JAVA

BEST PRACTICES TO PROGRAMMING CODE WITH JAVA

Learn to Write Effective Programming Code in Java!

CHARLIE MASTERSON

Java:

Best Practices to Programming Code with Java

Charlie Masterson

Table of Contents

Introduction

I want to thank you and congratulate you for reading mybook, “ *Java: Best Practices to Programming Code with Java*”.

This book contains proven steps and strategies on how to write Java code that doesn ’ t kick up errors, is neat and tidy and has a high level of readability. If your code looks familiar, it will be far easier to understand and even easier to cast your eye over to see if there are any obvious errors. To make sure your code looks familiar, there are a set of best practice guidelines that you should follow; not only will this help you, it will help other developers who look over your code as well.

This book assumes that you already have a level of familiarity and experience with Java computer programming language. My aim is to help you further your knowledge and have you programming like a pro in no time at all. If you are a completely new to the language, please familiarize yourself with the basics before you run through my best practice guide.

Thanks again for purchasing this book, I hope you enjoy it!

Chapter 1: Formatting Your Code

Really and truthfully, the first place to start, before we get into the nitty-gritty, is in how to format your Java code. There are several “rules” that you should follow to make sure that your code is readable and clean:

Indentation
All Java code indenting uses spaces rather than tabs and each indent is 4 spaces. The reason for this is because, like all other computer programming languages, Java works well with spaces. Some programs mix up spaces and tabs, leaving some lines indented with tabs and some with spaces. If you set your tabs to 4 and your file is shared with a person that has theirs set to 8 it is all going to look a mess.

When you use matching braces, they must vertically line up under their construct, in exactly the same column. For example:

```
void foo()
{
    while (bar > 0)
    {
        System.out.println();
        bar--;
    }

    if (Cheese == tasty)
    {
        System.out.println("Cheese is good and it is good for you");
    }
    else if (Cheese == yuck)
    {
        System.out.println("Cheese tastes like rubber");
    }
    else
    {
        System.out.println("please tell me, what is this cheesel'");
    }

    switch (yuckyFactor)
    {
        case 1:
            System.out.println("This is yucky");
            break;
        case 2:
            System.out.println("This is really yucky");
            break;
        case 3:
            System.out.println("This is seriously yucky");
            break;
        default:
            System.out.println("whatever");
            break;
    }
}
```

Use Braces for All if, for and while Statements

This is the case even if they are controlling just a single statement. Why? Because when you keep your formatting consistent, your code becomes much easier to read. Not only that, if you need to add or take away lines of code, there isn't so much editing to do. Have a look at this example – the first three show you the wrong way while the fourth shows the right way to do it:

Bad Examples:

```
if (superHero == theTock) System.out.println("Fork!");

if (superHero == theTock)
     System.out.println("Fork!");

 if (superHero == theTock) {
      System.out.println("Fork!");
    }
```

Good Example:

```
    if (superHero == thetock)
    {
        System.out.println("Fork!");
    }
```

Spacing

Whenever you have a method name, it should be followed immediately with a left parenthesis:

Bad Example:

```
foo (i, j);
```

Good Example:

```
foo(i, j);
```

A left square bracket must follow immediately after any array dereference:

Bad Example:

```
     args [0];
```

Good Example:

```
    args[0];
```

Make sure there is a space on either side of any binary operator:

Bad Examples:

```
     a=b+c;
    a = b+c;
    a=b + c;
```

Good Example:

```
    a = b + c;
```

Bad Example:

```
    z = 2*x + 3*y;
```

Good Examples:

```
    z = 2 * x + 3 * y;      !
    z = (2 * x) + (3 * y);!
```

Whenever you use a unary operator it should be followed immediately by its operand:

Bad Example:

```
    count ++;
```

Good Example:

```
    count++;
```

Bad Example:

```
    i --;
```

Good Example:

```
i--;
```

Whenever you use a semicolon or a comma, follow it immediately with a whitespace:

Bad Example:

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

Good Example:

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

Bad Example:

```
getOmelets(EggsQuantity,butterQuantity);
```

Good Example:

```
getOmelets(EggsQuantity, butterQuantity);
```

Write all casts without spaces

Bad Examples:

```
 (MyClass) v.get(3);
( MyClass )v.get(3);
```

Good Example:

```
(MyClass)v.get(3);
```

When you use the keywords while, if, for, catch and switch , always follow them with a space:

Bad Example:

```
if(hungry)
```

Good Example:

```
if (hungry)
```

Bad Example:

```
while(omelets < 7)
```

Good Example:

```
while (omelets < 7)
```

Bad Example:

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

Good Example:

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

Bad Example:

```
catch(TooManyOmeletsException e)
```

Good Example:

```
 catch (TooManyOmeletsException e)
```

Ordering of Class Members

When you use class members, they must always be ordered in the following way, without exception:

```
class Order
{
    // fields (attributes)

    // constructors

    // methods
}
```

Line Length

Try not to make any lines more than 120 characters long. This is because most text editors can handle this line length easily; any more than this and things start to get a little messy and frustrating. If you find that your code is being indented way off to the right, think about breaking it done into a few more methods.

