Coding Style

The following is an abridged version of Doug Lea's Coding Style Draft.

1. Structure and Documentation

1.1. Packages

Create a new java package for each self-contained project or group of related functionality. Create and use directories in accord with java package conventions.
Consider writing an `index.html` file in each directory briefly outlining the purpose and structure of the package.

1.2. Program Files

Place each class in a separate file. This applies even to non-public classes (which are allowed by the Java compiler to be placed in the same file as the main class using them) except in the case of one-shot usages where the non-public class cannot conceivably be used outside of its context.

Begin each file with a comment including:

- The file name and/or related identifying information including, if applicable, copyright information.
- A history table listing dates, authors, and summaries of changes.
- If the file contains more than one class, list the classes, along with a very brief description of each.
- If the file introduces a principal entry point for a package, briefly describe the rationale for constructing the package.

Immediately follow each file header with:

- The package name
- The import list.

Example:

```
/*
   File: Example.java
   Date      Author      Changes
   Sep 1  95 Doug Lea    Created
   Sep 13 95 Doug Lea    Added new doc conventions
*/

package demo;
import java.util.NoSuchElementException;
```

1.2.1. Classes and Interfaces

Write all /** ... **/ **comments using `javadoc` conventions**. (Even though not required by javadoc, end each /** comment with **/ to make it easier to read and check.)

Preface each class with a / ... **/ comment** describing the purpose of the class, guaranteed invariants, usage instructions, and/or usage examples. Also include any reminders or disclaimers about required or desired improvements. Use HTML format, with added tags:

```
@author author-name
@version version number of class
@see string
```

```
@see URL
@see classname#methodname

Example:

/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 * Window win = new Window(parent);
 * win.show();
 * </pre>
 *
 * @see awt.BaseWindow
 * @see awt.Button
 * @version 1.2 31 Jan 1995
 * @author Bozo the Clown
**/
class Window extends BaseWindow {
...
}
```

1.2.2. Class Variables

Use javadoc conventions to describe nature, purpose, constraints, and usage of instances variables and static variables. Use HTML format, with added tags:

```
@see string
@see URL
@see classname#methodname
```

Example:

```
    /**
     * The current number of elements.
     * must be non-negative, and less than or equal to capacity.
    **/
    protected int count_;
```

1.2.3. Methods

Use javadoc conventions to describe nature, purpose, preconditions, effects, algorithmic notes, usage instructions, reminders, etc. Use HTML format, with added tags:

```
@param paramName description.
@return description of return value
@exception exceptionName description
@see string
@see URL
@see classname#methodname
```

Be as precise as reasonably possible in documenting effects. Here are some conventions and practices for semi-formal specifications:

@return condition: (condition)
: describes postconditions and effects true upon return of a method.

@exception exceptionName IF (condition)
: indicates the conditions under which each exception can be thrown. Include conditions under which uncommon unchecked (undeclared) exceptions can be thrown.

@param paramname WHERE (condition)

indicates restrictions on argument values. Alternatively, if so implemented, list restrictions alongside the resulting exceptions, for example IllegalArgumentException. In particular, indicate whether reference arguments are allowed to be null.

WHEN (condition)
indicates that actions use guarded waits until the condition holds.

RELY (condition)
describes assumptions about execution context. In particular, relying on other actions in other threads to terminate or provide notifications.

GENERATE T
to describe new entities (for the main example, Threads) constructed in the course of the method.

ATOMIC
indicates whether actions are guaranteed to be uninterfered with by actions in other threads (normally as implemented via synchronized methods or blocks).

PREV(obj)
refers to the state of an object at the onset of a method.

OUT(message)
describes messages (including notifications such as notifyAll) that are sent to other objects as required aspects of functionality, or referrred to in describing the effects of other methods.

foreach (int i in lo .. hi) predicate
means that predicate holds for each value of i.

foreach (Object x in e) predicate
means that the predicate holds for each element of a collection or enumeration.

foreach (Type x) predicate
means that the predicate holds for each instance of Type.

-->
means `implies'.

unique
means that the value is different than any other. For example, a unique instance variable that always refers to an object that is not referenced by any other object.

fixed
means that the value is never changed after it is initialized.

EQUIVALENT to { code segment }
documents convenience or specialized methods that can be defined in terms of a few operations using other methods.

Example:

```
/**
 * Insert element at front of the sequence
 *
 * @param element the element to add
 * @return condition:
 * <PRE>
 *  size() == PREV(this).size()+1 &&
 *  at(0).equals(element) &&
 *  foreach (int i in 1..size()-1) at(i).equals(PREV(this).at(i-1))
 * </PRE>
**/

public void addFirst(Object element);
```

1.2.4. Local declarations, statements, and expressions

Use /* ... */ comments to describe algorithmic details, notes, and related documentation that spans more than a few code statements.

Example:

```
/*
 * Strategy:
 *    1. Find the node
 *    2. Clone it
 *    3. Ask inserter to add clone
 *    4. If successful, delete node
 */
```

Use Running // comments to clarify non-obvious code. But don't bother adding such comments to obvious code; instead try to make code obvious!

Example:

```
int index = -1; // -1 serves as flag meaning the index isn't valid
```

Or, often better:

```
static final int INVALID= -1;
int index = INVALID;
```

Use any consistent set of choices for code layout, including:

- Number of spaces to indent.
- Left-brace ("{"') placement at end of line or beginning of next line.
- Maximum line length.
- Spill-over indentation for breaking up long lines.
- Declare all class variables in one place (by normal convention, at the top of the class).

