Java Coding Standards

Created by

UPTOWN IT

January 2022

Contents

Naming Conventions

Names should properly describe objects

First of all, all variables and methods should have names that describe what they are. If you want a variable that holds the user's ID, you want the name to reflect that. So instead of something like "UI" – which is short for User Interface as well as User ID – you would use a name like "User ID number".

Variables almost always describe things, whether they are physical entities or ID numbers, so should often be nouns. Examples: "UserIDNumber", "UserName", "EnteredPassword", and so on.

Methods almost always describe actions, so should often be verbs, or action words. Some examples: ValidatePassword(), GetUserName(), UpdateScore().
In Java, as in programming in general, variable names and method names can't have spaces in them. This leads into the next point.

Combining multiple words

Because variable and method names can't have spaces in them, and we're trying to have more descriptive names, often you need a way to combine multiple words together. If you have a variable that is supposed to hold the user's ID number, you want the name to combine "user" and "ID number" together somehow.

Some common ways of combining multiple words are:

- **Pascal case**: Where every word in a variable or method name starts with a capital letter. For example, for a variable that holds the user's ID number: UserIdNumber.
- **Camel case**: Almost the same as Pascal case, except the first letter is in lower case. For example, "userIdNumber." This is called camel case because the capital letters look a bit like the humps on a camel's back.
- **Snake case**: Not as common as the above two. In this convention, every word is separated by a _ underscore character. For example, "user_ID_number."

Once you have decided on a naming convention, it is then important to stick to it. If you are working in a team, then the team will have to decide on what conventions to follow ahead of time, and then everyone should stick to these.

Organisational standards and naming conventions

Sometimes an organisation or team decides on some extra conventions to follow with variable and method names.

One common convention is with controls on a form. Variable names for text boxes should start with “txt”, variable names for labels should start with “lbl”, and buttons should start with “btn”, and so on.

Another example of an organisational standard, but much less common, is to start a variable name with some prefixes indicating extra information about that variable. Such as, “sopUserID”. The first “s” means it is a string, and the “op” means it is an output parameter. This type of convention is an older convention; modern IDEs mean this is not as necessary these days.

Where possible, avoid using the same name

Take a look at the following code:

```
public class student
{
   private String name;

   public void name(String name) {
      name = name;
   }

}
```

There are three things in this class that all have the name “name”. The private property is called “name”, the method is called “name”, and the parameter passed to the method is also called “name”. As they are all called the same, this can get very confusing. But Java still allows it.

Choose names, and follow naming conventions, to avoid this sort of situation. Let’s look at the above example again, but with some different names.

```
public class student
{
   private String name;

   public void getName(String inName) {
      name = inName;
   }
}
```

The three things are now called name, getName(), and inName, and so it is much easier to differentiate them.

Common Java naming conventions

Here are some commonly used conventions in Java programming:

- Package names start in lower case. For example, java.util.
- Class and Interface names are in Pascal case. (That is, they start with an uppercase letter.)
- Methods, property names, and variable names, are in camel case. For example, userName and getUserName().
- Constants are all in uppercase, and use snake case (the use of _ underscores) to separate words. For example, MINIMUM_VALUE, MAXIMUM_VALUE, GRAVITY.

Class members and properties should be private

In Java, the convention is that all class properties should be marked as private, and should only be accessed and modified through getter and setter methods.
For example:

Not this	But this

public class student
{
  public String name;
  public String ID;
}
	public class student
	{
	  private String name;
	  private String ID;
	
	  public String getName() {
	  return name;
	  }
	
	  public void setName(String inName) {
	  name = inName;
	  }
	
	  public String getID {
	  return ID;
	  }
	
	  public void setID(String inID) {
	  ID = inID;
	  }
	
	}

It is more work to set up the class this way, but since other parts of the program can't modify the class properties directly, it keeps the class more secure. It also promotes *encapsulation*, or the idea that the class is the only bit of code that knows how its internals work.

Taking the above example a little bit further, the class might make a check in setName() to make sure a valid ID number has been entered.

```
public boolean isIDValid (String inID) {
   check if inID is a valid number
   if it is, return True
}

public void setID(String inID) {
   if (isIDValid(inID)) {
      ID = inID;
   }
}
```

If other code could access the ID property directly (that is, ID is public), then the validation check would be bypassed.