Parentheses

When you use parentheses in expressions, don’t just use them for the precedence order; they should also be used as a way of simplifying things

Chapter 2: Naming Conventions

Naming methods, variables, and classes correctly are very important in Java and here are some of the more common conventions that, if you learn and follow, will help your code to be much clearer:

- When you are naming packages, the name should be lowercase, for example, mypackage

- All names that are representative of types have to be nouns and they are written in mixed case, beginning with an upper-case letter, for example, AudioSystem, Line

- All variable names have to be in a mixed case and start with a lower-case letter, for example, audioSystem, line . This makes it easier to distinguish the variables from types and also resolves any potential naming clashes.

- Where a name is representative of a constant, or a final variable, they must be in upper case and an underscore should be used separate each word, for example, COLOR_BLUE, SAM_ITERATIONS . Generally, use of these constants should be kept to a minimum and, more often than not, it is better to implement a value as a method, for example

```
Int getSamIterations() // NOT: SAM_ITERATIONS = 25
  {
   return 25;
  }
```

This is much easier to read and it also ensures that class values have a uniform interface

- If a name is representing a method it must be a verb and it must be mixed case with a lower-case at the beginning, for example, computeTotalLenght(), getNames() . This is exactly the same as for variables but, because they have a specific form, the Java method is already easily distinguished from the variable.

- Never use uppercase for acronyms or abbreviations when you use them as a name, for example,

Good Example:
```
exportHtmlSource();
```

Bad Example:
```
exportHTMLSource();
```

Good Example:
```
openDvdPlayer();
```

Bad Example:
```
openDVDPlayer();
```

The reason for this is because if you use all upper-case letters for a base name you will see clashes with the naming conventions we already talked about in this chapter. For example, if you had this type of variable, you would need to call it hTML or dVD and these, as you can see, are not incredibly readable. A separate problem is this - when you connect the name to another, readability is significantly compromised and the word that comes after the acronym doesn't stand out.

- All private class variables should have an underscore as a suffix, for example:

```
class Person
{
  private String name_;
  ...
}
```

Quite apart from the type and name, the most important feature of any variable is its scope. Using the underscore to indicate the scope of a class allows you to easily distinguish a class variable from a local scratch variable. The reason why this is so important is because the class variable is considered to be of higher importance than the method variable and, as such, you should treat it carefully.

One side effect of using the underscore convention is that it solves the issue of finding good variable names for the setter methods. For example:

```
  void setName(String name)
  {
    name_ = name;
  }
```

There is an issue about whether the underscore should be a suffix or whether it should be a prefix. Both of these are used commonly but adding it as a suffix is recommended because it is the best way to preserve name readability.

Do note that the scope identification of a variable has been something of a controversial issue for some time now but, at long last, the practice of using the underscore suffix is now gaining momentum and is more accepted within the professional Java development world

- All generic variables must be named the same as their type. For example:

Good Example:

```
void getTopic(Topic topic)
```

Bad Examples:

```
void getTopic(Topic value)
void getTopic(Topic aTopic)
void getTopic(Topic t)
```

Good Example:

```
void connect(Database database)
```

Bad Examples:

```
void connect(Database db)
void connect(Database oracleDB)
```

This is good practice because it helps you to cut down on intricacy by reducing the number of names and terms used. It also makes it much easier to decide what type a variable is just by the name. If this convention doesn't fit in with your programming, it is a good indicator that you have chosen a poor type name.

- All non-generic variables are assigned a specific role and you can often name them with a combination of the type and the role. For example:

```
Point  startingPoint, centerPoint;
  Name   loginName;
```

- Use the English language for all names. This is the preferred language and the only reason you can deviate from this is if you are 100% certain that your program is never going to be seen or read by anyone who does NOT speak or read your language.
- Scope and names should match. If you have a variable that has a large scope, you can give it a longer name and, in direct contrast, variables that have a smaller scope can be given a shorter name.

If you use a scratch variable for indices or as temporary storage, do keep them short. A programmer who is reading your code and sees these variables has to be able to assume that the value of the variable not been used outside of a small number of code lines. Some of the scratch variables used for integers are `i, j, k, m, n` and commonly used for integers are `c` and `d`.

- Object names are implicit and should not be used in method names. For example:

Good Example:

```
line.getLength();
```

Bad Example:

```
line.getLineLength();
```

While the last example might look natural when it comes to class declarations, it does prove to be unnecessary, as you can see in the example.

