Capgemini India Private Limited

# Java Coding Standards

## Version 0.1

Aliasgar Muchhala (Apps One – CSD)

Feb 2013

# Table of Contents

# Chapter 1: Purpose

This document describes a collection of standards, conventions, and guidelines for writing solid Java code. They are based on sound, proven software engineering principles that lead to code that is easy to understand, to maintain, and to enhance. Furthermore, by following these coding standards productivity should increase remarkably. Experience shows that by taking the time to write high-quality code right from the start, it will make it much easier to modify it during the development process. Finally, following a common set of coding standards leads to greater consistency, making teams of developers significantly more productive.

Coding standards for Java are important because they lead to greater consistency within your code and the code of your teammates. Greater consistency leads to code that is easier to understand, which means it is easier to develop and to maintain. This reduces the overall cost of the applications that you create.

Remember that the Java code will exist for a long time. An important goal during development is to ensure that the code can be transitioned to a support team. Code that is difficult to understand runs the risk of being scrapped and rewritten.

This document may be tailored to suit specific project needs, though should form the basis for most projects. Where you intend to deviate from a practice here, annotate the item along with reasons for traceability.

# Chapter 2: Scope

This document was written to covers up to JDK 1.4, although most of the principles are generic rather than version specific.

Language enhancements are covered in Chapter 5: Java 5 and 6 Language Enhancements

# Chapter 3 – Coding Styles & Techniques

## 3.1 Code Organization

Classes should be organized as follows;

- Package Identifier
- Class JavaDoc
- Class Declaration
- Log4J Logger Declaration
- Constants
- Member Variables
- Constructors
- Member Functions

## 3.2 Methods should do one thing and one thing well

Each method should perform a single task. Create additional methods for sub-functions, even where, for now, this function is only invoked from one place. This makes methods more re-usable.

Methods written in this way are easy to name. If you can't name a method easily then this may be a sign that it does too many things

## 3.3 Naming

One of the key tenets of good design is having a clear understanding the responsibility of packages, classes and methods. This should be clearly understood by the reader through the naming of packages, classes and methods.

The process of choosing good names challenges the core understanding of responsibilities in code and design.

The following are general naming convention guidelines.

- Use full English descriptors that accurately describe the variable, field, and class; for example, use names like firstName, grandTotal, or CorporateCustomer. Although names like x1, y1, or fn are easy to type because they're short, they do not provide any indication of what they represent and result in code that is difficult to understand, maintain, and enhance. Note that I, j, k are fine for loop variables.

- Use terminology applicable to the domain. If your users refer to their clients as customers, then use the term Customer for the class, not Client. Many developers make the mistake of creating generic terms for concepts when perfectly good terms already exist in the industry or domain.

- Use mixed case to make names readable. Use lowercase letters in general, but capitalize the first letter of class names and interface names, as well as the first letter of any non-initial word.

- Use abbreviations sparingly, but if you do so then use them intelligently. A list of standard short forms (abbreviations) should be maintained.

- Avoid long names (< 15 characters is a good idea). Although the class name PhysicalOrVirtualProductOrService might seem to be a good class name at the time, this name is simply too long and you should consider renaming it to something shorter, perhaps something like Offering.

- Avoid names that are similar or differ only in case. For example, the variable names persistentObject and persistentObjects should not be used together, nor should anSqlDatabase and anSQLDatabase.

- Avoid (leading or trailing) underscores. Names with leading or trailing underscores are usually reserved for system purposes and may not be used for any user-created names except for preprocessor defines. More importantly, underscores are annoying and difficult to type so try to avoid their use whenever possible.

Avoid using types in names e.g. custArray, nameString, as the type name used may not match the actual type.

### 3.3.1 Member Data Names

All member data will begin with a lowercase letter, and then follow the normal naming convention for other identifiers (mixed case, each word beginning with uppercase). Do not provide publicly visible member data, use access functions instead. Respect the Java Beans coding patterns for member access unless you have a good reason not to.

```
public class MyClass {
 private int myOrdinalValue;

 public void setMyOrdinalValue(int aValue);

 public int getMyOrdinalValue();

};
```

### 3.3.2 Parameter Names

Parameter names should be constructed like identifiers, do not include the type in the name.

### 3.3.3 Variable Names

Method variable names (not class member variables) should be constructed like identifiers, the exceptions are the commonly used i, j ,k etc for **for** loop variables.

### 3.3.4 Component Factory Names

A component factory is a public class that implements only static methods. These static methods are "Factory functions" or "component constructors". Factory class names should include the word "Factory". Factory method names should start with the word "get", for example,

```
public class WidgetFactory
{
    static Button getButton(int aButtonType);
    static ListBox getListBox();
};
```

## 3.4 Methods should have one exit point

Code with multiple returns can be hard to read / understand. This also tends to be a symptom of large methods thats should be refactored down.

## 3.5 Use equals() when comparing objects

Use equals() instead of == when comparing objects. If you use == you are comparing an objects id's not its values.

If someone defined an equals method to compare objects, then they want you to use it. Otherwise, the default implementation of Object.equals is just to use ==.

```
Integer a = new Integer(99);
Integer b = new Integer(99);
(a == b); //Returns False\!
```

Given the above when you create a class consider seriously overriding the equals method.

## 3.6 Always override hashcode when overriding equals

Whenever a.equals(b), then a.hashCode() must be same as b.hashCode().

In practice that means you have to use the same set of fields that you use to compute equals() to compute hashCode().

## 3.7 Log Messages

Always check the logging level before issuing a log statement if the statement being logged contains any computation e.g. concatenation

```
if (logger.isDebug) {
logger.debug("Message length is " + message.length());
}
```

This reduces the impact of lower level messages when a higher logging level is applicable.

## 3.8 What to log and when

Consider what is logged in a live defect scenario. A supportable piece of code should be "debuggable" with using just log levels in live (the live service test is could you diagnose a problem 1 year after you wrote the code at 2am based on the source and log files running at debug).

Based on log4j (your logging framework may differ) the following rules apply to the logging levels.

FATAL – server failed, operations have to do something immediately to stand it back up (if the system runs unattended at some times, this would – through OpenView or Tivoly – alert the on-call operator)

ERROR – data loss, but can proceed
WARN – no data loss, but something fishy (e.g. inconsistent data  received on an interface – no immediate concern, but someone should look  into this)
INFO – regular events, e.g. batch has completed, server startup completed, shutdown initiated etc.
DEBUG – debugging information by the developer, can stay activated during a performance test
TRACE – very detailed information, should be disabled during performance tests (would be active during unit tests etc.).

