            System.out.println(°Never to be seen,");
        }
    }
}
```

This technique also works to ensure that a method will remain synchronized, even when overridden. A derived class may violate the synchronization semantics of a base class by overriding a synchronized method with an unsynchronized version of the method - derived class methods do not inherit the

synchronized qualifier. A superclass can guarantee synchronization by providing a public final synchronized method that calls the nonfinal method.

Using what we know about dead code elimination, we can write an assertion class that will enable us to choose when to include assertions in the compiler-generated code:

```
public class Assert
{
    public static final boolean ENABLED=true;
    public static final void iSTrue(boolean assertion)
    {
        if(Assert.ENABLED && !assertion)
            throw new RuntimeException("Assertion Failed");
    }
}
...
if(Assert.ENABLED)
    Assert.isTrue(a>b);
...
```

To turn assertions off, set the ENABLED variable in the Assert class to false.

Failed assertions indicate an error in program logic, either in the use of a method or in the implementation of that method. For this reason, report assertion failures by throwing an unchecked exception such as `RuntimeException` or some derivation thereof.

Use assertions to catch logic errors in your code

An assertion is a `Boolean` expression that must hold true for a program to operate correctly. Use assertions to validate the assumptions made by a program.

Use assertions to test pre- and postconditions of a method

A method's *preconditions* are those conditions required for the method's proper execution. For example, a precondition may test the validity of the parameters passed to the method or test that the object is in a valid state.

Postcondition assertions execute at the completion of a method to verify the object is still in a valid state and the return values of the method are reasonable.

Concurrency

Concurrency exists when two or more threads make progress, executing instructions at the same time. A single processor system can support concurrency by switching execution between two or more threads. A multiprocessor system can support parallel concurrency by executing a separate thread on each processor. A class is multithread-hot or MT-hot if it creates additional threads to accomplish its task.

Many applications can benefit from the use of concurrency in their implementation. In a concurrent model of execution, an application is divided into two or more processes or threads, each executing in its own sequence of statements or instructions. An application may consist of one or more processes and a process may consist of one or more threads. Execution may be distributed on two or more machines in a network, two or more processors in a single machine, or interleaved on a single processor.

The separately executing processes or threads must generally compete for access to shared resources and data, and must cooperate to accomplish their overall task.

Concurrent application development is a complicated task. Designing a concurrent application involves determining the necessary number of processes or threads, their particular responsibilities, and the methods by which they will interact. It also involves determining the good, legal, or invariant program

states and the bad or illegal program states. The critical problem is to find and implement a solution that maintains or guarantees good program states while prohibiting bad program states, even in those situations where two or more threads may be acting on the same resource.

In a concurrent environment, a programmer maintains desirable program states by limiting or negotiating access to shared resources using synchronization. The principle role of synchronization is to prevent undesirable or unanticipated interference between simultaneously executing instruction sequences.

Use threads only where appropriate

Threads are not a "silver bullet" for improving application performance. An application not suited for multithreading may run slower following the introduction of multiple threads because of the overhead required to switch between threads.

Before you introduce threads into your application, determine whether it can benefit from their use. Use threads if your application needs:

- To react to many events simultaneously. Examples: An Internet browser or server.
- To provide a high level of responsiveness. Example: A user-interface implementation that can continue to respond to user actions even while the application is performing other computations.
- To take advantage of machines with multiple processors. Example: An application targeted for particular machine architectures.

Synchronization

Synchronization describes the set of mechanisms or processes for preventing undesirable interleaving of operations or interference between concurrent threads. A programmer may choose between one of two synchronization techniques: mutual exclusion or condition synchronization.

Mutual exclusion involves combining fine-grained atomic actions into coarse-grained actions and arranging to make these composite actions atomic.

Condition synchronization describes a process or mechanism that delays the execution of a thread until the program satisfies some predicate or condition.

A thread that is no longer executing because it is delayed or waiting on some synchronization mechanism is blocked. Once unblocked, awakened, or notified, a thread becomes runnable and eligible for further execution.

Two basic uses exist for thread synchronization: to protect the integrity of shared data and to communicate changes efficiently in program state between cooperating threads.

Java supports both mutual exclusion and condition synchronization via a mechanism provided by the object class.

Avoid synchronization

Synchronization is expensive. It takes time to acquire and release the synchronization objects necessary to synchronize a section of code. Moreover, synchronization serializes access to an object, minimizing concurrency. Think before you synchronize and only synchronize when it's truly necessary.

Do not arbitrarily synchronize every public method. Before synchronizing a method, consider whether it accesses shared and nonsynchronized states. If it does not - if the method only operates on its local variables, parameters, or synchronized objects-then synchronization is not required.

Do not synchronize classes that provide fundamental data types or structures. Let the users of the object determine whether synchronization is necessary. Users may always synchronize the object externally under the jurisdiction of a separate lock object.

Use synchronized wrappers to provide synchronized interfaces

Use synchronized wrappers to provide synchronized versions of classes. Synchronized wrappers provide the same interface as the original class, but its methods are synchronized. A static method of the wrapped class provides access to the synchronized wrapper. The following example demonstrates a stack, which has a default, nonsynchronized interface and a synchronized interface provided by a wrapper class:

```
public class Stack
{
    public void push(Object o) {...}
    public object pop() {...}
    public static Stack createSynchronizedStack()
    {
        return new SynchronizedStack();
    }
}
class SynchronizedStack extends Stack
{
    public synchronized void push(Object o)
    {
        super.push(o);
    }
    public synchronized object pop()
    {
        return super.pop();
    }
}
```

Do not synchronize an entire method if the method contains significant operations that do not need synchronization

To maximize concurrency in a program, we must minimize the frequency and duration of lock acquisition. A thread entering a synchronized method or block attempts to acquire a lock. Only one thread at a time may acquire ownership of a lock, so a lock may be used to serialize access to code or a program state. When a thread has finished executing in the synchronized section of code, it releases the lock so others threads may attempt to acquire ownership.

A method annotated with the synchronized keyword acquires a lock on the associated object at the beginning of the method and holds that lock until the end of the method. As is often the case, however, only a few operations within a method may require synchronization. In these situations, the method-level synchronization can be much too coarse.

The alternative to method-level synchronization is to use the synchronized block statement:

```
protected void processRequest()
{
    Request request=getNextRequest();
    Requestid id=request.getid{);
    synchronize(this)
    {
        RequestRandler handler=this.handlerMap.get(id);
        handler.handle(request);
    }
}
```

Avoid unnecessary synchronization when rearing or writing instance variables

The Java language guarantees read-and-write actions are atomic for object references and all primitives, except for type long and type double. We can, therefore, avoid the use of synchronization

when reading or writing atomic data. Be careful, though. If the value of an atomic variable is dependent on or related to the other variables, then synchronization is necessary.

In this example, the assignment of x and y trust be synchronized together because they are interdependent values:

```
public void synchronized setCenter(int x,int y)
{
    this.x=x;
    this.y=y;
}
```

The following example does not require synchronization because it uses an atomic assignment of an object reference:

```
public void setCenter(Point p)
{
    this point=(Point)p.clone();
}
```

Consider using notify() instead of not notifyAll()

The `notify()` method of `java.lang.Object` awakens a single thread waiting on a condition, while `notifyAll()` awakens all threads waiting on the condition. If possible, use `notify()` instead of `notifyAll()` because `notify()` is more efficient.

Use `notify()` when threads are waiting on a single condition and when only a single waiting thread may proceed at a time. For example, if the `notify()` signals that an item has been written to a queue, only one thread will be able to read the item from the queue. In this case, waking up more than one thread is wasteful.

Use `notifyAll()` when threads may wait on more than one condition or if it is possible for more than one thread to proceed in response to a signal.

Use the double-check pattern for synchronized initialization.

Use the double-check pattern in situations where synchronization is required during initialization, but not after it.

In the following code, the instance variable log needs to be initialized, but only if it is null. To prevent two threads from trying to initialize the field simultaneously, the function `getLog()` is declared synchronized:

```
synchronized Log getLog()
{
    if(this.log==null)
    {
        this.log = new Log();
    }
    return this.log;
}
```

This code also protects against simultaneous initialization, but it uses the double-check pattern to avoid synchronization except during initialization:

```
Log getLog()
{
    if(this.log==null)
    {
        synchronized (this)
        {
```

```
            if(this.log==null)
                this.log = new Log();
        }
    }
    return this.log;
}
```

Efficiency

Use lazy initialization.

Do not build something until you need it. If an object may not be needed during the normal course of program execution, then do not build the object until it is required.

Use an accessor method to gain access to the object. All users of that object, including within the same class, must use the accessor to get a reference to the object:

```
class PersonalFinance
{
    LoanRateCalculator _loanCalculator=null;

    LoanRateCalculator getLoanCalculator()
    {
        if(this._loanCalculator==null)
        {
            this._loanCalculator=new LoanRateCalculator();
        }
        return this._loanCalculator;
    }
}
```

Avoid creating unnecessary objects

This is especially important if the new objects have short life spans or are constructed, but never referenced. This not only wastes execution time to create the object, but it also uses time during garbage collection.

Redundant initialization, as illustrated in the following code, is quite common, and wasteful:

```
public Color getTextColor()
{
    Color c=new Color(...);
    if(this.m_state<2)
    {
        c=new Color(...);
    }
    return c;
}
```

Avoid creating an object until you know what you want:

```
public Color getTextColor()
{
    Color c = null;
    if(this.m_state < 2)
    {
        c=new Color(...);
    }
    else
    {
```

```
            c=new Color(...);
        }
        return c;
}
```

Reinitialize and reuse objects to avoid new object construction

Cache and reuse frequently created objects that have limited life spans.

Use accessor methods instead of constructors to reinitialize the object.

Take care to choose an implementation that does not need to create its own objects to manage the objects being cached. This would defeat the purpose!

Use a factory implementation to encapsulate mechanisms for caching and reusing objects. To manage these mechanisms properly, you must return objects obtained from an object factory back to the same factory. This means the association between an object and its factory must be maintained somewhere:

- In the class - a single static factory is associated with the class of the object, and that factory manages all objects of that class,
- In the object - the object maintains a reference to the factory that manages it.
- In the owner of the object - an "owner" of an object maintains a reference to the factory from which the object obtained.

Leave optimization for last.

> **First Rule of Optimization:**
> Don’t do it.
> **Second Rule of Optimization (For experts only):**
> Don’t do it yet.
> - *Michael Jackson,*
> *Michael Jackson Systems Ltd.*

Do not waste time optimizing code until you are sure you need to do it. Remember the 80-20 rule - 20 percent of the code in a system uses 80 percent o£ the resources (on average). If you are going to optimize, make sure it falls within the 20 percent portion.

Packaging conventions

This section contains guidelines for creating packages.

Place types that are commonly used, changed and released together, or mutually dependent on each other, into the same package

This rule encompasses several related package design principles:

> **The Common Reuse Principle**
> A package consists of classes you reuse together. If you use one of the classes in the package, you use all of them.

Place classes and interfaces you usually use together into the same package. Such classes are so closely coupled you cannot use one class without usually using the other. Some examples of closely related types include:

- Containers and iterators.
- Database tables, rows, and columns.
- Calendars, dates, and times.
- Points, lines, and polygons.

> **The Common Closure Principle**
> A package consists of classes, all closed against the same kind of changes. A change that affects the package affects all the classes in that package.

Combine classes that are likely to change at the same time, for the same reasons, into a single package. If two classes are so closely related that changing one of them usually involves changing the other, then place them in the same package.

> **The Reuse/Release Equivalence Principle**
> The unit of reuse is the unit of release. Effective reuse requires tracking of releases from a change control system. The package is the effective unit of reuse and release.

Treating individual classes as a unit of release is not very practical. A typical application may consist of tens of hundreds of classes, so releasing code on a class-by-class basis will dramatically complicate the integration and testing process, and dramatically increase the overall rate of change within the software.

A package provides a much more convenient mechanism for releasing several classes and interfaces. Each class or interface within a package may undergo several independent revisions between releases, but a package release captures only the latest version of each class and interface. Use packages as the primary unit of release and distribution.

> **The Acyclic Dependencies Principle**
> The dependency structure between packages must be a directed acyclic graph; there must be no cycles in the dependency structure.

If two packages directly or indirectly depend on each other, you cannot independently release one without releasing the other because changes in one package will often force changes in the other. Such cyclic dependencies dramatically increase the fragility of a system and can eliminate any reduction in schedule realized by assigning the development of each package to separate developers or teams.

Take steps to eliminate cyclic dependencies, either by combining the mutually dependent packages or by introducing a new package of abstractions that both packages can depend on instead of each other.

Isolate volatile classes and interfaces in separate packages

Avoid placing volatile classes and interfaces in the same package with stable classes and interfaces. If you use packages as your principle unit of release and distribution, users can gain access to the latest

changes in the volatile classes and interfaces only if you re-release the entire package. Each time you release the package, your users must absorb the cost of reintegrating and retesting against all the classes in the package, although many may not have changed.