Specific Naming Conventions

- Where an attribute is to be directly accessed, you must use the terms get and set. For example:

```
employee.getName();
employee.setName(name);

matrix.getElement(2, 4);
matrix.setElement(2, 4, value);
```

- Use the is prefix for all Boolean methods and variable. For example:

```
isSet
isVisible
isFinished
isFound
isOpen
```

Using this prefix will solve the common issue of selecting bad names for Booleans like flag or status. isStatus and isFlag just don't work and you will be forced to choose names that are more appropriate. Setter methods used for the Boolean variables should have set prefixes, as in:

```
void setFound(boolean isFound);
```

There are some alternatives that you can use instead of the prefix is and some of these work better in certain situations. There are three other prefixes you can use – has, should and can:

```
boolean hasLicense();
boolean canEvaluate();
boolean shouldAbort = false;
```

- When the name is representative of an object collection, use the plural form:

```
Collection<Point>  points;
int[]              values;
```

This immediately makes the code more readable because the user can see the variable type and the operations that are to be performed on the elements.

Chapter 3: Commenting

Comments are incredibly useful and are used to annotate programs. These comments are there for the reader of a program, to help them to understand your program or elements of it. As a rule of thumb, the code is used to tell the computer and the computer programmer what is to be done and the comments tell the programmer why it is being done.

Comments can go anywhere in a program where whitespace can go and the Java compiler will not take any notice of them – provided they have been written correctly of course. If you don't write the comments in the correct format, not only will your program throw up errors, your code is going to look very messy and confusing.

Let’s look at the different types of comments and how they should be written:

- **Line Comments -** When you write an end-of-line comment, you should start it with a pair of forward slashes (//) and it will finish at the end of the same line the forward slashes are on. The compiler will ignore anything that comes between // and the end of the line

- **Block Comments –** Block comments start with the forward slash, then the asterisk - /* and they end with the asterisk and the forward slash - */ e.g. /*this is a block comment*/ . Any text that appears between these two delimiters will be ignored, even if it goes over several lines.

- **Bold Comments -** These are special types of block comment that are designed with drawing attention in mind. For example:

```
/*-----------------------------------------------------
*  This is a block comment that will draw attention
*  to itself.
*-----------------------------------------------------*/
```

- **Javadoc Comments –** this is another special type of block comment that starts with a forward slash, followed by a pair of asterisks - /**. These tend to be used as a way of automatically generating a class API and the following are the general guidelines for writing a Javadoc comment:

There are no actual set-in-stone rules because a good programmer will write good code that will document itself

Do be sure that your comments agree with your code; if you ever update your code, you must remember to update the comments as well.

Never write a comment that does nothing more than repeat the code. Comments should describe what the code does and why, not how it is done:

```
g++;      //  increment g by one
```

If you have code that can be confusing, either comment it or, better still, rewrite it so it doesn’t look confusing.

At the start of each file, add a bold comment that contains your name, the date, what the program is for and how it should be executed:

```
/*-----------------------------------------------------------
*  Author:       Sharon Stone
*  Written:       5/12/2014
*  Last updated:  12/17/2006
*
*  Compilation:   javac HelloWorld.java
*  Execution:     java HelloWorld
*
*  Prints "Hello, World". This is the first program
* that everyone learns
*  % java HelloWorld
*  Hello, World
*
*-------------------------------------------------------------*/
```

Wherever you have a variable name, including an instance variable, that is important, comment it with the // comment and make sure that the comments are aligned vertically.

```
private double rx, ry;    //  position
private double q;         //  charge
```

Each method must be commented with a description of what the method does, including what it needs as an input, what the output will be and if there are any side effects – list them if there are. Your comment must include the parameter names.

```
/**
*   This method will use Knuth’s shuffling algorithm to
*   rearrange all the array elements in [] random order
 *
*   If it is null, it will throw a NullPointerException
*/
public static void shuffle(String[] a)
```

Chapter 4: Java Files

Another important part of Java that you have to get right for efficiency is how you use files. For a start, every Java file must have the .java extension. In this chapter, we are also going to take a brief look at the layout of statements, classes, methods, types, variables, and numbers. Here are some more things you must get right if you want your code to be effective:

- The content of each file must be kept to no more than 80 columns.

This is the common measure for printers, terminal emulators, editors, and debuggers. If your file is shared with other developers, you must keep to this constraint otherwise, readability will be low – when you pass a file to another developer that doesn't keep to this, there will be line breaks that you possibly didn't mean to be there.

- Avoid the use of special characters like page break and TAB

Special characters cause trouble for debuggers, editors, terminal emulators and printers when they are used in files that are shared across developers and across platforms.