Use StringBuilder and StringBuffer for string manipulation

In Java it is possible to add strings together, like this.

```
String greeting = "Hello";
String audience = "world";
String message = greeting + ", " + audience + "!";

System.out.println(message);
```

Using + is inefficient, and creates extra string objects in memory. When doing significant amounts of string manipulation, it is preferred to use StringBuilder or StringBuffer.

The same example again, but with StringBuilder:

```
String greeting = "Hello";
String audience = "world";

StringBuilder sBuilder = new StringBuilder();
sBuilder.append(greeting);
sBuilder.append(", ");
sBuilder.append(audience);
sBuilder.append("!");

System.out.println(sBuilder);
```

Use of variables

Declare and create a variable close to where you will use it

Where possible, create a variable close to where it is going to be used.
For example, rather than creating all of a method's variables at the top of the method, create them where you need them.

The following example shows the difference. In the left-hand example, to see what's happening with sBuilder2, you have to look all the way back to the beginning of the code. In the right-hand example, you do not have to go as far to see what the variables are.

Creating all variables at once	Creating variables where They are used

`String greeting = "Hello";`
`String audience = "world";`
`String job = "guide";`
`String dayOfWork = "today";`
 	
`StringBuilder sBuilder1 =`
`  new StringBuilder();`
 	
`StringBuilder sBuilder2 =`
`  new StringBuilder();`
 	
`sBuilder1.append(greeting);`
`sBuilder1.append(", ");`
`sBuilder1.append(audience);`
`sBuilder1.append("!");`
 	
`sBuilder2.append("I am your ");`
	`String greeting = "Hello";`
	`String audience = "world";`
	
	`StringBuilder helloLine1 =`
	`  new StringBuilder();`
	
	`helloLine1.append(greeting);`
	`helloLine1.append(", ");`
	`helloLine1.append(audience);`
	`helloLine1.append("!");`
	
	`String job = "guide";`
	`String dayOfWork = "today";`
	
	`StringBuilder helloLine2 =`
	`  new StringBuilder();`

```
sBuilder2.append(job);
sBuilder2.append(" ");
sBuilder2.append(dayOfWork);
sBuilder2.append(".");

System.out.println(sBuilder1);
System.out.println(sBuilder2);
```

```
helloLine2.append("I am your ");
helloLine2.append(job);
helloLine2.append(" ");
helloLine2.append(dayOfWork);
helloLine2.append(".");

System.out.println(helloLine1);
System.out.println(helloLine2);
```

Do not re-use variables

Only use a variable for its specified purpose, and do not suddenly use it for something else. Once upon a time, when computers didn't have as much disk space and memory, you might re-use variables to save on space. Modern computers do not have to worry about this restriction, and you can create additional variables as you need them.

Here's an example of what re-using variables looks like, and an example of how to avoid it.

Re-using variables

```
String s1 = "Hello";
String s2 = "world";

StringBuilder sBuilder =
   new StringBuilder();

sBuilder.append(s1);
sBuilder.append(", ");
sBuilder.append(s2);
sBuilder.append("!");

System.out.println(sBuilder);

s1 = "guide";
s2 = "today";

// Clear the string builder
sBuilder.setLength(0);

sBuilder.append("I am your ");
sBuilder.append(s1);
sBuilder.append(" ");
sBuilder.append(s2);
sBuilder.append(".");
```

Using variables for their intended purpose

```
String greeting = "Hello";
String audience = "world";

StringBuilder helloLine1 =
   new StringBuilder();

helloLine1.append(greeting);
helloLine1.append(", ");
helloLine1.append(audience);
helloLine1.append("!");

job = "guide";
dayOfWork = "today";

StringBuilder helloLine2 =
   new StringBuilder();

helloLine2.append("I am your ");
helloLine2.append(guide);
helloLine2.append(" ");
helloLine2.append(dayOfWork);
helloLine2.append(".");

System.out.println(helloLine1);
```

`System.out.println(sBuilder);`	`System.out.println(helloLine2);`

In the example on the left, s1, s2, and sBuilder get re-used and it is hard to tell what they mean. You also have to use a bit of a hack to clear sBuilder, and this hack needs a comment. In the example on the right, it is clearer what each variable means. Also, we do not need to use a hack to clear a stringBuilder and thus we do not need the comment.