## 3.9 Avoid 'weird' language constructs

Avoid using non-intuitive language constructs just to reduce the amount of code. This is also an example of avoiding doing more than one thing in a single line of code. This makes code easier to read and maintain.

Examples

```
boolean c = (a==b)?true:false;
```

is quite obscure and much clearer written as

```
boolean c;
if (a==b) {
c = true;
} else {
c = false;
}
```

but (in this example)

```
booloean c = (a==b);
```

Doesn't really improve readability.

Similarly

```
int a = ++b;
int a = --b;
int a = b++;
int a = b--;
```

can cause confusion, better to use

```
b++
int a = b;
```

or

```
int a = b;
b++;
```

## 3.10 Avoid * forms of import

Avoid * forms of import. Be precise about what you are importing so as to avoid subtle bugs where you think you are using classes of the same name but that live in different packages.

Check that all declared imports are actually used (a good IDE will highlight those not used). This makes it easier for readers of your code to understanding its context and dependencies.
Use your IDE's 'add import' and 'organise imports' commands to make this easier.

## 3.11 Declare a local variable where you know its initial value

Declare a local variable only at that point in the code where its initial value is known. This minimizes bad assumptions about values of variables. The only exceptions to this rule are loop variables which should be declared outside the loop for efficiency reasons and return values that should be declared at the beginning of the method e.g.

```
private String getData() {
  String data = null;  //Used for return value
  String name;         //Used in loop
  String value;        //Used in loop

  for (int i=0; i < 10; i++) {
    name = arrya1i;
    value = array2i;
    System.out.println(name + "=" + value);
  }

  if (condition) {
    data = someValue;
  }

  return data;
}
```

## 3.12 Use a new local variable

Declare and initialize a new local variable rather than reusing (reassigning) an existing one whose value happens to no longer be used at that program point. Especially loop variables. This minimizes bad assumptions about values of variables.

## 3.13 Comments

- Aim to write code which is self explanatory and needs little to no commenting. Ideally you will be able to structure your code and name your methods and variables so that only a minimum of additional explanation is required. Public methods **should** however contain javadoc comments.
- Comments should add to the clarity of your code. The reason you document your code is to make it more understandable to you, your colleagues, and to any other developer who comes after you.
- If your program is not worth documenting, it is probably not worth running.
- Avoid decoration; that is, do not use banner-like comments. In the 1960s and 1970s, COBOL programmers got into the habit of drawing boxes, typically with asterisks, around their internal comments. This added little value to the end product. Write clean code, not pretty code. As many of the fonts used to display and print code are proportional, and many are not boxes can't be lined up properly anyway.
- Keep comments simple. Some of the best comments are simple, point-form notes. Just provide enough information so that others can understand the code.

- Write the documentation before you write the code. The best way to document code is to write the comments before you write the code. This gives you an opportunity to think about how the code will work before you write it and will ensure that the documentation gets written. Alternatively, you should at least document your code as you write it. Because documentation makes your code easier to understand, you are able to take advantage of this fact while you are developing it. If you are going to invest the time writing documentation, you should at least get something out of it.

- Document why something is being done, not just what. This is particularly true of simple statements such as 'i++' where adding a comments such as 'Increment the counter' adds no benefit where as 'Increment byte counter to allow for marker character' does.

### 3.13.1 Types of Java Comment

Java has three styles of comments:

- documentation comments that start with /** and end with */
- C-style comments that start with /* and end with */
- single-line comments that start with // and go until the end of the source-code line

The following chart is a summary of a *suggested* use for each type of comment, as well as several examples.

| ***Comment Type*** | ***Usage*** | ***Example*** |
|---|---|---|
| **Documentation** | Use documentation comments immediately before declarations of interfaces, classes, member functions, and fields to document them. Documentation comments are processed by javadoc, see below, to create external documentation for a class. | /**Customer: A customer is any<br>*person or organization that we<br>*sell services and products to.<br>@author S.W. Ambler*/ |
| **Single line** | Use single line comments internally within member functions to document business logic, sections of code, and declarations of temporary variables. | // Apply a 5% discount to all invoices<br>//over $1000 due to generosity<br>//campaign started in Feb. of 1995. |

### 3.13.2 Javadoc

Included in Sun's Java Development Kit (JDK) is a program called *javadoc* that processes Java code files and produces external documentation, in the form of HTML files, for your Java programs. *Javadoc* supports a limited number of tags; reserved words that mark the beginning of a documentation section. Please refer to the JDK *javadoc* documentation for further details. The following are the required javadoc tags;

| ***Tag*** | ***Used for*** | ***Purpose*** |
|---|---|---|
| **@author** | Classes, Interfaces | Indicates the author(s) of a given piece of code. |
| **@exception name description** | Member Functions | Describes the exceptions that a member function throws. You should use one tag per exception and give the full class name for the exception. |
| **@param name description** | Member Functions | Used to describe a parameter passed to a member function, including its type or class and its usage. Use one tag per parameter. |
| **@return description** | Member Functions | Describes the return value, if any, of a member function. You should indicate the type or class and the potential use(s) of the return value. |

The way that you document your code has a huge impact, both on your own productivity and on the productivity of everyone else who later maintains and enhances it. By documenting your code early in the development process, you become more productive because it forces you to think through your logic before you commit it to code. Furthermore, when you revisit code you wrote days or weeks earlier, you can easily determine what you were thinking when you wrote it because it's already documented for you.

## 3.14 Unfinished Code

Mark unfinished code with an @TODO annotation including a description. You should generally avoid checking in code with outstanding // TODO comments in it - this means you're not finished. Change then to @TODO instead.

## 3.15 Declaration

In general declarations should be **private** except where design dictates otherwise. The obvious exceptions are constructors and bean get/set methods.

### 3.15.1 Classes

Classes should be declared as follows using capitalized English words for the class name.

```
/**
 * Class Description
 *
 * @author PJF
 */
public class ClassName extends A implements B, C, D {

}
```

### 3.15.2 Constants

Constants should be declared as follows, using all Capitals separated by an underscore.

```
/**
 * Status for successful operation
 */
public static final String STATUS_OK = "Y";

/**
 * Status for unsuccessfully operation
 */
public static final String STATUS_FAILED= "F";
```

The name is constructed using a word or words to describe the group of constants (in this case STATUS values) followed by a word or words describing the meaning of the value.

### 3.15.3 Logger Declaration

Each class should declare a Log4J logger instance.

```
private static Logger logger = Logger.getLogger(ThisClass.class);
```

The Logger declaration may be omitted for an Abstract class that does no logging. Subclasses must declare their own logger.  It is not necessary to comment or javadoc a logger declaration. It should be clear what it is.