- If a split line is not complete, it must be made obvious. For example:

```
totalSum = a + b + c +
           d + e;

method(param1, param2,
       param3);

setText ("Long line split" +
         "into two parts.");

for (int tableNo = 0; tableNo < nTables;
     tableNo += tableStep) {
  ...
}
```

Split lines happen when you go over the 80-column constraint and, although it isn't easy to give exact rules on how to split lines, the examples above should give you some idea. As a rule of thumb:

- A break should follow a comma

- A break should follow an operator

- New lines should be aligned with the start of the expression on the line before

Statements

Import and Package Statements

- Package statements are always the first statement in a file and every file must belong to a specified package

By placing all files into a specified package, and not the default Java package, allows the object-oriented programming of Java to be enforced.

- Import statements must be placed after a package statement. They should also be sorted so that the fundamental packages come first and then group all associated packages. There should be a single blank line between each group:

```
import java.io.IOException;
import java.net.URL;

import java.rmi.RmiServer;
import java.rmi.server.Server;

import javax.swing.JPanel;
import javax.swing.event.ActionEvent;

import org.linux.apache.server.SoapServer;
```

The location of the import statement will be enforced by Java. By sorting the packages, it makes it easier to browse through when there are a lot of imports and it also makes it easier to determine any dependencies on the present package. Grouping packages together collapses all related information into one unit and makes it less complicated. Less complicated = increased readability.

- Always explicitly list imported classes:

Good Example:

```
import java.util.List;
```

Bad Examples:

```
import java.util.*;
import java.util.ArrayList;
import java.util.HashSet;
```

When you explicitly import a class, you are providing the best documentation value for the present class and it also makes it much easier to understand the class and maintain it.

Interfaces and Classes

Make sure that all interface and class declarations are organized in the following way:

- Interface or class documentation

- Interface or class statement

- Static, or class, variables in this order – Public; Protected; Package (with no access modifiers; Private

- Instance variables in this order – Public; Protected; Package (with no access modifiers);

Private

- Constructors
- Methods but there is no particular order for these

Make the location of each of the class elements predictable so it reduces complexity.

Methods
Put method modifiers in this order:

- <access> static abstract synchronized <unusual> final native

The first modifier should always be the <access modifier> if there is one:

Good Example:
```
public static double square(double a);
```

Bad Example:
```
static public double square(double a);
```

The <access> modifier is public, private or protected while the <unusual> modifier will include transient and volatile. Out of all of this, the most important thing to learn is to put the <access> modifier at the beginning – this is a very important modifier and it has to stand out in your method declarations. The order of other modifiers isn't so important.

Types
- All type conversions should be explicitly done; you should never rely on implicit conversions:

Good Example:
```
floatValue = (int) intValue;
```

Bad Example:
```
floatValue = intValue;
```

By doing this, you are indicating that you fully aware of all the different types that are in use and that the mix was deliberate.

- Attach array specifiers to type, never to the variable:

Good Example:
```
int[] a = new int[20];
```

Bad Example:

```
int a[] = new int[20]
```

Arrayness is not a feature of the variable, and it is unknown why Java allows it to be when it should be attached to the base type

Variables

- When you declare a variable, it must be initialized and you should also use the smallest possible scope to declare as well.

This is to make sure that a variable will be valid all the time. On occasion, it may be possible to initialize the variable to a proper value in the location it is declared – in this case, leave the variable uninitialized, rather than initializing it to a false value

- Never let your variables have a dual meaning

This way, all of your concepts are uniquely represented and there is a lower chance of errors popping up

- Never declare a class variable publicly

One of the main concepts of Java is to hide and encapsulate information. The declaration of a class variable as public violates that so use access functions and private variables instead. There is just one exception to this – if a class is a data structure that has no behavior. In this case, you can make the instance variable public

- Brackets beside type when you declare an array:

Good Example:
```
double[] vertex;
```

Bad Example:
```
double vertex[];
```

Good Example:
```
int[]    count;
```

Bad Example:
```
int    count[];
```

Good Examples"
```
public static void main(String[] arguments)

public double[] computeVertex()
```

There are two reasons for this. First, as we mentioned earlier, arrayness is NOT a feature of the variable, it is attached to class. Second, when an array is returned from a method, you can't have the brackets anywhere but with type.

- Keep the lifespan of a variable as short as you can

This makes it easier to control the side effects and the effects of the variable.

Loops

- Only include loop control statements in the for() construction:

Good Example:

```
sum = 0;
```

Bad Examples:

```
for (i = 0, sum = 0; i < 100; i++)
for (i = 0; i < 100; i++)
sum += value[i];
sum += value[i];
```

This increases readability and makes it easier to maintain. Always have a clear distinction of what is in the loop and what controls it

- Do not initialize a loop variable right before a loop

Good Example:

```
isDone = false;
```

Bad Examples:

```
bool isDone = false;
while (!isDone) {        //      :
  :                      //      while (!isDone) {
}                        //        :
                         //      }
```

- Avoid using do-while loops

These are not as readable as the normal while and for loops because the conditional goes at the bottom. This means that a reader has to scan through the entire loop to understand what the scope is. You don’t actually need to use a do-while loop because they can be written very easily into standard while or for loops. Cutting down on the amount of constructs you use enhances readability.

