JAVA CODING STYLE GUIDELINES

By : Med Asta

Contenu

I. Package Names 2
II. Class and Interface Names 2
III. Method Names 3
IV. Attribute and Local Variable Names 4
V. Returning Arrays and Lists 5
Don't "Hide" Names 6
A. Class Attributes 6
B. Modifier Usage 6
C. Class and Package Imports 7
D. Methods 7
E. Keep it simple 8
F. Place Constants on Left Side of Comparisons 8
3. Optimization vs Abstraction 8
i. Overall Guidelines 9
ii. Standard .java file header 9
iii. Method Documentation 9
iv. Use the `@deprecated` Tag 10

❖ Naming Conventions

I. Package Names

Following a developing, de-facto standard in the Java world, ***all packages should be prefixed with: "com.asta"***

Further subdivision is up to the individual project teams. In the case of common code, ***reusable components will be placed within a package named according to the component, such as "com.asta.security" or "com.asta.ui".** Business objects will be segregated by business domains, or product offerings, as appropriate.

***Package names should always be expressed in lowercase**.

In ***the event that multiple words** are used to define a level in the package naming hierarchy, these ***words should be run together with no separating space or other character**.

***Abbreviations can be used if they are commonly used**, i.e. ui for user interface. An incorrect package name would be "com.asta.user_interface" due to the separating character between "user" and "interface".

Acceptable: `com.asta.ui` **Unacceptable:** `com.asta.user_interface`

II. Class and Interface Names

***Class names are always nouns, not verbs**. Avoid making a noun out of a verb, example DividerClass. **If you are having difficulty naming a class then perhaps it is a bad class.**

*** Interface names should always be an adjective** (wherever possible) **describing the enforced behaviors of the class** (noun). Preferably, said adjective should end in "able" following an emerging preference in Java. i.e. Clonable, Versionable, Taggable, etc.

*** Class, and interface names begin with an uppercase letter**, and should not be pluralized unless it is a collection class (see below).

Acceptable:`class FoodItem,  interface Digestable`
Unacceptable: `class fooditem  ,class Crackers  ,interface Eat`

***Naming collection classes (in the generic sense of collection) can be tricky with respect to pluralization**. In general each collection *should be **identified as a plural item**, but not redundantly so. ***If you are a collection type as part of the class name (List, Map, etc.) it is not necessary to use the plural form in the class name.** ***If you are not using the collection type in the name it is necessary to pluralize the name**. ***If you are extending one of the java colletction class** (Map, HashMap, List, ArrayList, Collection, etc.) **it is good practice to use the name of the collection type in the class name.**

Acceptable:`class FoodItems extends Object` ,`class FoodItemList extends ArrayList` ,`class FoodItemMap extends HashMap`
Unacceptable:`class FoodItem extends ArrayList` , `class FoodItemsList extends ArrayList`

***Class names should be descriptive in nature without implying implementation.** The internal implementation of an object should be encapsulated and not visible outside the object. ***Since implementation can change, to imply implementation in the name forces the class name and all references to it to change or else the code can become misleading.**

Acceptable: `AbstractManagedPanel` ,`LayeredPanel` **Unacceptable:** `PanelLayerArray`

***When using multiple words in a class name, the words should be concatenated with no separating characters between them. The first letter of each word should be capitalized**.

Acceptable: `InverntoryItem` **Unacceptable:** `Inverntory_item` , `Inventoryitem`

Other than prefixes, no abbreviations should be used ***unless it is a well-known abbreviation.**

Acceptable:`CD=Compact Disc` ,`US=United States`
Unacceptable: `Cust=Customer` , `DLR=Dealer`

III. Method Names

***Method names are typically verbs.** However **they can also be nouns for example accessor methods** (see accessor methods below**).** ***In general, when a method modifies the object (or a related object) somehow use a verb appropriate to the nature of the modification**. i.e. set, free, sort etc. If the method is an accessor method use a noun appropriate to the information that is returned.

Acceptable: `label getTag(),void setWidth(int) ,void resetCounter()`
Unacceptable: `label tagIs(), void counterNew()`

* **Names should reflect exactly what the method does** (no more or no less).
* **A method should only have a single purpose**. If your method contains too much functionality, then you should break it into more than one method.

***Strive for names that promote self-documenting code**. The method name should read well in the code. Picture how the method will appear in your code.

* **Method and function names begin with a lowercase letter. The initial letter of any subsequent words in the name is capitalized, and underscores are not used to separate words.**

Acceptable: `setInitState(),getAttributeCollection()`
Unacceptable: `set_init_state(), getattributecollection()`

Method names should be defined so as to describe the function of the method without implying implementation.

Acceptable: `addItem(),getItem()`
Unacceptable: `addItemToVector(),getHashItem()`

IV. Attribute and Local Variable Names

Do not use abbreviations, * use full names.