### 3.15.4 Member Variables

Member variables should be declared as follows, first letter lowercase, then using Capitalised English names.

```
private string customerName;
```

### 3.15.5 Constructors

A default (no args) constructor may be omitted, as the compiler will define one.

Where practical an additional (or several) constructor(s) should be supplied which fully initializes the class. Classes which are constructed using the default constructor and which subsequently require lots of calls to setXXX() methods to initialize them should be avoided.

### 3.15.6 Member Functions

Member functions should be declared as follows, using capitalized (except for the first word) English names.

```
/**
 * Function Description.
 *
 * @param param1 First Parameter description
 * @param param2 Second Parameter description
 * @exception MyException Error description */
public MyThing functionName(MyParam param1, MyParam param2) throws MyException {

}
```

Member functions should follow the definitions for Constants and Member Variables in the source file.

## 3.16 Expressions and Statements

Much of the formatting recommendations that follow can be done automatically using your IDE's 'Format' command (or similar).

For the loop constructs, **break** and **continue** should be used rather than return. The use of **break** is particularly important in the switch statement as code execution will by default continue into the next block.

### 3.16.1 Braces

The starting brace should be at the end of the condition. The ending brace must be on a separate line and aligned with the conditional. All conditional constructs should define a block of code even for single lines of code. The exception to this rule is where the closing and opening brace share the same line as the 'else'.

### 3.16.2 If/Else

Place the if / else keyword and conditional expression on the same line.

```
if (expression) {
  statement;
} else if (expression) {
  statement;
} else {
  statement;
}
```

### 3.16.3 While

The While construct uses the same layout format as the IF construct. The WHILE keyword should appear on its own line, immediately followed by the conditional expression. The statement block is placed on the next line.

```
while (expression) {
  statement;
}
```

### 3.16.4 Do/While

The Do While form of the while construct should appear as shown below:

```
do {
  statement;
} while (expression);
```

### 3.16.5 Switch

The Switch construct uses the same layout format as the if construct. The Switch keyword should appear on its own line, immediately followed by its test expression. The statement block is placed on the next line.

```
switch (expression) {
  case n:
    statement;
    break;
  case x:
    statement;
    // Continue to default case
  default:                        //always add the default case
    statement;
    break;
}
```

Always ensure that a switch statement has a default case.

### 3.16.6 Try/Catch/Finally

The try/catch construct is similar to the others.  Try keyword should appear on its own line; followed by the open brace on the same line; followed by the statement body; followed by the close brace on its own line.  Any number of CATCH phrases are next consisting of the CATCH keyword and the exception expression on its own line; followed by the CATCH body; followed by the close brace on its own line. The FINALLY clause is the same as a CATCH.

```
try {
  statement;
} catch (ExceptionClass e) {
  statement;
} finally {
  statement;
}
```

The finally clause should be used to tidy up any resources which have been opened e.g. Files, Streams, JDBC Connections.

## 3.17 Exception Handling

Always code for the non happy day scenario.

### 3.17.1 Exception Handling Best Practices

- Use unchecked exceptions for system exceptions.
- Use checked exceptions for application exceptions. (See below for some discussion on what can be considered as an "ApplicationException").
- Only catch an exception if you can do something with it or you need to wrap it in something more meaningful to a higher level in the application. By doing so we will avoid redundant copy-and-paste code for exception handling.
- Log an exception as early as possible. That way it is easier to narrow down the problem.
- Log an exception only once unless you are about to throw it beyond a boundary back to a remote client, in which case logging it again is acceptable.  Note that this does not preclude you from logging elsewhere at "Debug" level as this will be disabled in Production.
- Provide *useful* information when throwing an exception. Think about the layers above the layer where your Exception throwing code lives.
- Do not implement any of the anti-patterns (see Anti Patterns) unless you have a *very* good reason. In these cases it is valid to annotate them with an @TODO to explain yourself.

Exceptions being thrown between physical machines will need to be remotable as potentially they will be going over physical boundaries.

### 3.17.2 Exception Handling Anti Patterns

#### *Do not use System.out.XX or System.err.XX*

```
catch (NoSuchMethodException e) {
  System.out.println(e.getMessage());
}

catch (NoSuchMethodException e) {
  System.err.println(e.getMessage());
}
```

### *Avoid log and throw*

All these examples are equally wrong. Either log and handle it or throw it. Don't do both. Do that to avoid an Exception being logged more than once.

```
catch (NoSuchMethodException e) {
  LOG.error("My message", e);
  throw e;
}

catch (NoSuchMethodException e) {
  LOG.error("My message", e);
  throw new MyServiceException("My message", e);
}

catch (NoSuchMethodException e) {
  e.printStackTrace();
  throw new MyServiceException("My message", e);
}
```

**NOTE:** An exception to this is when an Exception is crossing a physical boundary and back to a client. In this case it is perfectly valid to log an exception which is also (re)thrown. This means that in a multi-channel environment, where the client may not be known but does need the exception in order to react accordingly, stack traces are not lost.

### *Avoid throwing java.lang.Exception*

Never use the clause 'throws Exception' in a method. Either throw a specific checked exception or use an unchecked exception. Using the 'throws Exception' clause just means, that something might go wrong.

```
public void doSomething() throws Exception {
  // do something
}
```

**NOTE:** Furthermore, both Checked and Unchecked Exceptions which may be thrown by a subsystem method called from different channels should be declared in the method signature.  This ensures that, when different channels (such as Web Services) are opened up, the full client contract is clearly communicated in the API.

### *Avoid throwing multiple exceptions for the same reason*

Don't throw multiple checked exceptions which basically mean the same for the caller. If there are for example three different checked exceptions thrown by a method, which can be handled *differently* by the caller, then it's ok. Otherwise they should be wrapped to a single exception and only include this one in your throw clause.

```
public void doSomething() throws MyException,
AnotherException, SomeOtherException
{
  // do something
}
```

### *Avoid Catch Exception*

This is generally wrong. Catch only the specific exceptions that can be thrown.

```
try
{
// do something
} catch (Exception e) {
  LOG.error("something went wrong", e);
}
```

### *Avoid Destructive Wrapping*

Always preserve the stacktrace if you wrap an exception!

```
catch (NoSuchMethodException e) {
  throw new MyServiceException("My message " +
    e.getMessage());
}
```

### *Avoid Catch and Ignore*

By doing so, an exception will be swallowed which is generally wrong.

```
catch (NoSuchMethodException e) {
  return null;
}
```

```
catch (NoSuchMethodException e) {
  // do nothing
}
```