- Avoid using continue and break in loops

The only reason you should use these is they provide better readability than their more structured equivalent

Conditionals

- Avoid the use of complex conditional expressions; a better option is a temporary

Boolean variable

Good Example:
```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) {
  :
}
```

Bad Example:
```
if ((elementNo < 0) || (elementNo > maxElement)||
     elementNo == lastElement) {
  :
}
```

When you assign a Boolean variable to an expression, your program is given automatic documentation, it will be better constructed, easier to read and to debug

- Place nominal case within the `if` section and exceptions in the `else` section of the `if` statement

```
boolean isOk = readFile(fileName);
if (isOk) {
  :
}
else {
  :
}
```

Ensure that the exception doesn't cover up the normal path the execution should take; this is an important point, not just for readability but for performance as well

- Put the conditional on another line

Good Example:
```
if (isDone)
```

Bad Example:
```
if (isDone) doCleanup();
  doCleanup();
```

We do this for the purposes of debugging. If you place them on the same line, it will not be clear if the test is true or false

- Avoid using executable statements in a conditional

Good Example:
```
InputStream stream = File.open(fileName, "w");
if (stream != null) {
  :
}
```

Bad Example:

```
if (File.open(fileName, "w") != null)) {
  :
}
```

A conditional that contains an executable statement is incredibly hard to read, especially for those who are new to Java programming

Numbers

- Avoid using magic numbers in your code. Anything other than 0 and 1 is considered to be declared as a named constant

Good Example:

```
private static final int  TEAM_SIZE = 11;
:
Player[] players = new Player[TEAM_SIZE];
```

Bad Example:

```
Player[] players = new Player[11];
```

If the number on its own does not have any obvious meaning, add a named constant instead to enhance readability

- Always write floating point numbers with a decimal point and a minimum of one decimal

Good Example:

```
double total = 0.0;
```

Bad Example:

```
double total = 0;
```

Good Example:

```
double speed = 3.0e8;
```

Bad Example:

```
double speed = 3e8;

double sum;
:
sum = (a + b) * 10.0;
```

These examples show you the difference in the natures of the floating-point number and the integer. In mathematical terms, they are both different and neither is compatible with the other. Also, as you can see in the above example, this emphasizes the assigned variable type at a place in your code where it might not be obvious or evident

- Always put a digit in front of the decimal point in a floating-point constant

Good Example:

```
double total = 0.5;
```

Bad Example:

```
double total = .5;
```

The system of expressions and numbers in Java comes from the mathematical system and you should stick to the mathematical convention as far as possible for syntax purposes. Not only that, it is easier to read 0.5 than it is to read .5 and you can't muddle it up with the integer 5.

- Refer to static methods or variables using the class name, not the instance variable

Good Example:

```
Thread.sleep(1000);
```

Bad Example:

```
thread.sleep(1000);
```

These examples show that the element reference is independent of any specific instance and is static. For the very same reason, you should include the class name when you access a method or a variable from the same class.

Chapter 5:
White Space

Whitespace is another important part of coding, especially knowing how and where to use it. The general rules are:

- Always surround operators with space characters
- Follow reserved keywords with a white space
- Follow a comma with a white space
- Surround colons with white space
- Follow semi-colons that are in a `for` statement with a space character

Good Example:
```
a = (b + c) * d;
```

Bad Example:
```
a=(b+c)*d
```

Good Example:
```
while (true) {
```

Bad Example:
```
while(true){
  ...
```

Good Example:
```
doSomething(a, b, c, d);
```

Bad Example:
```
doSomething(a,b,c,d);
```

Good Example:
```
case 100 :
```

Bad Example:
```
case 100:
```

Good Example:
```
for (i = 0; i < 10; i++) {
```

Bad Example:
```
for(i=0;i<10;i++){
  ...
```

By doing all of this you are making the components of each statement stand out, increasing readability. While it isn’t very easy to provide you with a full list of white space use, the above examples should give you some idea.

- Use a white space after a method name if it has another name after it

```
doSomething (currentFile);
```

This ensures that each individual name stands out and makes it easier to read. If there is no name following the method name, leave out the white space as the name will be obvious.

```
(doSomething())
```

There is an alternative method here – follow the opening parenthesis with a white space. Some who do this place another white space before the closing parenthesis but this is not really needed. It does make things stand out more though and there is no harm in doing it.

