|  |  |
| --- | --- |
| [chicken.gif (7083 bytes)](https://javaranch.com/style.jsp) | coophead.gif (2368 bytes)  **Java Programming Style Guide** |

|  |
| --- |
| [Introduction](https://javaranch.com/styleLong.jsp#intro)  [1 Formatting](https://javaranch.com/styleLong.jsp#format)  [1.1 Indentation](https://javaranch.com/styleLong.jsp#indent) [1.2 Spacing](https://javaranch.com/styleLong.jsp#spacing) [1.3 Class Member Ordering](https://javaranch.com/styleLong.jsp#order) [1.4 Maximum Line Length](https://javaranch.com/styleLong.jsp#length) [1.5 Parentheses](https://javaranch.com/styleLong.jsp#paren)  [2 Identifiers](https://javaranch.com/styleLong.jsp#identifiers)  [2.1 Classes and Interfaces](https://javaranch.com/styleLong.jsp#class) [2.2 Packages](https://javaranch.com/styleLong.jsp#pack) [2.3 All Other Identifiers](https://javaranch.com/styleLong.jsp#other)  [2.3.1 Hungarian Notation and Scope Identification](https://javaranch.com/styleLong.jsp#hungarian) [2.3.2 Test Code](https://javaranch.com/styleLong.jsp#test_code)  [3 Coding](https://javaranch.com/styleLong.jsp#code)  [3.1 Constructs to Avoid](https://javaranch.com/styleLong.jsp#avoid)  [3.1.1 do..while](https://javaranch.com/styleLong.jsp#do) [3.1.2 return in the Middle of a Method](https://javaranch.com/styleLong.jsp#return) [3.1.3 continue](https://javaranch.com/styleLong.jsp#cont) [3.1.4 break Other Than in a Switch Statement](https://javaranch.com/styleLong.jsp#break)  [3.2 Do Not Compound Increment or Decrement Operators](https://javaranch.com/styleLong.jsp#inc) [3.3 Initialization](https://javaranch.com/styleLong.jsp#init) [3.4 Access](https://javaranch.com/styleLong.jsp#access)  [4 Self-Documenting Code](https://javaranch.com/styleLong.jsp#doc) |

|  |
| --- |
| **Introduction**  The Java language gives you all the room you need to write code that would be very difficult for others to understand. Java also permits you to write code that is very easy to understand. Most development teams would prefer the latter.  A style guide provides provides a map so that the code generated by a group of programmers will be consistent and, therefore, easier to read and maintain. Many people do not care for the [style guide](http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html) offered by Sun. This document is one alternative.  This document covers most areas where there could be confusion or difference of opinion. Areas that have never been a problem in our experience are undocumented, but we try to provide at least one example to demonstrate proper use somewhere in the document.  **1 - Formatting**  **1.1 - Indentation**  **All indenting is done with spaces, not tabs. All indents are four spaces.**  *Reasoning: All programs work well with spaces. Most programs will mix tabs and spaces so that some lines are indented with spaces and some with tabs. If your tabbing is set to 4 and you share a file with someone that has tabbing set to 8, everything comes out goofy.*  **Matching braces always line up vertically in the same column as their construct.** example:  void foo()  {  while (bar > 0)  {  System.out.println();  bar--;  }  if (oatmeal == tasty)  {  System.out.println("Oatmeal is good and good for you");  }  else if (oatmeal == yak)  {  System.out.println("Oatmeal tastes like sawdust");  }  else  {  System.out.println("tell me pleeze what iz dis 'oatmeal'");  }  switch (suckFactor)  {  case 1:  System.out.println("This sucks");  break;  case 2:  System.out.println("This really sucks");  break;  case 3:  System.out.println("This seriously sucks");  break;  default:  System.out.println("whatever");  break;  }  }  **All if, while and for statements must use braces even if they control just one statement.**  *Reasoning: Consistency is easier to read. Plus, less editing is involved if lines of code are added or removed.*  if (superHero == theTick) System.out.println("Spoon!"); // NO!  if (superHero == theTick)  System.out.println("Spoon!"); // NO!  if (superHero == theTick) {  System.out.println("Spoon!");  } // NO!  if (superHero == theTick)  {  System.out.println("Spoon!");  } // YES!  **1.2 - Spacing**  **All method names should be immediately followed by a left parenthesis.**  foo (i, j); // NO!  foo(i, j); // YES!  **All array dereferences should be immediately followed by a left square bracket.**  args [0]; // NO!  args[0]; // YES!  **Binary operators should have a space on either side.**  a=b+c; // NO!  a = b+c; // NO!  a=b + c; // NO!  a = b + c; // YES!  z = 2\*x + 3\*y; // NO!  z = 2 \* x + 3 \* y; // YES!  z = (2 \* x) + (3 \* y); // YES!  **Unary operators should be immediately preceded or followed by their operand.**  count ++; // NO!  count++; // YES!  i --; // NO!  i--; // YES!  **Commas and semicolons are always followed by whitespace.**  for (int i = 0;i < 10;i++) // NO!  for (int i = 0; i < 10; i++) // YES!  getPancakes(syrupQuantity,butterQuantity); // NO!  getPancakes(syrupQuantity, butterQuantity); // YES!  **All casts should be written with no spaces.**  (MyClass) v.get(3); // NO!  ( MyClass )v.get(3); // NO!  (MyClass)v.get(3); // YES!  **The keywords if, while, for, switch, and catch must be followed by a space.**  if(hungry) // NO!  if (hungry) // YES!  while(pancakes < 7) // NO!  while (pancakes < 7) // YES!  for(int i = 0; i < 10; i++) // NO!  for (int i = 0; i < 10; i++) // YES!    catch(TooManyPancakesException e) // NO!  catch (TooManyPancakesException e) // YES!  **1.3 - Class Member Ordering**  class Order  {  // fields (attributes)  // constructors  // methods  }  **1.4 - Maximum Line Length**  **Avoid making lines longer than 120 characters. If your code starts to get indented way to the right, consider breaking your code into more methods.**  *Reasoning: Editors and printing facilities used by most programmers can easily handle 120 characters. Longer lines can be frustrating to work with.*  **1.5 - Parentheses**  **Parentheses should be used in expressions not only to specify order of precedence, but also to help simplify the expression. When in doubt, parenthesize.**  **2 - Identifiers**  **All identifiers use letters ('A' through 'Z' and 'a' through 'z') and numbers ('0' through '9') only. No underscores, dollar signs or non-ascii characters.**  **2.1 - Classes and Interfaces**  **All class and interface identifiers will use mixed case. The first letter of each word in the name will be uppercase, including the first letter of the name. All other letters will be in lowercase, except in the case of an acronym, which will be all upper case.**  Examples:  Customer SalesOrder TargetURL URLTarget  **2.2 - Packages**  **Package names will use lower case characters only. Try to keep the length under eight (8) characters. Multi-word package names should be avoided.**  Examples:  common core lang  **2.3 - All Other Identifiers**  **All other identifiers, including (but not limited to) fields, local variables, methods and parameters, will use the following naming convention. This includes identifiers for constants.**  *Reasoning: Using all upper case, as traditionally done in C, is a violation of OO abstraction. For example, a variable which starts out as a constant may be refactored later to not be a constant.*  **The first letter of each word in the name will be uppercase, except for the first letter of the name. All other letters will be in lowercase, except in the case of an embedded acronym, which will be all uppercase. Leading acronyms are all lower case.**  Examples:  customer salesOrder addToTotal() targetURL urlTarget  **2.3.1 Hungarian Notation and Scope Identification**  [**Hungarian notation**](http://en.wikipedia.org/wiki/Hungarian_notation)**and scope identification are not allowed.**  *Reasoning: Hungarian Notation, which specifies type as part of the identifier, violates OO abstraction. Scope identification specifies scope as part of the identifier, which also violates OO abstraction.*  public class Person  {  private String sFirstName; // NO! (Hungarian notation: s for String)  private String firstName; // YES!  private String mLastName; // NO! (Scope identification: m for member variable)  private String \_lastName; // NO! (Scope identification: \_ for member variable)  private String lastName; // YES!  // ...  }  **2.3.2 Test Code**  **Test code is permitted to use underscores in identifiers for methods and fields.**  *Code for testing other code often needs to be able to refer to existing identifiers, but also be able to append other qualifiers to the name. This is easier to read if an additional delimiter is allowed.*  Example:  Code to test a method double eatSomePie(double amount) may use variables such as:  eatSomePie\_count eatSomePie\_amount eatSomePie\_return eatSomePie\_exception  **3 - Coding**  **3.1 - Constructs to Avoid**  **3.1.1 - Never use do..while**  **Do not use do..while loops.