### *Never execute code in a 'finally' block, that might throw an exception*

If in the following example, the developer has to make sure, that the 'cleanUp'-method doesn't throw an exception. It would be obviously wrong to allow the 'cleanUp'-method to throw any kind of exception, because a possible exception in the try/catch-block would be lost forever, if the 'cleanUp'-method would throw its on its own.

```
try {
  // do something
} finally {
  cleanUp();
}
```

### *Never Log and return null*

Although it might not be wrong, it is still bad style. A better solution is to throw an exception and let the caller handle it. By the way, the use of 'printStackTrace' is wrong anytime.

```
catch (NoSuchMethodException e) {
  LOG.error("My message", e);
  return null;
}

catch (NoSuchMethodException e) {
  e.printStackTrace();
  return null;
}
```

### *Always Throw UnsupportedOperationException instead of returning null*

If you implement an abstract base class and provide hooks for some base classes to override, use a UsupportedOperationException instead of returning null. The following example shows the wrong and the right behaviour:

```
// wrong
```

```
public MyObject getMyObject() {
  // Not supported in this implementation.
  return null;
}

// correct
public MyObject getMyObject() {
  // Not supported in this implementation.
  throw new UnsupportedOperationException("Reasonable Error Message");
}
```

## 3.18 Memory Management

Java provides it's own memory handling mechanism called the Garbage Collector. Programmers should be aware of how this works.

Be aware of

- The implications of String concatenation (and the alternative methods using StringBuffer) and other operations on immutable types.
- Create arrays, buffers etc at their anticipated size to reduce re-sizing overheads.
- Don't create unnecessary objects e.g. In loops don't

```
for (int i=0; I < 10000; I++) {
  MyObject ob = someVar.getMyClass;
}
```

better to

```
MyObject ob = null;

for (int i=0; I < 10000; I++) {
  ob = someVar.getMyClass;
}
```

## 3.19 Portability

Java is inherently portable. Programmers should never use operating system specific features. In particular

File names should be constructed using the File.pathSeperator or File.pathSeperatorChar static fields on java.io.File.

## 3.20 Compilation Issues

Application compilation should only be dependent on 3rd party API libraries and the Root Application. Applications should not be dependent on one another. If this situation is anticipated to arise, the dependant classes should be moved to the Root Application.

# Chapter 4 – Recommended Reading

- CheckStyle - Code format checking and standards compliance tool. http://sourceforge.net/projects/checkstyle
- Sun Code Conventions http://www.oracle.com/technetwork/java/codeconv-138413.html
- Thinking in Java - Bruce Eckel http://www.mindview.net/Books/TIJ/
- CSD Build Automation and C.I. guidelines http://wiki.capgemini.com/index.php/Javacop:industrialisation:autoBuildAndCI

# Chapter 5 – General Principles

1. **Elegance always pays off**.

In the short term it might seem like it takes much longer to come up with a truly graceful solution to a problem, but when it works the first time and easily adapts to new situations instead of requiring hours, days, or months of struggle, you'll see the rewards (even if no one can measure them). Not only does it give you a program that's easier to build and debug, but it's also easier to understand and maintain, and that's where the financial value lies. This point can take some experience to understand, because it can appear that you're not being productive while you're making a piece of code elegant. Resist the urge to hurry; it will only slow you down.

2. **First make it work, then make it fast**.

This is true even if you are certain that a piece of code is really important and that it will be a principal bottleneck in your system. Don't do it. Get the system going first with as simple a design as possible. Then if it isn't going fast enough, profile it. You'll almost always discover that "your" bottleneck isn't the problem. Save your time for the really important stuff.

3. **Remember the "divide and conquer" principle**.

If the problem you're looking at is too confusing, try to imagine what the basic operation of the program would be, given the existence of a magic "piece" that handles the hard parts. That "piece" is an object---write the code that uses the object, then look at the object and encapsulate *its* hard parts into other objects, etc.

4. **Separate the class creator from the class user (*client programmer*)**.

The class user is the "customer" and doesn't need or want to know what's going on behind the scenes of the class. The class creator must be the expert in class design and write the class
so that it can be used by the most novice programmer possible, yet still work robustly in the application. Think of the class as a *service provider* for other classes. Library use will be easy only if it's transparent.

5. **When you create a class, attempt to make your names so clear that comments are unnecessary**.

Your goal should be to make the client programmer's interface conceptually simple. To this end, use method overloading when appropriate to create an intuitive, easy-to-use interface.

6. **Your analysis and design must produce, at minimum, the classes in your system, their public interfaces, and their relationships to other classes, especially base classes**.

If your design methodology produces more than that, ask yourself if all the pieces produced by that methodology have value over the lifetime of the program. If they do not, maintaining them will cost you. Members of development teams tend not to maintain anything that does not contribute to their productivity; this is a fact of life that many design methods don't account for.

7. **Automate everything**.

Write the test code first (before you write the class), and keep it with the class. Automate the running of your tests through a build tool---you'll probably want to use **ant**, the defacto standard Java build tool. This way, any changes can be automatically verified by running the test code, and you'll immediately discover errors. Because you know that you have the safety net of your test framework, you will be bolder about making sweeping changes when you discover the need. Remember that the greatest improvements in languages come from the built-in testing provided by type checking, exception handling, etc., but those features take you only so far. You must go the rest of the way in creating a robust system by filling in the tests that verify features that are specific to your class or program.

8. **Write the test code first (before you write the class) in order to verify that your class design is complete**.

If you can't write test code, you don't know what your class looks like. In addition, the act of writing the test code will often flush out additional features or constraints that you need in the class---these features or constraints don't always appear during analysis and design. Tests also provide example code showing how your class can be used.

9. **All software design problems can be simplified by introducing an extra level of conceptual indirection**.

This fundamental rule of software engineering1 is the basis of abstraction, the primary feature of object-oriented programming. In OOP, we could also say this as: "if your code is too complicated, make more objects."

10. **An indirection should have a meaning** (in concert with guideline 9).

This meaning can be something as simple as "putting commonly used code in a single method." If you add levels of indirection (abstraction, encapsulation, etc.) that don't have meaning, it can be as bad as not having adequate indirection.

11. **Make classes as atomic as possible**.

Give each class a single, clear purpose---a cohesive service that it provides to other classes. If your classes or your system design grows too complicated, break complex classes into simpler ones. The most obvious indicator of this is sheer size: if a class is big, chances are it's doing too much and should be broken up. Clues to suggest redesign of a class are:

- A complicated switch statement: consider using polymorphism.
- A large number of methods that cover broadly different types of operations: consider using several classes.
- A large number of member variables that concern broadly different characteristics: consider using several classes.
- Other suggestions can be found in *Refactoring: improving the design of existing code* by Martin Fowler (Addison-Wesley 1999).

12. **Watch for long argument lists**.