Example – two white spaces

```
doSomething( currentFile );
```

Example – one white space

```
(doSomething( currentFile);)
```

- Separate all logical units in a block with a blank line

```
// Create a new identity matrix
Matrix4x4 matrix = new Matrix4x4();

// Precompute angles for efficiency
double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
matrix.setElement(1, 1,  cosAngle);
matrix.setElement(1, 2,  sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2,  cosAngle);

// Apply rotation
transformation.multiply(matrix);
```

By bringing in a white space between each logical unit, you are making the code more readable. Blocks are usually started with a comment, as you can see above.

- Use three blank lines to separate each method

By enlarging the space bigger than the space in the method, all of the methods will stand out

- Align variables in a declaration to the left

```
TextFile   file;
int        nPoints;
double     x, y;
```

This makes the code read better and, because they are aligned, you will better be able to see the variable types.

- Align statements wherever it makes the code more readable

```
if      (a == lowValue)    compueSomething();
else if (a == mediumValue) computeSomethingElse();
else if (a == highValue)   computeSomethingElseYet();

value = (potential        * oilDensity)   / constant1 +
        (depth            * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity)   / constant3;

minPosition     = computeDistance(min,     x, y, z);
averagePosition = computeDistance(average, x, y, z);

switch (phase) {
  case PHASE_OIL   : text = "Oil";   break;
  case PHASE_WATER : text = "Water"; break;
  case PHASE_GAS   : text = "Gas";   break;
}
```

There are quite a few places in Java code where white space can be added to make it more readable, even if adding it causes a violation of the common guidelines Many of these places are associated with code alignment and, while it isn’t easy to provide a full list of the general guidelines, the examples above should help you.

Chapter 6:
Coding Do’s and Dont’s

These are just a few more do’s and don’ts to help make you a more effective programmer:

- Only use a return at the end of the method, never in the middle

If you use a return in the middle, you will struggle to break the method down into smaller ones later if you want to. It will also force you to look at several exit points in one method.

- Don't use continue

Again, this makes it difficult to break a construct down into smaller ones or into methods later on, as well as forcing you to look at several endpoints in one construct

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.

- Use separate lines for increments or decrements; never compound them

When you compound an increment or a decrement operator into a method call makes it difficult to read, not good for the programmer with little experience when it comes to modifying code.

Bad Example:

```
foo(x++);
```

Good Example:

```
foo(x);
    x++;
```

Bad Example:

```
y += 100 * x++;
```

Good Example:

```
y += 100 * x;
    x++;
```

- Always declare a variable as near to where it is going to be used as you can.

Example 1:

```
    int totalWide;
    int firstWide = 20;
    int secondWide = 12;
    firstWide = doFoo(firstWide, secondWide);
    doBar(firstWide, secondWide);
    totalWide = firstWide + secondWide;          //  wrong!
```

Example 2:

```
     int firstWide = 20;
    int secondWide = 12;
    firstWide = doFoo(firstWide, secondWide);
    doBar(firstWide, secondWide);
```

```
int totalWide = firstWide + secondWide;     //  right!
```

Example 3:

```
int secondWide = 12;
int firstWide = doFoo(20, secondWide);
doBar(firstWide, secondWide);
int totalWide = firstWide + secondWide;     //  even better!
```

Quote:

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."

- *Martin Fowler*

Instead of trying to document how a complex algorithm is performed, just make the algorithm more readable in the first place by adding in identifiers. This will help if the algorithm is changed in the future but the documentation is not updated.

Example:
Rather than this:

```
 if ( (hero == theTick) && ( (sidekick == arthur) || (sidekick == speak) ) )
```

Do this instead::

```
boolean isTickSidekick = ( (sidekick == arthur) || (sidekick == speak) );
if ( (hero == theTick) && isTickSidekick )
```

OR
Instead of this:

```
 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.");
    }
}
```

Do this:

```
 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.");
    }
}
```

Chapter 7:
Common Java Syntax Mistakes

To finish off, I am going to go over some of the more common Java syntax mistakes – getting these wrong can mess everything up!

- Keyword Capitalization

Because class names are capitalized, on occasion you might find that you do the same with a keyword, like the class **andint .** If you capitalize the first letter of this class, your compiler will not be happy and will throw up an error message – the message will depend on the keyword that you have capitalized but you will see something like this:

```
Line nn: class or interface declaration expected
```

- Extending strings over new lines

Sometimes your strings are going to be long but one of the most basic errors is to have a newline in the string. Again, your compiler is not going to like this and you will get an error message like:

```
Line nn: ';' expected
```

If that happens, the way around it to is to split your string into two, ensuring that neither contains a new line and then concatenate the strings with a '+':

Instead of this:

```
String s = "A rather long string which spills over the end of the line
and will cause your compiler a big problem";
```

with:

```
String s = "A rather long string which spills over the end "+
"of the line and will cause your compiler a big problem"
```