Separate volatile classes from stable classes to reduce the code footprint affected by new releases of code, thereby reducing the impact on users of that code.

Avoid making packages that are difficult to change dependent on packages that are easy to change.

This rule derives from the following design principle:

> **The Stable Dependencies Principle**
> The dependencies between packages should be oriented in the direction of increasing stability. A package should only depend on packages more stable than it is.

If a package containing difficult-to-change types is dependent on a package that contains easy, or likely to change, types, then the dependent package effectively acts to impede change in the volatile package.

In a software system, especially one that is incrementally developed, some packages will always remain somewhat volatile. The developers of such a system must feel free to modify and extend these volatile packages to complete the implementation of the system and must be able to do so without worrying too much about downstream effects.

Do not create a package that depends on less stable packages. If necessary, create new abstractions that can be used to invert the relationship between the stable code and the unstable code.

Maximize abstraction to maximize stability

This rule derives from the following design principles:

> **The Stable Abstractions Principle**
> The stability exhibited by a package is directly proportional to its level of abstraction. The more abstract a package is, the more stable it tends to be. The more concrete a package is, the more unstable it tends to be.

Use stable abstractions to create stable packages. Capture high-level, stable concepts in abstract classes and interfaces, and provide implementations using concrete classes. Separate abstract classes and interfaces from the concrete classes to form stable and unstable packages. This ensures the derived classes in the unstable packages depend on the abstract base classes and interfaces in the stable packages.

Capture high-level design and architecture as stable abstractions organized into stable packages.

To plan and manage a software development effort successfully, the top-level design must stabilize quickly and remain that way. No development manager can hope to accurately plan, estimate, schedule, and allocate resources if the architecture of the system continues to change.

Once the design of the high-level architecture is complete, use packages to separate the stable parts of the design from the volatile implementation. Create packages to capture the high-level abstractions of the design. Place the detailed implementation of those abstractions into separate packages that depend on the high-level abstract packages.

Summary

1. Adhere to the style of the original
2. Adhere to the Principle of Least Astonishment
3. Do it right the first time
4. Document any deviations
5. Indent nested code
6. Break up long lines
7. Declare class interfaces on a separate line
8. Declare method exceptions on a separate line
9. Include white space
10. Do not use "hard" tabs
11. Use meaningful names
12. Use familiar names
13. Question excessively long names
14. Join the vowel generation
15. Capitalize only the first letter in acronyms
16. Do not use names that differ only in case
17. Use the reversed, lowercase form of your organization's Internet domain name as the root qualified for names of your organization's private packages
18. Use a single, lowercase word as a root name of each package
19. 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
20. Capitalize the first letter of each word that appears in a class or interface name
21. Prefix the names of non-public classes or interfaces with an underscore
22. Use nouns when naming classes
23. Pluralize the names of classes that group related attributes, static services, or constants
24. Use nouns or adjectives when naming interfaces
25. Prefix names of "true" interfaces with 'I'
26. Use lowercase for the first word and capitalize only the first letter of each subsequent word that appears in a method name
27. Prefix the names of private methods with an underscore
28. Use verbs with naming methods
29. Follow the JavaBeans conventions for naming property accessor methods
30. Use lowercase for the first word and capitalize only the first letter of every subsequent word that appears in a variable name
31. Use nouns to name variables

32. Pluralize the names of collection references
33. Establish and use a set of standard names for "throwaway" variables
34. You may qualify field variables with "this" to distinguish them from local variables
35. Prefix the names of private fields with "m_"
36. Prefix the names of package fields with an underscore
37. When a constructor or "set" method assigns a parameter to a field, give that parameter the same name as a field
38. Use uppercase letters for each word and separate each pair of words with an underscore when naming constants
39. Write documentation for those who must use your code and those who must maintain it
40. Keep comments and code in sync
41. Use the active voice and omit needless words
42. Use documentation comments to describe the programming interface
43. Use standard comments to hide code without removing it
44. Use one-line comments to explain implementation details
45. Describe the programming interface before you write the code
46. Document at least public and protected members
47. Provide a summary description and overview for each package
48. Provide a summary description and overview for each application or group of packages
49. Use a single consistent format and organization for all documentation comments
50. Wrap keywords, identifiers, and constants with <code>...</code> tags