Method calls then become difficult to write, read, and maintain. Instead, try to move the method to a class where it is (more) appropriate, and/or pass objects in as arguments.

13. **Don't repeat yourself**.

If a piece of code is recurring in many methods in derived classes, put that code into a single method in the base class and call it from the derived-class methods. Not only do you save code space, you provide for easy propagation of changes. Sometimes the discovery of this common code will add valuable functionality to your interface. A simpler version of this guideline also occurs without inheritance: if a class has methods that repeat code, factor that code into a common method and call it from the other methods.

14. **Watch for *switch* statements or chained *if-else* clauses**.

This is typically an indicator of *type-check coding*, which means you are choosing what code to execute based on some kind of type information (the exact type may not be obvious at first). You can usually replace this kind of code with inheritance and polymorphism; a polymorphic method call will perform the type checking for you, and allow for more reliable and easier extensibility.

15. **From a design standpoint, look for and separate things that change from things that stay the same**.

That is, search for the elements in a system that you might want to change without forcing a redesign, then encapsulate those elements in classes. You can learn much more about this concept in *Thinking in Patterns with Java* at *www.BruceEckel.com*.

16. **Don't extend fundamental functionality by subclassing**.

If an interface element is essential to a class it should be in the base class, not added during derivation. If you're adding methods by inheriting, perhaps you should rethink the design.

17. **Less is more**.

Start with a minimal interface to a class, as small and simple as you need to solve the problem at hand, but don't try to anticipate all the ways that your class *might* be used. As the class is used, you'll discover ways you must expand the interface. However, once a class is in use you cannot shrink the interface without breaking client code. If you need to add more methods, that's fine; it won't break code. But even if new methods replace the functionality of old ones, leave the existing interface alone (you can combine the functionality in the underlying implementation if you want). If you need to expand the interface of an existing method by adding more arguments, create an overloaded method with the new arguments; this way you won't disturb any calls to the existing method.

18. **Read your classes aloud to make sure they're logical**.

Refer to the relationship between a base class and derived class as "is-a" and member objects as "has-a."

19. **When deciding between inheritance and composition, ask if you need to upcast to the base type**.

If not, prefer composition (member objects) to inheritance. This can eliminate the perceived need for multiple base types. If you inherit, users will think they are supposed to upcast.

20. **Use fields for variation in value and method overriding for variation in behavior**.

That is, if you find a class that uses state variables along with methods that switch behavior based on those variables, you should probably redesign it to express the differences in behavior within subclasses and overridden methods.

21. **Watch for overloading**.

A method should not conditionally execute code based on the value of an argument. In this case, you should create two or more overloaded methods instead.

22. **Use exception hierarchies**

Use exception hierarchies, preferably derived from specific appropriate classes in the standard Java exception hierarchy. The person catching the exceptions can then write handlers for the specific types of exceptions, followed by handlers for the base type. If you add new derived exceptions, existing client code will still catch the exception through the base type.

23. **Sometimes simple aggregation does the job**.

A "passenger comfort system" on an airline consists of disconnected elements: seat, air conditioning, video, etc., and yet you need to create many of these in a plane. Do you make private members and build a whole new interface? No---in this case, the components are also part of the public interface, so you should create public member objects. Those objects have their own private implementations, which are still safe. Be aware that simple aggregation is not a solution to be used often, but it does happen.

24. **Consider the perspective of the client programmer and the person maintaining the code**.

Design your class to be as obvious as possible to use. Anticipate the kind of changes that will be made, and design your class so that those changes will be easy.

25. **Watch out for "giant object syndrome."**

This is often an affliction of procedural programmers who are new to OOP and who end up writing a procedural program and sticking it inside one or two giant objects. With the exception of application frameworks, objects represent concepts in your application, not the application itself.

26. **If you must do something ugly, at least localize the ugliness inside a class**.

27. **If you must do something nonportable, make an abstraction for that service and localize it within a class**.

This extra level of indirection prevents the nonportability from being distributed throughout your program. (This idiom is embodied in the *Bridge* Pattern, among others).

28. **Objects should not simply hold some data**.

They should also have well-defined behaviors. (Occasionally, "data objects" are appropriate, but only when used expressly to package and transport a group of items when a generalized container is innappropriate.)

29. **Choose composition first when creating new classes from existing classes**.

You should only used inheritance if it is required by your design. If you use inheritance where composition will work, your designs will become needlessly complicated.

30. **Use inheritance and method overriding to express differences in behavior, and fields to express variations in state**.

An extreme example of what not to do is inheriting different classes to represent colors instead of using a "color" field.

31. **Watch out for *variance***.

Two semantically different objects may have identical actions, or responsibilities, and there is a natural temptation to try to make one a subclass of the other just to benefit from inheritance. This is called variance, but there's no real justification to force a superclass/subclass relationship where it doesn't exist. A better solution is to create a general base class that produces an interface for both as derived classes---it requires a bit more space, but you still benefit from inheritance, and will probably make an important discovery about the design.

32. **Watch out for *limitation* during inheritance**.

The clearest designs add new capabilities to inherited ones. A suspicious design removes old capabilities during inheritance without adding new ones. But rules are made to be broken, and if you are working from an old class library, it may be more efficient to restrict an existing class in its subclass than it would be to restructure the hierarchy so your new class fits in where it should, above the old class.

33. **Use design patterns to eliminate "naked functionality."**

That is, if only one object of your class should be created, don't bolt ahead to the application and write a comment "Make only one of these." Wrap it in a singleton. If you have a lot of messy code in your main program that creates your objects, look for a creational pattern like a factory method in which you can encapsulate that creation. Eliminating "naked functionality" will not only make your code much easier to understand and maintain, it will also make it more bulletproof against the well-intentioned maintainers that come after you.

34. **Watch out for "analysis paralysis."**

Remember that you must usually move forward in a project before you know everything, and that often the best and fastest way to learn about some of your unknown factors is to go to the next step rather than trying to figure it out in your head. You can't know the solution until you *have* the solution. Java has built-in firewalls; let them work for you. Your mistakes in a class or set of classes won't destroy the integrity of the whole system.

35. **When you think you've got a good analysis, design, or implementation, do a walkthrough**.

Bring someone in from outside your group---this doesn't have to be a consultant, but can be someone from another group within your company. Reviewing your work with a fresh pair of eyes can reveal problems at a stage when it's much easier to fix them, and more than pays for the time and money "lost" to the walkthrough process.

36. **In general, follow the Sun coding conventions**.