- Not putting brackets into a message with no arguments

If you have a method that has no arguments, there should be brackets after the method name. For example, if you declared the method **carryOut** without any arguments and you send a message that corresponds to the method that is to the object **objSend ,** it should be written like this:

```
objSend.carryOut()
```

and not:

```
objSend.carryOut
```

- Forgetting to import packages

This is a basic mistake and one of the more common for those not experienced with Java. When you omit the import statement from the start of the program, your compiler will give you a message like this:

Line nn: Class xxxx not found in type declaration

That said, as java.lang is automatically imported, it doesn't require an import statement

- Muddling up static and instance methods

A very common mistake is to send a static method message to an object whereas these methods are associated with classes. For example, let's say you wanted to work out the absolute value of **intvalue** and then put it in the int variable; you would write:
int result = Math.abs(value);
rather than:
int result = value.abs();

Several syntax errors can arise the most common one being:
Line nn: Method yyyy not found in class xxxx.

In this, xxxx is the class name and yyyy is the method name

- Using the wrong case with classes

Another common error; because Java is a case sensitive language, if you use the wrong case, it won't recognize it. For example, if you write string and not String, you would get an error message like this:

Line nn: Class xxxx not found in type declaration.

In this, xxxx is the class name with the wrong case

- Using the wrong case with variables

The same thing applies here; variables are case sensitive as well and, if you were to declare a variable **linkEdit** as an int and then attempted to refer to it in a class, you would get an error message like this:

Line nn: Undefined variable: xxxx

In this, xxxx is the mistyped variable name

- Using the wrong format for writing class methods

Class methods are always written in this form:
ClassName.MethodName(Argument(s))

One of the most common mistakes is to omit the name of the class and that will throw up a message like this:

```
Line nn: '}' expected
```

- Wrongly specifying a method argument

When a class is defined, each argument should be prefixed with either the name of a class

```
public void tryIt(int a, int b, URL c)
```

One of the more common errors, especially if you have come to Java from another programming language, is to not prefix ALL arguments with the type. For example, taking

```
public void tryIt(int a, b URL c)
```

This will generate an error message something like this:

```
Line nn: Identifier expected
```

- Forgetting to send a message to an object

This isanother error that more common to those who are new to oop, or object oriented programming. Let's say that you have the **tryIt** method; it has two int arguments and it will return an int value. If we assume that the method is to send a message to an object, we would write it like this:

```
int newVal = destination. tryIt(arg1, arg2)
```

The arguments, or ints, have been declared in this code whereas, if you write it like the next example:

```
int newVal = tryIt(destination, arg1,arg2)
```

You would get an error message like this:

```
Line nn: ')' expected
```

- Treating == as value equality

We use == to compare values with scalars and, when you apply it to an object, it will compare addresses instead. For example, let's look at an **if** statement:

```
if(newObj1 == newObj2){
...
}
```

This executes the code that is denoted by … but ONLY when the first object has the same address as the second one. When they have different addresses but the same instance variable values, you would get an evaluation to false. This won't give you any syntax errors but you will see it when you execute the program.

- Not putting voids into methods

If a method carries out an action but doesn't return any result, you need to insert the keyword **void** before the method name. If you don't, you will likely see an error message that looks something like:

```
Line nn: Invalid method declaration; return type required
```

- Not putting a break in a case statement

This is common in both procedural and object oriented languages. You must put a break statement at the end of a case statement if you want it to finish and then exit at the end of said case statement. If you don't put the break statement in, execution will go on to the branch beneath the one where you left out the break statement

- Not putting a return into a method

If a method is returning a value, there should be a minimum of one return statement in the method body that returns the right value type. Not doing this will result in an error message like this:

```
Line nn: Return required at end of xxxx
```

In this case, xxxx is the method that has no return

- Declaring an instance variable as private and referring to it in another class by name

When you declare an instance variable as private, you cannot access it by its name outside of the class. The only way to do this is to use a method that is declared within the class in which the instance variable resides. You will see an error message like this:

```
Line nn: Variable xx in class xxxx not accessible from class yyyy
```

In this example, xx is a private variable, xxxx is the class where the variable is defined and yyyy is the class in which the variable is referred to

- Making use of a variable before you assign it a value

Another common error in procedural and object oriented languages. Scalers are initialized to a default value or to zero in Java so that no errors are indicated. If there are any, they are shown by an array that goes beyond its bounds, or a false result. Objects are initialized to null and if you try to reference an object that has not been initialized, it will be caught when it comes to run time.