* **Variable names begin with a lowercase letter. The initial letter of any subsequent words in the name are capitalized, and underscores are not used to separate words (or scope variables).**

Clarity of variable names can be increased by providing some indication of the type of class they might become. ***Attributes that are not collections should not be pluralized.**

Acceptable:`Item menuItem; JPanel managerJPanel;`

Unacceptable:`JPanel Managerpanel; JPanel Manager_panel;int _localInt; InventoryItem i;`

***Collection classes, such as vectors and hashes should always be pluralized.**
Alternately, collection classes can also be prefixed with the word some.

Acceptable:`Vector menuItems; Vector menuItemsVector; Vector someMenuItems`
Unacceptable: `Vector menuItemVector; Vector aBunchOfMenu`

Name variables with the most abstract class that they can hold. For example if `startButton` could be any control object, then it should be named a `startControl`.

*** If the variable truly represents an anonymous object but is restricted by an interface, then including the interface name in the variable can increase clarity. i.e. `clonableInventoryItem`.**

**** Declare each variable separately on a single line. Do not comma separate variables of the same type.**

Acceptable: `int counter; int lastCounter;`

Unacceptable: `int counter, lastCounter;`

*** Constant values should have uppercase letters for each word and each word should be separated by an underscore. The capitalization of constants helps to distinguish them from other nonfinal variables.**

Acceptable:
```
public final static int MAX_AGE = 100;
public final static Color RED = new Color(count++);
```

Unacceptable:
```
public final static int MAXAGE = 100;
public final static int maxAge = 100;
public final static int MaxAge = 100;
```

V. Returning Arrays and Lists

*** Any method that will returns an list of homogeneous or heterogeneous items should return a Collection (or other collection class) object -- never an array.**

Example: for a method that returns a list of keys represented as strings.

Acceptable: `List getKeys();` **Unacceptable:** `String[] getKeys();`

***Also, any method that returns a Collection should always return a valid Collection -- never null. However, the returned Collection can be empty.**

Acceptable:
```
public ArrayList getKeys() {
   if (0 == this.numValidKeys) {
      return new ArrayList();
   }
   return myKeyList;
}
```

Unacceptable:
```
public ArrayList getKeys() {
   if (0 == this.numValidKeys) {
      return null;
   }
return myKeyList;
}
```

Don't "Hide" Names

Name hiding refers to the practice of naming a local variable, argument, or field the same (or similar) as that of another of greater scope. For example, if you have a class attribute called `firstName` do not create a local variable called `firstName`or anything close to it, such as `firstNames` or `fName`. Try to avoid this, it makes you code difficult to understand and prone to bugs because other developers will misread your intentions and create difficult to detect errors.

❖ Usage Conventions

A. Class Attributes

* **Class attributes should always be accessed through accessors and mutators (getters and setters)**. Accessors and mutators are methods used to return and set the attribute value. These methods typically begin with "get" or "set", followed by the attribute name. As with other methods, the first letter of the method name should be in lowercase.
Note: **Boolean accessors typically begin with "is" i.e. isFixed(), and return a Boolean value.** Example:

Attribute(s): `int useCounter;`

Getter: `int getUseCounter(){return useCounter;}`

Setter(s):
```
void setUseCounter(int count){  useCounter = count; }

void incrementUseCounter() {setUseCounter(getUseCounter()+1);}

void decrementUseCounter() {setUseCounter(getUseCounter()-1);}
```

Be safe and initialize all local variables at creation and all class attributes in the constructor(s)!

B. Modifier Usage

Always use "public", "protected" and "private" keywords, **and use them correctly.** ***Attributes should rarely be public as this violates encapsulation**. ***Declare attributes as "private" and access them through public or protected getters and setters.** Use static methods to define constants instead of public static attributes (e.g. `UILayerdPanel.getActiveLayers()`). * **Methods in the public interface of a class should be declared as public. Other methods should be declared as protected.**

C. Class and Package Imports

To make for more readable code, types used in code should be imported rather that fully qualifying the class name. In general, import only those classes necessary. Importing java.util.* when only two or three classes are needed will increase the runtime footprint of the application.

Acceptable: `import java.util.Date; Date aDate = new Date();`
Unacceptable: `java.util.Date aDate = new java.util.Date();`

Importing rather than fully qualifying references adds an advantage in architecting an application. For example, imagine a platform-specific implementation of some classes which do CD drive manipulation. Imagine that the behavior resides in the class CDPlayer. If references are coded as

```
com.asta.ui.winnt.CDPlayer aPlayer = new com.asta.ui.winnt.CDPlayer();
```

then all those references would have to change when you port to UNIX. However, if the class is referenced as below:

```
import com.asta.ui.winnt.CDPlayer;   CDPlayer aPlayer = new CDPlayer()
```

Then changing to UNIX simply requires
importing `com.asta.ui.unix.CDPlayer` instead.

Imports should be done on specific classes only. Imports should not include entire packages.