These are available at *http://www.oracle.com/technetwork/java/codeconv-138413.html* (the code in this book follows these conventions as much as I was able). These are used for what constitutes arguably the largest body of code that the largest number of Java programmers will be exposed to. If you doggedly stick to the coding style you've always used, you will make it harder for your reader. Whatever coding conventions you decide on, ensure they are consistent throughout the project. There is a free tool to automatically reformat Java code at: *home.wtal.de/software-solutions/jindent.*

37. **Whatever coding style you use, it really does make a difference if your team (and even better, your company) standardizes on it**.

This means to the point that everyone considers it fair game to fix someone else's coding style if it doesn't conform. The value of standardization is that it takes less brain cycles to parse the code, so that you can focus more on what the code means.

38. **Follow standard capitalization rules**.

Capitalize the first letter of class names. The first letter of fields, methods, and objects (references) should be lowercase. All identifiers should run their words together, and capitalize the first letter of all intermediate words. For example:

```
ThisIsAClassName
thisIsAMethodOrFieldName
```

Capitalize *all* the letters (and use underscore word separators) of **static final** primitive identifiers that have constant initializers in their definitions. This indicates they are compile-time constants. **Packages are a special case**---they are all lowercase letters, even for intermediate words. The domain extension (com, org, net, edu, etc.) should also be lowercase. (This was a change between Java 1.1 and Java 2.)

39. **Don't create your own "decorated" private field names**.

This is usually seen in the form of prepended underscores and characters. Hungarian notation is the worst example of this, where you attach extra characters that indicate data type, uses, location, etc., as if you were writing assembly language and the compiler provided no extra assistance at all. These notations are confusing, difficult to read, and unpleasant to enforce and maintain. Let classes and packages do the name scoping for you. If you feel you must decorate your names to prevent confusion, your code is probably too confusing anyway and should be simplified.

40. **Follow a "canonical form" when creating a class for general purpose use**.

Include definitions for **equals( )**, **hashCode( )**, **toString( )**, **clone( )** (implement **Cloneable**, or choose some other object copying approach, like serialization), and implement **Comparable** and **Serializable**.

41. **Use the JavaBeans "get," "set," and "is" naming conventions**

For methods that read and change **private** fields, even if you don't think you're making a JavaBean at the time. Not only does it make it easy to use your class as a Bean, but it's a standard way to name these kinds of methods and so will be more easily understood by the reader.

42. **For each class you create, include JUnit tests for that class**

You don't need to remove the test code to use the class in a project, and if you make any changes you can easily rerun the tests. This code also provides examples of how to use your class.

43. **Sometimes you need to inherit in order to access *protected* members of the base class**.

This can lead to a perceived need for multiple base types. If you don't need to upcast, first derive a new class to perform the protected access. Then make that new class a member object inside any class that needs to use it, rather than inheriting.

44. **Avoid the use of *final* methods for efficiency purposes**.

Use **final** only when the program is running, but not fast enough, and your profiler has shown you that a method invocation is the bottleneck.

45. **If two classes are associated with each other in some functional way (such as containers and iterators), try to make one an inner class of the other**.

This not only emphasizes the association between the classes, but it allows the class name to be reused within a single package by nesting it within another class. The Java containers library does this by defining an inner **Iterator** class inside each container class, thereby providing the containers with a common interface. The other reason you'll want to use an inner class is as part of the **private** implementation. Here, the inner class beneficial for implementation hiding rather than the class association and prevention of namespace pollution noted above.

46. **Anytime you notice classes that appear to have high coupling with each other, consider the coding and maintenance improvements you might get by using inner classes**.

The use of inner classes will not uncouple the classes, but rather make the coupling explicit and more convenient.

47. **Don't fall prey to premature optimization**.

This way lies madness. In particular, don't worry about writing (or avoiding) native methods, making some methods **final**, or tweaking code to be efficient when you are first constructing the system. Your primary goal should be to prove the design. Even if the design requires a certain efficiency, *first make it work, then make it fast*.

48. **Keep scopes as small as possible so the visibility and lifetime of your objects are as small as possible**.

This reduces the chance of using an object in the wrong context and hiding a difficult-to-find bug. For example, suppose you have a container and a piece of code that iterates through it. If you copy that code to use with a new container, you may accidentally end up using the size of the old container as the upper bound of the new one. If, however, the old container is out of scope, the error will be caught at compile time.

49. **Use the containers in the standard Java library**.

Become proficient with their use and you'll greatly increase your productivity. Prefer **ArrayList** for sequences, **HashSet** for sets, **HashMap** for associative arrays, and **LinkedList** for stacks (rather than **Stack**, although you may want to create an adapter to give a stack interface) and queues (which may also warrant an adapter, as shown in this book). When you use the first three, you should upcast to **List**, **Set** and **Map**, respectively, so that you can easily change to a different implementation if necessary.

50. **For a program to be robust, each component must be robust**.

Use all the tools provided by Java: access control, exceptions, type checking, synchronization, and so on, in each class you create. That way you can safely move to the next level of abstraction when building your system.

51. **Prefer compile-time errors to run-time errors**.

Try to handle an error as close to the point of its occurrence as possible.
Catch any exceptions in the nearest handler that has enough information to deal with them. Do what you can with the exception at the current level; if that doesn't solve the problem, rethrow the exception.

52. **Watch for long method definitions**.

Methods should be brief, functional units that describe and implement a discrete part of a class interface. A method that is long and complicated is difficult and expensive to maintain, and is probably trying to do too much all by itself. If you see such a method, it indicates that, at the least, it should be broken up into multiple methods. It may also suggest the creation of a new class. Small methods will also foster reuse within your class. (Sometimes methods must be large, but they should still do just one thing.)

53. **Keep things as "*private* as possible."**

Once you publicize an aspect of your library (a method, a class, a field), you can never take it out. If you do, you'll wreck somebody's existing code, forcing them to rewrite and redesign. If you publicize only what you must, you can change everything else with impunity, and since designs tend to evolve this is an important freedom. In this way, implementation changes will have minimal impact on derived classes. Privacy is especially important when dealing with multithreading---only **private** fields can be protected against un**synchronized** use.
Classes with package access should still have **private** fields, but it usually makes sense to give the methods of package access rather than making them **public**.

54. **Use comments liberally, and use the *javadoc* comment documentation syntax to produce your program documentation**.

However, the comments should add genuine meaning to the code; comments that only reiterate what the code is clearly expressing are annoying. Note that the typical verbose detail of Java class and method names reduce the need for some comments.

55. **Avoid using "magic numbers"**---which are numbers hardwired into code.

These are a nightmare if you need to change them, since you never know if "100" means "the array size" or "something else entirely." Instead, create a constant with a descriptive name and use the constant identifier throughout your program. This makes the program easier to understand and much easier to maintain.