51. Wrap code with <pre>...</pre> tags
52. Consider marking the first occurrence of an identifier with a {@link} tag
53. Establish and use a fixed ordering for Javadoc tags
54. Write in the third person narrative form
55. Write summary descriptions that stand alone
56. Omit the subject in summary descriptions of actions or services
57. Omit the subject and the verb in summary descriptions of things
58. Use "this" rather than "the" when referring to instances of the current class
59. Do not add parentheses to a method or constructor name unless you want to specify a particular signature
60. Provide a summary description for each class, interface, fell and method
61. Fully describe the signature of each method
62. Include examples
63. Document preconditions, postconditions, and invariant conditions
64. Document known defects and deficiencies
65. Document synchronization semantics

66. Add internal comments only if they will aid others in understanding your code
67. Describe why the code is doing what it does, not what the code is doing
68. Avoid the use of end-line comments
69. Explain local variable declarations with an end-line comment
70. Establish and use a set of keywords to flag unresolved issues
71. Label closing braces in highly nested control structures
72. Add a "fall-through" comment between two ease labels, if no break statement separates those labels
73. Label empty statements
74. Consider declaring classes representing fundamental data types as final
75. Build concrete types from native types and other concrete types
76. Define small classes and small methods
77. Define subclasses so they may be used anywhere their superclasses may be used
78. Make all fields private
79. Use polymorphism instead of instanceof
80. Wrap general-purpose classes that operate on java.1ang.Object to provide static type checking
81. Encapsulate enumerations as classes
82. Replace repeated nontrivial expressions with equivalent methods
83. Use block statements instead of expression statements in control flow constructs
84. Clarify the order of operations with parentheses
85. Always code a break statement in the last case of a switch statement
86. Use equals(), not ==, to test for equality of objects.
87. Always construct objects in a valid state.
88. Do not call nonfinal methods from within a constructor
89. Use nested constructors to eliminate redundant code
90. Use unchecked run-time exceptions to report serious unexpected errors that may indicate an error in the program's logic
91. Use checked exceptions to report errors that may occur, however rarely, under normal program operation.
92. Use return codes to report expected state changes
93. Only convert exceptions to add information.
94. Do not silently absorb a run-time or error exception
95. Use a finally block to release resources
96. Program by contract
97. Use dead code elimination to implement assertions
98. Use assertions to catch logic errors in your code

99. Use assertions to test pre- and postconditions of a method

100. Use threads only where appropriate

101. Avoid synchronization

102. Use synchronized wrappers to provide synchronized interfaces

103. Do not synchronize an entire method if the method contains significant operations that do not need synchronization

104. Avoid unnecessary synchronization when rearing or writing instance variables

105. Consider using notify() instead of not notifyAll()

106. Use the double-check pattern for synchronized initialization.

107. Use lazy initialization.

108. Avoid creating unnecessary objects

109. Reinitialize and reuse objects to avoid new object construction

110. Leave optimization for last.

111. Place types that are commonly used, changed and released together, or mutually dependent on each other, into the same package

112. Isolate volatile classes and interfaces in separate packages

113. Avoid making packages that are difficult to change dependent on packages that are easy to change.

114. Maximize abstraction to maximize stability

115. Capture high-level design and architecture as stable abstractions organized into stable packages.

Glossary

abstract class
A class that exists only as a superclass of another class and can never be directly instantiated. In Java, an abstract class contains or inherits one or more abstract methods or includes the abstract keyword in its definition.

abstract method
A method that has no implementation.

abstract data type
Defines a type that may have many implementations. Abstract data types include things like stacks, queues, and trees.

abstract type
Defines the type for a set of objects, where each object must also belong to a set of objects that conform to a known subtype of the abstract type. An abstract type may have one or more implementations.

abstraction
The process and result of extracting the common or general characteristics from a set of similar entities.

accessor
A method that sets or gets the value of an object property or attribute.

active object
An object that possesses its own thread of control.

acyclic dependency
A dependency relationship where one entity has a direct or indirect dependency on a second entity, but the second entity has no direct or indirect dependency on the first.

aggregation
An association representing a whole-part containment relationship.

architecture
A description of the organization and structure of a software system.

argument
Data item specified as a parameter in a method call.

assertion
A statement about the truth of a logical expression.

attribute
A named characteristic or property of a type, class, or object.

behavior
The activities and effects produced by an object in response to an event.

binary compatible
A situation where one version of a software component may be directly and transparently substituted for another version of that component without recompiling the component's clients.