Acceptable:
```
import javax.servlet.http.HttpServletRequest;
import javax.ejb.SessionBean;
```

Unacceptable: `import javax.servlet.http.*; import javax.ejb.*;`

Importing specific classes rather than entire packages simplifies class resolution for both the compiler and the developer, since there is no ambiguity about the package of an imported class. Also, importing java.util.* when only two or three classes are needed will increase the runtime footprint of the application.

D. Methods

Methods in well-designed object-oriented code are short. Strive to keep methods less than 10 lines. Reconsider methods that are over a page in length, breaking them into several methods representing smaller blocks of functionality.

Break up long methods into small methods. This promotes code reuse and allows for more combinations of methods. If the number of methods grows to be difficult to understand, then look at decomposing the class into more than one class.

Follow the 30-second rule. Another programmer should be able to look at your method and fully understand what it does, why it does it and how it does it in less than 30-seconds. If that is not possible, then your code is too complex and difficult to maintain. A good rule of thumb is that a method should be no more than a screen in length.

E. Keep it simple

Avoid nesting blocks of statements more than 2 or 3 levels deep. This adds to the complexity of the code. A method should be easy to read and understand. Easy to maintain is the goal.

Avoid nesting method calls too deeply, since this can introduce undue complexity.

Avoid using compound predicates:

```
if (x>0 && x<100 && y>0 && y<100 || z==1000)
```

Think of all the combinations you will have to write to adequately test the above condition (2^5 or 32 different combinations).

F. Place Constants on Left Side of Comparisons

Consider the code examples below. They are both equivalent, at least on first inspection. However, example 1 compiles and example 2 does not. With a closer look you'll see that the second statement isn't doing a comparison its doing an assignment. Example 2 tries to make an assignment to the constant value 0, and thus fails to compile. Assignments vs. comparison mistakes can be difficult to find in your code. But, by placing the constant on the left side you can let the compiler do the work for you.

```
1.    if (something == 1) { ... }
      if (x = 0) { ... }
2.    if (1 == something) { ... }
      if (0 = x) { ... }
```

3. Optimization vs Abstraction

Code in a two pass mode. First, implement with good OO abstractions and a well though out design. Second, when integrating your class into the its framework or application, measure performance and seek out the bottlenecks. Then optimize the bottlenecks.

❖ Javadoc Conventions

Included in sun's JDK is a documenting tool called javadoc that processes Java source code files and produces external documentation in the form of html files. Javadoc is a great utility, but it does have limitations. Javadoc supports a limited number of tags -- learn them and use them well. The following is a general guideline for commenting your code to support javadoc.

i. Overall Guidelines

- Document What code does as well as why it was developed. For example, youcan look at a piece of code, or class and figure out what it does internally. However, it may not exactly be apparent what requirements, or other systems this class supports or, why the code is trapping for and throwing certain exceptions.
- Document difficult or complex functionality
- Document dependencies, if methods call other methods internally it are important to note that in the method description.
- Document members of the Domain Object Model that your class is based on, or inspired by.
- Methods that implement an interface should not provide javadoc comments, since the javadoc comments are included in the interface and will be referenced when javadoc executes.

ii. Standard .java file header

Every .java source file should include the standard iwombat header template and detail out the items as appropriate.

iii. Method Documentation

Each method should include the `@exception`, `@param`, and `@return` javadoc tags where appropriate.

Acceptable:

```
/**
* Method to check if proscribed operation is allowed for this object.
* This method is needed to provided some level of security on operations.
*
* @param action must be an operation that has registered itself with the
object
* @return boolean true if the operation is allowed, false otherwise.
* @exception UnknownOperation exception is thrown when an operation that
```

```
has not
* registered with the object is passed as a parameter.*/

public boolean operationIsAllowed(Operation action) throws
UnknownOperation{}
```

Unacceptable:

```
/**
* operation check takes an operation and returns true or false */

public boolean operationIsAllowed(Operation action)throws
UnknownOperation{}
```

iv. Use the `@deprecated` Tag

In general it is not a good idea to remove methods from a class (or classes from a package), instead label old methods with the `@deprecated` tag. With liberal use of this tag you are less likely to break builds and code in use elsewhere. However, the compiler will produce warnings letting other developers know that they are using a deprecated method (or class).

Acceptable:

```
/**
* Method to check if proscribed operation is allowed for this object.
* This method is needed to provided some level of security on operations.
* @param Operation must be an operation that has registered itself with the
object
* @return boolean true if the operation is allowed, false otherwise.
* @exception UnknownOperation exception is thrown when an operation that
has not
* registered with the object is passed as a parameter.
* @deprecated No longer used, SecurityAccessor class in
com.iwombat.security replaces functionality
* @see com.asta.security */

public boolean operationIsAllowed(Operation action)
throws UnknownOperation {}
```

Unacceptable:

```
/**
* operation check takes an operation and returns true or false
* no longer needed use SecurityAccessor */

public boolean operationIsAllowed(Operation action) throws
UnknownOperation{}
```