- Assuming the incorrect value type will be generated by a message

This is another popular error when you use Java packages. One example is the use of a method that will deliver a string with digits in it but treating it as an integer. The method **getInteger** inside of **java.lang.Integer** will deliver an integer and if you try to use that value as an int , for example, you would see an error message like this:

```
Line nn: Incompatible type for declaration can't convert xxxx to yyyy
```

- Mixing up prefix and postfix operators

The postfix operators, like – and ++, are used to take an old value of the variable they are applied to while the prefix operators are for the new value. So, if x is 45 and you have this statement:

```
y = ++x
```

When it is executed, x and y will both be 46. However, if this statement were to be executed:

```
y = x++
```

y would be 45 and x would be 46. You won’t see these errors while you are compiling but they will become evident at run time

- Failing to remember that, if an argument is an object, it is passed to a method by reference

When you use an object as a method argument, you pass the address and not the value. This means that values can be assigned to these arguments. While treating them as a value won't particularly kick up an error, you won’t be making proper use of the object oriented language

- Failing to remember that a scalar is passed to a method by value

Arguments that take the form of a scalar cannot be treated if they can be assigned to. While this won’t come up as a syntax error, it will show up at run time if you write a piece of code that assumes a method has been used to pass a value to a scalar

- Not using size properly for arrays and strings

Size is an instance variable that is associated with a method when it is associated with a string and with arrays. If you were to muddle them up by writing something like:

```
arrayVariable.size()
```

or

```
stringVariable.size
```

the first example would give you an error message like:

```
Line nn: Method size() not found in class java.lang.Object
```

And the second example would give an error message like:

```
Line nn: No variable size defined in java.lang.String
```

- Using non-existent constructors

It is a common error to use an undefined constructor. For example, you could have a class in which you have the following constructors – one int , two int, three int – but you might have used a four int constructor. You would see this when compiling with an error message like:

```
Line nn: No constructor matching xxxx found in class yyyy
```

In this example, xxxx is the signature for the non-existent constructor and yyyy is the class where you should have defined it

- Calling constructors within a constructor that has the same name

Let's say, for example, that you defined a class called x with a **one int** anda **two int** constructor. In the **two int** you have referenced the argument x and this will kick up an error like:

```
Line nn: Method xxxx not found in yyyy
```

In this example, xxxx is the constructor name with the arguments and yyyy is class name where it is defined. The answer to this is to use the keyword this

- Assuming that a 2-dimensional array will be implemented directly in Java

This will result in wrong code, like:

```
int [,] arrayVariable = new [10,20] int
```

In java, this is illegal and you will see errors like this:

```
Line nn: Missing term
```

and:

```
Line nn: ']' expected
```

Multi-dimensional arrays can be implemented in Java but must be treated as single-dimension arrays that contain another single-dimension array, and so on.

- Treating scalars as objects

Scalars, like `float` and `int`, are not objects but you may want to treat them like objects when you deposit them into a Vector, like this:

```
Vector vec = new Vector();
vec.addElement(12);
```

This will result in a syntax error like this:

```
Line nn: No method matching xxxx found in yyyy
```

In this example, xxxx is the method name and yyyy is the class name that is expecting the object.

To solve this, convert them into objects using the object wrapper class that you will find in java.lang

- Confusing a scalar and its corresponding object type

When you use a scalar like `int` it isn't difficult to write a piece of code that assumes it can be treated like an object. For example:

```
int y = 22;
Integer x = y;
```

Will throw up an error message like:

```
Line nn: Incompatible type for declaration. Can't convert xxxx to yyyy
```

In this example, both yyyy and xxxx are the class names

- Not typing the main method header correctly

Execution of a Java application requires a method declarations starting with:

```
public static void main (String []args){
```

If any of that line is not types properly or you omit a keyword, you will get an error at run time.

Let's say you omitted the keyword `static`, you would get an error message like:

```
Exception in thread main.....
```
will be generated at run time.

Conclusion

Thank you again for reading this book!

I hope this book was able to help you to understand how to neaten up your Java code and how to write cleaner and neater code for more effective and efficient programming.

The next step is to practice. And keep on practicing because that is the only way you will improve. The trick with computer programming languages like Java is to practice regularly because things fall out of fashion and new coding rules and styles come in every day. And if you don ’ t keep up, you will find yourself having to go right back to basics every time.

Finally, if you enjoyed this book, then I ’ d like to ask you for a favor, would you be kind enough to leave a review for this book on Amazon? It ’ d be greatly appreciated!

Thank you and good luck!

About the Author

Charlie Masterson is a computer programmer and instructor who have developed several applications and computer programs.

As a computer science student, he got interested in programming early but got frustrated learning the highly complex subject matter.

Charlie wanted a teaching method that he could easily learn from and develop his programming skills. He soon discovered a teaching series that made him learn faster and better.

Applying the same approach, Charlie successfully learned different programming languages and is now teaching the subject matter through writing books.

With the books that he writes on computer programming, he hopes to provide great value and help readers interested to learn computer-related topics.