Counter variables in for loops

The convention is that in a “for loop”, variables such as “i”, “j”, and “k” are used as counter variables. Sometimes x, y, and z get used, when the loop is to do with maths, geometry, or graphics. This convention comes from math, long before modern computer programming. So while these names follow none of the naming conventions mentioned above, these are the usual variable names used in for loops.

The letter “**i**” is the default counter variable for loops. When a second, or third for loop is used inside another for loop, j and then k are used.
For example:

```
for (int i = 0; i < 10; i++) {
   for (int j = i; j < 10; j++) {

      System.out.println("i is " + i + " and j is " + j);
   }
}
```

Loops and Iteration

The enhanced for loop

Java’s enhanced for loop is another way to go through all the items in a collection, or array. This is a bit easier to use and does not require a counter variable.

When you want to go through the items in an array, or collection, the usual Java convention is to use the enhanced for loop where possible. It is shorter to type and when you use it, it is clear that you’re going through the items in an array. This might sometimes be referred as Java’s version of the “for each” loop.

Here’s an example. The usual for loop is on the left, and the enhanced for loop is on the right.

Traditional for loop	Enhanced for loop

String[] words =
  new String[]{"Hello", ", ", "World", "!"};
 	
for (int i = 0; i < words.length; i++)
{
  System.out.print(words[i]);
}
System.out.println();
	String[] words =
	  new String[]{"Hello", ", ", "World", "!"};
	
	for (String currentWord : words)
	{
	  System.out.print(currentWord);
	}
	System.out.println();

In the “for loop” one on the left we need a counter variable and use the traditional counter variable “*i*”. In the enhanced for loop on the right, no counter variable is needed, and by using this type of for loop it is also clear that the loop is being used to go through everything in the *words* array.

Exception handling

Do not have empty catch blocks

Try catch blocks are used to prevent errors crashing the program. But they shouldn’t be used to just prevent errors causing problems, but also to handle the error in some way.

The catch block in following example is empty. The code does not handle any errors, it just ignores errors and continues. It gives the impression that everything is OK, when in fact an error just occurred.

Do not do this

```
try {
   // Do something
}
catch {
   // Even if an error occurs, do nothing
   // we want to just continue on.
}
```

Always make sure to handle the error. It could be as simple as a message to the user to try again, or it could be an error logged in several different log files.

Always clean up resources after a try catch block

Try catch blocks are used to execute code that might run into problems, because it is dealing with unreliable external resources such as printers, files, or user input.

Here's an example, to do with file input. In Java, to read from a file, first it must be opened. Once finished, the file has to be closed.

```
Open file

Read from file

Close file
```

However, that "read from file" might cause a problem, so try catch blocks are often used. But when using a try catch block, you want to make sure that the file is closed. Here's an example of doing that, with the "finally" part of a try catch block.

```
try {
   Open file
   Read from file
}
catch {
   Deal with the error
}
finally {
   Close file
}
```

The "finally" occurs regardless of whether an error occurred or not.

Use of Comments

Do not use comments to explain obscure code

If you have written some extremely complex code, do not try to get away with it by putting a comment above the code, explaining how it works. If you have to do this, this means you probably need to rewrite the code so that it is clearer what the code does.

Occasionally this isn't possible – there might be a block of code that does something extremely important, but has become extremely complex over time and nobody on the team wants to deal with it. This is where you definitely need a comment to explain it, or perhaps a comment saying, "Do not modify the following code ever. It works, but we do not know how."

Starting comment at the beginning of the program

All files should start with a comment that lists things like the class (or file) name, a description of the file, version information, date, author, and copyright information.

Here's an example:

```
/*
 * Class, or file name
 *
 * This class does this.
 *
 * Version information
 * Date
 * Author
 *
 */
```

Use of Javadoc

Put a Javadoc comment before all classes, and all of a class's non-simple public methods. They start with /**.

Have a look at the following example:

```
/*
 * The Student class is holds identification and administrative
information about a student.
 */
public class Student
{
   private String _name;

   public void GetName(String inName) {
      _name = inName;
   }

   /**
    * Validates a given ID value and returns true if valid.
    */
   public static boolean isIDValid (String inID) {
      check if inID is a valid number
      if it is, return True
   }
}
```

The class gets a JavaDoc comment and the isIDValid() method gets a JavaDoc comment. The GetName() method does not, as it is a very simple and typical getter method.