56. **When creating constructors, consider exceptions**.

In the best case, the constructor won't do anything that throws an exception. In the next-best scenario, the class will be composed and inherited from robust classes only, so they will need no cleanup if an exception is thrown. Otherwise, you must clean up composed classes inside a **finally** clause. If a constructor must fail, the appropriate action is to throw an exception, so the caller doesn't continue blindly, thinking that the object was created correctly.

57. **Inside constructors, do only what is necessary to set the object into the proper state**.

Actively avoid calling other methods (except for **final** methods) since those methods can be overridden by someone else to produce unexpected results during construction. (See Chapter 7 for details.) Smaller, simpler constructors are less likely to throw exceptions or cause problems.

58. **If your class requires any cleanup when the client programmer is finished with the object, place the cleanup code in a single, well-defined method**---

…with a name like **dispose( )** that clearly suggests its purpose. In addition, place a **boolean** flag in the class to indicate whether **dispose( )** has been called so that **finalize( )** can check for "the termination condition" (see Chapter 4).

59. **The responsibility of *finalize( )* can only be to verify "the termination condition" of an object for debugging.** (See Chapter 4.)

In special cases, it might be needed to release memory that would not otherwise be released by the garbage collector. Since the garbage collector might not get called for your object, you cannot use **finalize( )** to perform necessary cleanup. For that you must create your own **dispose( )** method. In the **finalize( )** method for the class, check to make sure that the object has been cleaned up and throw a class derived from **RuntimeException** if it hasn't, to indicate a programming error. Before relying on such a scheme, ensure that **finalize( )** works on your system. (You might need to call **System.gc( )** to ensure this behavior.)

60. **If an object must be cleaned up (other than by garbage collection) within a particular scope, use the following idiom:**

Initialize the object and, if successful, immediately enter a **try** block with a **finally** clause that performs the cleanup.

61. **When overriding *finalize( )* during inheritance, remember to call *super.finalize( )*.**

This is not necessary if **Object** is your immediate superclass.
You should call **super.finalize( )** as the *final* act of your overridden **finalize( )** rather than the first, to ensure that base-class components are still valid if you need them.

**62. When you are creating a fixed-size container of objects, transfer them to an array**

**…**especially if you're returning this container from a method. This way you get the benefit of the array's compile-time type checking, and the recipient of the array might not need to cast the objects in the array in order to use them. Note that the base-class of the containers library, **java.util.Collection**, has two **toArray( )** methods to accomplish this.

63. **Choose *interfaces* over *abstract* classes**.

If you know something is going to be a base class, your first choice should be to make it an **interface**, and only if you're forced to have method definitions or member variables should you change it to an **abstract** class. An **interface** talks about what the client wants to do, while a class tends to focus on (or allow) implementation details.

64. **To avoid a highly frustrating experience, make sure that there is only one unpackaged class of each name anywhere in your classpath**.

Otherwise, the compiler can find the identically-named other class first, and report error messages that make no sense. If you suspect that you are having a classpath problem, try looking for **.class** files with the same names at each of the starting points in your classpath. Ideally, put all your classes within packages.

65. **Watch out for accidental overloading**.

If you attempt to override a base-class method and you don't quite get the spelling right, you'll end up adding a new method rather than overriding an existing method. However, this is perfectly legal, so you won't get any error message from the compiler or run-time system---your code simply won't work correctly.

66. **Watch out for premature optimization**.

First make it work, then make it fast---but only if you must, and only if it's proven that there is a performance bottleneck in a particular section of your code. Unless you have used a profiler to discover a bottleneck, you will probably be wasting your time. The hidden extra cost of performance tweaks is that your code becomes less understandable and maintainable.

67. **Remember that code is read much more than it is written**.

Clean designs make for easy-to-understand programs, but comments, detailed explanations, tests and examples are invaluable. They will help both you and everyone who comes after you. If nothing else, the frustration of trying to ferret out useful information from the JDK documentation should convince you.

# Chapter 6 – Java 5 and 6 Language Enhancements

## 6.1 Generics

J2SE 5.0 included the concept of generics or as it is known in the C++ world Templates. Generics enable the development of generic classes or methods that provide common functionality that can be used with multiple data types.

### 6.1.1 Interfaces or Generics

It is difficult to see, outside of container development, how generics really deliver a more elegant solution than traditional interface development. This is illustrated in Bruce Eckel's paper 3-10-04 Generics Aren't'.

Before considering using generics, ask yourself the question - Can what I am trying to achieve be accomplished using interfaces?

Typically the answer will be yes and an interface approach should be favoured as the subsequent code will be more readable to a wider Java audience.

That being said if generified classes do provide a cleaner implementation than interfaces there are naming conventions to follow when implementing them:

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Also note that a generic type may have multiple type parameters, but each parameter must be unique within its declaring class or interface. A declaration of Box<T,T>, for example, would generate an error on the second occurrence of T, but Box<T,U>, however, would be allowed.

**Use generics only when a cleaner implementation can be achieved over using interfaces.**

### 6.1.2 Working with Generic types

One area where generics do add value is in typecasting. Generics aim to prevent run-time ClassCastExceptions by enabling, through the <> notation, compile-time type checking.

Consider the following 1.4.2 Collection API code sample:

```
String str = "Item";
ArrayList list = new ArrayList();
list.add(0, str);
String item = (String)list.get(0);
```

The cast to String on the last line is necessary as the 1.4.2 Collection API uses the Object class to store the Collection objects, which means it cannot pick up type mismatches at compile time.

If an Integer had been added to the list instead of a String the first notification of a problem would have been at runtime as a ClassCastException.

The same example with the generified Collections library can be written as follows:

```
String str = "Item";
ArrayList<String> list = new ArrayList<String>();
list.add(0, str);
String item = list.get(0);
```

The <> notation has constrained the ArrayList to hold String values. This benefit is twofold as it is now no longer necessary to perform typecasting improving readability and any attempt to add a type other than String to this collection will result in a compile-time error.

**Use generified libraries to enforce compile-time type checking.**

## 6.2 Autoboxing & Unboxing of Primitive Types

Another refinement in J2SE 5.0 is Autoboxing and Auto-Unboxing of primitive types.

Converting between primitive types, like int, boolean, and their equivalent Object-based counterparts like Integer and Boolean, can require unnecessary amounts of extra coding, especially if the conversion is only needed for a method call to the Collections API (collections do not permit the use of primitive types).