1.2.5. Naming Conventions

Packages: lowercase.
Consider using the recommended domain-based conventions described in the Java Language Specification, page 107 as prefixes. (For example, edu.oswego.cs.dl.)

Files
The java compiler enforces the convention that file names have the same base name as the public class they define.

Classes: `CapitalizedWithInternalWordsAlsoCapitalized`

Exception class: `ClassNameEndsWithException`.

Interface. When necessary to distinguish from similarly named classes: `InterfaceNameEndsWithIfc`.

Class. When necessary to distinguish from similarly named interfaces: `ClassNameEndsWithImpl` OR `ClassNameEndsWithObject`

Constants (finals): `UPPER_CASE_WITH_UNDERSCORES`

Variables: `firstWordLowerCaseButInternalWordsCapitalized`

Methods: `firstWordLowerCaseButInternalWordsCapitalized()`

Factory method for objects of type X: newX

Converter method that returns objects of type X: toX

Method that reports an attribute x of type X: X getX().

Method that changes an attribute x of type X: void setX(X value).

2. Recommendations

1. Minimize * forms of import. Be precise about what you are importing. Check that all declared imports are actually used.

2. When sensible, consider writing a main for the principal class in each program file. The main should provide a simple unit test or demo.

3. For self-standing application programs, the class with main should be separate from those containing normal classes.

4. Consider writing template files for the most common kinds of class files you create: Applets, library classes, application classes.

5. If you can conceive of someone else implementing a class's functionality differently, define an interface, not an abstract class. Generally, use abstract classes only when they are "partially abstract"; i.e., they implement some functionality that must be shared across all subclasses.

6. Consider whether any class should implement Cloneable and/or Serializable.

7. Declare a class as `final` only if it is a subclass or implementation of a class or interface declaring all of its non-implementation-specific methods. (And similarly for final methods).

8. Never declare instance variables as `public`.

9. Minimize reliance on implicit initializers for instance variables (such as the fact that reference variables are initialized to `null`).

10. Minimize statics (except for `static final` constants).
    Rationale: Static variables act like globals in non-OO languages. They make methods more context-dependent, hide possible side-effects, sometimes present synchronized access problems, and are the source of fragile, non-extensible constructions. Also, neither static variables nor methods are overridable in any useful sense in subclasses.

11. Generally prefer `long` to `int`, and `double` to `float`. But use `int` for compatibility with standard Java constructs and classes (for the major example, array indexing, and all of the things this implies, for example about maximum sizes of arrays, etc).

12. Use `final` and/or comment conventions to indicate whether instance variables that never have their values changed after construction are intended to be constant (immutable) for the lifetime of the object (versus those that just so happen not to get assigned in a class, but could in a subclass).

13. Generally prefer `protected` to `private`.
    Rationale: Unless you have good reason for sealing-in a particular strategy for using a variable or method, you might as well plan for change via subclassing. On the other hand, this almost always entails more work. Basing other code in a base class around `protected` variables and methods is harder, since you you have to either loosen or check assumptions about their properties. (Note that in Java, `protected` methods are also accessible from unrelated classes in the same package. There is hardly ever any reason to exploit this though.)

14. Avoid unnecessary instance variable access and update methods. Write get/set-style methods only when they are intrinsic aspects of functionality.

15. Minimize direct internal access to instance variables inside methods. Use `protected` access and update methods instead (or sometimes public ones if they exist anyway).

16. Avoid giving a variable the same name as one in a superclass.

17. Prefer declaring arrays as Type[] arrayName rather than Type arrayName[].

18. Ensure that non-private statics have sensible values even if no instances are ever created. (Similarly ensure that static methods can be executed sensibly.) Use static initializers (static { ... }) if necessary.

19. Write methods that only do "one thing". In particular, separate out methods that change object state from those that just rely upon it. For a classic example in a Stack, prefer having two methods Object top() and void removeTop() versus the single method Object pop() that does both.

20. Define return types as `void` unless they return results that are not (easily) accessible otherwise. (i.e., hardly ever write "`return this`").

21. Avoid overloading methods on argument type. (Overriding on arity is OK, as in having a one-argument version versus a two-argument version). If you need to specialize behavior according to the class of an argument, consider instead choosing a general type for the nominal argument type (often Object) and using conditionals checking instanceof. Alternatives include techniques such as double-dispatching, or often best, reformulating methods (and/or those of their arguments) to remove dependence on exact argument type.

22. Use method `equals` instead of operator == when comparing objects. In particular, do not use == to compare Strings.

23. Declare a local variable only at that point in the code where you know what its initial value should be.

24. Declare and initialize a new local variable rather than reusing (reassigning) an existing one whose value happens to no longer be used at that program point.

25. Assign null to any reference variable that is no longer being used. (This includes, especially, elements of arrays.) Rationale: Enables garbage collection.

26. Avoid assignments ("=") inside `if` and `while` conditions.

27. Document cases where the return value of a called method is ignored.

28. Ensure that there is ultimately a `catch` for all unchecked exceptions that can be dealt with.
Rationale: Java allows you to not bother declaring or catching some common easily-handlable exceptions, for example `java.util.NoSuchElementException`. Declare and catch them anyway.

29. Embed casts in conditionals. For example:

```
C cx = null;
if (x instanceof C) cx = (C)x;
else evasiveAction();
```