**  *Reasoning: Consider that the programmer looking at your code is probably examining each method starting at the top and working down. When encountering a loop, the first thing the programmer wants to know is what terminates the loop. If you have that logic at the bottom, it is harder to read. Further, many less experienced programmers are not familiar with do..while, but may be required to modify your code.*  So rather than:  boolean done = false;  do  {  ...  } while (!done)  use:  boolean done = false;  while (!done)  {  ...  }  **3.1.2 - Never use return in the middle of a method**  **return is to be used at the end of a method only.**  *Reasoning: Using return in the middle of a method makes it difficult to later break the method into smaller methods. It also forces the developer to consider more than one exit point to a method.*  **3.1.3 - Never use continue**  **Never use continue.**  *Reasoning: Using continue makes it difficult to later break the construct into smaller constructs or methods. It also forces the developer to consider more than one end point for a construct.*  **3.1.4 - Never use break other than in a switch statement**  **break is used only for switch statement control.**  *Reasoning: Using break, other than for switch statement control, makes it difficult to later break a construct into smaller constructs or methods. It also forces the developer to consider more than one end point for a construct.*  **3.2 - Do Not Compound Increment or Decrement Operators**  **Use a separate line for an increment or decrement.**  *Reasoning: Compounding increment or decrement operators into method calls or math is not clear to less experienced programmers who may be required to modify your code.*  Examples:  foo(x++); // NO!  foo(x); // YES!  x++;  y += 100 \* x++; // NO!  y += 100 \* x; // YES!  x++;  Note: i++ and ++i are equally fast, and i++ seems more consistent with the rest of the language. Since the above prevents any use of a case where ++i could make a difference, never use pre- increment/decrement.  **3.3 - Initialization**  **Declare variables as close as possible to where they are used.**   Examples:  int totalWide;  int firstWide = 20;  int secondWide = 12;  firstWide = doFoo(firstWide, secondWide);  doBar(firstWide, secondWide);  totalWide = firstWide + secondWide; // wrong!  int firstWide = 20;  int secondWide = 12;  firstWide = doFoo(firstWide, secondWide);  doBar(firstWide, secondWide);  int totalWide = firstWide + secondWide; // right!  int secondWide = 12;  int firstWide = doFoo(20, secondWide);  doBar(firstWide, secondWide);  int totalWide = firstWide + secondWide; // even better!  **3.4 - Access**  **All fields must be private, except for some constants.**  **4 - Self-Documenting Code**  *"Any fool can write code that a computer can understand.*  *Good programmers write code that humans can understand."*  --- Martin Fowler, Refactoring: Improving the Design of Existing Code  **Rather than trying to document how you perform a complex algorithm, try to make the algorithm easier to read by introducing more identifiers. This helps in the future in case the algorithm changes but someone forgets to change the documentation.**  Example:  Instead of:  if ( (hero == theTick) && ( (sidekick == arthur) || (sidekick == speak) ) )  Use:  boolean isTickSidekick = ( (sidekick == arthur) || (sidekick == speak) );  if ( (hero == theTick) && isTickSidekick )  Or:  Instead of:  public static void happyBirthday(int age)  {  // If you're in the US, some birthdays are special:  // 16 (sweet sixteen)  // 21 (age of majority)  // 25, 50, 75 (quarter centuries)  // 30, 40, 50, ... etc (decades)  if ((age == 16) || (age == 21) || ((age > 21) && (((age % 10) == 0) || ((age % 25) == 0))))  {  System.out.println("Super special party, this year!");  }  else  {  System.out.println("One year older. Again.");  }  }  Use:  public static void happyBirthday(int age)  {  boolean sweet\_sixteen = (age == 16);  boolean majority = (age == 21);  boolean adult = (age > 21);  boolean decade = (age % 10) == 0;  boolean quarter = (age % 25) == 0;  if (sweet\_sixteen || majority || (adult && (decade || quarter)))  {  System.out.println("Super special party, this year!");  }  else  {  System.out.println("One year older. Again.");  }  } |