In these instances the autoboxing and unboxing of Java primitives produces code that is more concise and easier to follow. Consider a collection that wishes to store ints:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = (list.get(0)).intValue();
```

This is no longer necessary as the required transition is managed by the compiler and can be written:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

Use autoboxing and unboxing only when there is an "impedance mismatch" between reference types and primitives.

The example above is a classic impedance mismatch. It is not appropriate to use autoboxing and unboxing for scientific computing, or other performance-sensitive numerical code. An Integer is not a substitute for an int; autoboxing and unboxing blur the distinction between primitive types and reference types, but they do not eliminate it.

## 6.3 For-Each Loop

The new For-Each loop included in J2SE 5.0 simplifies iterating over collections.
Traversing over a collection using an Iterator would look something like this:?

```
ArrayList<Integer> list =new ArrayList<Integer>();
for (Iterator i = list.iterator(); i.hasNext()) {
  Integer value = (Integer) i.next();
}
```

The iterator is just clutter. Furthermore, it is an opportunity for error. The iterator variable occurs three times in each loop: that is two chances to get it wrong. The for-each construct gets rid of the clutter and the opportunity for error. Here is how the example looks with the for-each construct:?

```
ArrayList<Integer> list = new ArrayList<Integer>();
for (Integer i : list)
{ ... }
```

The for-each construct is also applicable to arrays, where it hides the index variable rather than the iterator. The following example method returns the sum of the values in an int array:?

```
int sum(int[] a);
int result =0;
for (int i : a) result += i;
return result;
```

Use the for-each loop any time you can, it simplifies your code. The foreach loop is not usable for filtering. Similarly it is not usable for loops where you need to replace elements in a list or array as you traverse it. Finally, it is not usable for loops that must iterate over multiple collections in parallel.

### 6.3.1 Avoiding NoSuchElementExceptions with the For-Each Loop

Here is a common mistake made when trying to do a nested iteration over two collections using iterators:

```
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();

// BROKEN - throws NoSuchElementException
for (Iterator i = suits.iterator(); i.hasNext(); ) {
    for (Iterator j = ranks.iterator(); j.hasNext(); ) {
        sortedDeck.add(new Card(i.next(), j.next()));
    }
}
```

The problem is that the next method is being called too many times on the "outer" collection (suits). It is being called in the inner loop for both the outer and inner collections, which is wrong.
In order to fix it using iterators you have to add a variable in the scope of the outer loop to hold the suit:

```
for (Iterator i = suits.iterator(); i.hasNext(); ) {
  Suit suit = (Suit) i.next();
  for (Iterator j = ranks.iterator(); j.hasNext(); ) {
    sortedDeck.add(new Card(suit, j.next()));
  }
}
```

Whilst the NoSuchElementException has been corrected it is all a bit messy. The For Each loop on the other-hand is tailor made for nested iteration.

```
for (Suit suit : suits)
for (Rank rank : ranks)
sortedDeck.add(new Card(suit, rank));
```

## 6.4 Varargs

J2SE 5.0 supports varargs functionality allowing multiple arguments to be passed as parameters to methods. It requires the simple ... notation for the method that accepts the argument list.

```
void argtest(Object ... args) {
  for (int i=0;i <args.length; i++) { ... }
}

argtest("test", "data");
```

Use varargs, as a caller, whenever the API offers them e.g as in core APIs like reflection, message formatting, and the new printf facility.
As an API designer, you should use them sparingly, only when the benefit is truly compelling. Such as in a base DAO class accepting parameters for an encapsulated PreparedStatement.
Generally speaking, you should not overload a varargs method, or it will be difficult for programmers to figure out which overloading gets called.

## 6.5 Formatted Output (printf)

New to J2SE 5.0 is formatted output based on C printf-type functionality.

```
System.out.printf("name count%n");
System.out.printf("%s %5d%n", user,total);
```

Printf should not be used with the only exception being a phased migration of legacy C applications, as the same text layout can be preserved with little or no change.

## 6.6 Static Import

The static import feature, implemented as import static, enables you to refer to static constants from a class without needing to inherit from it. Instead of BorderLayout.CENTER each time we add a component, we can simply refer to CENTER as shown below:

```
import static java.awt.BorderLayout.*;

getContentPane().add(new JPanel(), CENTER);
```

Do not use static import it can make your program unreadable and un maintainable. Where multiple static imports have been applied readers of the code will not know which class a static member comes from.

## 6.7 Enumerated Types

J2SE 5.0 introduces an enumerated type.

```
public enum StopLight
  { red, amber, green }
;
```

Use an enumerated type in preference to static final constants when a fixed set of constants are required. Especially in the instance where it includes natural enumerated types such as, days of the week where you know all possible values at compile time.

## 6.8 Annotations

J2SE 5.0 introduced Annotations (meta-data) which enable programmers to decorate Java code with their own attributes. These attributes can be used for code documentation, code generation, code configuration and even during runtime, for providing special services such as logging and security.

### 6.8.1 Annotations vs Configuration Files

Annotations represent a powerful mechanism to embed configuration information in Java source code. The main advantage of this approach is that everything is in one place and the configuration information is associated directly with the Java component. In contrast annotations can clutter source code with all sorts of meta-data that is irrelevant to the actual program logic, interfering  with the readability of the code. Additionally, while annotations are ideal for metadata that relates to a particular component, they're not well suited to metadata with cross-component concerns.

Configuration files, on the other hand, can provide an organized view of the interrelationships across components in an application. Because they’re separate from the actual source code, they don't interfere with readability. The main drawback of configuration files is the separation of artefacts that need to be maintained in parallel with the source code with no obvious connection between the two.

**What type of configuration information needs to be represented?** Configuration information that is always going to be linked to a specific Java component is a good candidate to be represented by annotations. Annotations work especially well in this case when the configuration is core to the purpose of the code. Because of the limitations on annotations, it's also best when each component can only ever have one configuration. If you need to deal with multiple configurations, especially ones that are conditional on anything outside the class such as environment concerns, annotations may create more problems than they solve. Finally, it should be understood that annotations cannot be modified without recompilation so anything that needs to be reconfigurable at run time can't use annotations.

### 6.8.2 Annotations and Frameworks

The discussion so far has focused on writing custom annotations and what should be considered when determining their use. In practicality project teams will be utilizing a framework such as Spring or EJB 3.0.  These frameworks already provide annotations that implement those areas where annotations provide most benefit.

In the case of these frameworks annotations provide an alternative approach to using configuration files / deployment descriptors or a mixture of both.

**Be consistent in your approach.** Projects should define which approach will be adopted. If annotations are chosen over configuration files they should be used throughout. Consistency is key, as with all coding conventions and guidelines being consistent accelerates understanding.

One area where a mix of configuration files and annotations may apply is in unit testing. A project may mandate the use of configuration files for production code but annotations for unit tests. Using annotations in unit tests reduces the baggage accompanying a test artefact.