block statement
The Java language construct that combines one or more statement expressions into a single compound statement, by enclosing them in curly braces "`{ . . . }`".

Boolean
An enumerated type whose values are true and false.

built-in type
A data type defined as part of the language. The built-in or native types defined by Java include the primitive types boolean, byte, char, double, float, int, long, short, and void, and the various classes and interfaces defined in the standard Java API, such as Object, String, Thread, and so forth.

checked exception
Any exception that is not derived from `java.lang.RuntimeException` or `java.lang.Error`, or that appears in the throws clause of a method. A method that throws, or is a recipient of, a checked exception must handle the exception internally or otherwise declare the exception in its own throws clause.

class
A set of objects that share the same attributes and behavior.

class hierarchy
A set of classes associated by inheritance relationships.

client
An entity that requests a service from another entity.

cohesion
The degree to which two or more entities belong together or relate to each other.

component
A physical and discrete software entity that conforms to a set of interfaces.

composition
A form of aggregation where an object is composed of other objects.

concrete class
A completely specified class that may be directly instantiated. A concrete class defines a specific implementation for an abstract class or type.

concrete type
A type that may be directly instantiated. A concrete type may refine or extend an abstract type.

concurrency
The degree by which two or more activities occur or make progress at the same time.

constraint
A restriction on the value or behavior of an entity.

constructor
A special method that initializes a new instance of a class.

container
An object whose purpose is to contain and manipulate other objects.

contract
A clear description of the responsibilities and constraints that apply between a client and a type, class, or method.

coupling
The degree to which two or more entities are dependent on each other.

critical section
: A block of code that allows only one thread at a time to enter and execute the instructions within that block. Any threads attempting to enter a critical section while another thread is already executing within that section are blocked until the original thread exits.

cyclic dependency
: A dependency relationship where one entity has a direct or indirect dependency on a second entity and the second entity also has a direct or indirect dependency on the first.

data type
: A primitive or built-in type that represents pure data and has no distinct identity as an object.

delegation
: The act of passing a message, and responsibility, from one object to a second object to elicit a desired response.

dependency
: A relationship where the semantic characteristics of one entity rely upon and constrain the semantic characteristics of another entity.

derivation
: The process of defining a new type or class by specializing or extending the behavior and attributes of an existing type or class.

generalization
: The process of extracting the common or general characteristics from a set of similar entities to create a new entity that possesses these common characteristics.

documentation comment
: A comment that begins with a "`/**`" and ends with "`*/`"and contains a description and special tags that are parsed by the Javadoc utility to produce documentation.

domain
: An area of expertise, knowledge, or activity.

encapsulation
: The degree to which an appropriate mechanism is used to hide the internal data, structure, and implementation of an object or other entity.

enumeration
: A type that defines a list of named values that make up the allowable range for values of that type.

factor
: The act of reorganizing one or more types or classes by extracting responsibilities from existing classes and synthesizing new classes to handle these responsibilities.

field
: An instance variable or data member of an object.

fundamental data type
: A type that typically requires only one implementation and is commonly used to construct other, more useful types. Dates, complex numbers, linked-lists, and vectors are examples of common fundamental data types.

implementation
: The concrete realization of a contract defined by a type, abstract class, or interface. The actual code.

implementation class

A concrete class that provides an implementation for a type, abstract class, or interface.

implementation inheritance
The action or mechanism by which a subclass inherits the implementation and interface from one or more parent classes.

inheritance
The mechanism by which more specialized entities acquire or incorporate the responsibilities or implementation of more generalized entities.

inner class
A class defined within the scope of another class.

instance
The result of instantiating a class-the concrete representation of an object.

instantiation
The action or mechanism by which a type or class is reified to create an actual object. The act of allocating and initializing an object from a class.

interface
The methods exposed by a type, class, or object. Also a set of operations that define an abstract service.

interface inheritance
The action or mechanism by which a subtype or subinterface inherits the interface from one or more parent types or interfaces.

invariant
An expression that describes the well-defined, legal states of an object.

keyword
A language construct. The keywords of the Java language include:

abstract	finally	public
boolean	float	return
break	for	short
byte	[goto]	static
case	if	super
catch	implements	switch
char	import	synchronized
class	instanceof	this
[const]	int	throw
continue	interface	throws
default	long	transient
do	native	try
double	new	void
else	package	volatile
extends	private	while
final	protected	

Bracketed keywords are reserved but not used.

lazy initialization
When an implementation delays the initialization of a data value until the first use or access of the data value.

local variable
A variable that is automatically allocated and initialized on the call "stack." Includes variables bound as function arguments.

method
The implementation of an operation. An operation defined by an interface or class.

multiple inheritance
Inheritance relationship where a subtype inherits from two or more supertypes. Java supports multiple inheritance by allowing an interface to extend multiple interfaces.

mutex
A synchronization mechanism used to provide mutually exclusive access to a resource.

native type
A data type defined as part of the language. The native or built-in types defined by Java include the primitive types boolean, byte, char, double, float, int, long, short, and void, and the various classes and interfaces defined in the standard Java API, such as Object, String, and Thread.

object
The result of instantiating a class. An entity with state, behavior, and identity.

operation
A service that may be requested from an object to effect behavior. Alternatively viewed as a message sent from a client to an object.

package
A mechanism organizing and naming a collection of related classes.

package access
The default access-control characteristic applied to interfaces, classes, and class members. Class members with package access are accessible only to code within the same package and are heritable by subclasses in the same package. Classes and interfaces with package access are not visible to code outside the package. They are only accessible and extendable by classes and interfaces in the same package.

parameter
A variable that is bound to an argument value passed into a method.

polymorphic
A trait or characteristic of an object whereby that object can appear as several different types at the same time.

polymorphism
The concept or mechanism by which objects of different types inherit the responsibility for implementing the same operation, but respond differently to the invocation of that operation.

postcondition
A constraint or assertion that must hold true following the completion of an operation.

precondition
A constraint or assertion that must hold true at the start of an operation.

primitive type
A basic language type that represents a pure value and has no distinct identity as an object. The primitives provided by Java include boolean, byte, char, double, float, int, long, and short.

private access
An access-control characteristic applied to class members. Class members declared with the private access modifier are only accessible to code in the same class and are not inherited by subclasses.

property

A named characteristic or attribute of a type, class, or object.

protected access
An access-control characteristic applied to class members. Class members declared with the protected access modifier are accessible to code in the same class and package, and from code in subclasses, and they are inherited by subclasses.

public access
An access-control characteristic applied to interfaces, classes, and class members. Class members declared with the public access modifier are accessible anywhere the class is accessible and are inherited by subclasses. Classes and interfaces declared with the public access modifier are visible, accessible, and heritable outside of a package.

qualifier
A name or value used to locate or identify a particular entity within a set of similar entities.

realization
A relationship where one entity agrees to abide by or to carry out the contract specified by another entity.

responsibility
A purpose or obligation assigned to a type.

role
The set of responsibilities associated with an entity that participates in a specific relationship. A Java interface often defines a role for an object.

service
One or more operations provided by a type, class, or object to accomplish useful work on behalf of one or more clients.

signature
The name, parameter types, return type, and possible exceptions associated with an operation.

state
The condition or value of an object between events.

static type checking
Compile-time verification of the assumptions made about the use of object reference and data value types.

subclass
A class that inherits attributes and methods from another class.

subtype
The more specific type in a specialization-generalization relationship.

superclass
A class whose attributes and methods are inherited by another class.

supertype
The more general type in a specialization-generalization relationship.

synchronization
The process or mechanism used to preserve the invariant states of a program or object in the presence of multiple threads.

synchronized

A characteristic of a method or a block of code. A synchronized method or block allows only one thread at a time to execute within the critical section defined by that method or block.

thread
A single flow of control flow within a process that executes a sequence of instructions in an independent execution context.

type
Defines the common responsibilities, behavior, and operations associated with a set of similar objects. A type does not define an implementation.

unchecked exception
Any exception that is derived from `java.lang.RuntimeException` or `java.lang.Error`. A method that throws, or is a recipient of, an unchecked exception is not required to handle the exception or declare the exception in its throws clause.

variable
A typed, named container for holding object references or a data values.

visibility
The degree to which an entity may be accessed from outside of a particular scope.