

Notes from Well House Consultants

These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit

*<http://www.wellho.net/net/whcotnl.html>
for details.*

1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

graham@wellho.net

technical contact

lisa@wellho.net

administration contact

Our full postal address is

404 The Spa

Melksham

Wiltshire

UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

Naming Conventions and Code Management

One of the most important aspects of coding is that your code should be easy for others to read and maintain later. This module examines "best practice" coding conventions ranging from variable and method naming through commenting standards to code structure issues.

<i>Within a class</i>	4
<i>The specification of the interface to a class</i>	5
<i>Grouping together classes into packages</i>	7
<i>Deploying multiple classes</i>	10
<i>Accessing shared classes</i>	12

Java is a language that is well suited to larger projects, where there can be substantial amounts of code spread across a number of different processors, all of which needs to be managed. At the same time, Java is a language in which the code author is encouraged (or even required!) to split the code down into relatively small, maintainable sections, each section being the definition of a class.

This module shows you how you'll manage your code, right through from the "micro" level within your class definitions to the "macro" level if you're deploying complete systems.

2.1 Within a class

Variable Naming

Many elements make up a Java program – variable names, method names, operators, constants, keywords, etc. In the Java specification, many of those elements (or tokens) of the language conform to the same rules of structure. In other words *abc123* could be a variable name, or a method name, or potentially a keyword, or a host of other things.

By adopting conventions for naming the various elements that conform to the same rules, code originators can make their work much easier for others to follow as development continues and enters the maintenance phase of the project. You might like to note:

- 80% of the lifetime cost of a piece of software goes to maintenance
- Hardly any software is maintained for its whole life by the original author

The following is Sun's recommended convention for naming variables. In other words you can break the suggestions if you really want, but it ain't good practice to do so!

- All instance, class, and class constants are in mixed case with a lower case first letter. Internal words start with capital letters. Variable names should not start with underscore `_` or dollar sign `$` characters, even though both are allowed.
- Variable names should be short yet meaningful. The choice of a variable name should be mnemonic, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throw-away" variables.

Common names for temporary variables are **i**, **j**, **k**, **m**, and **n** for integers; **c**, **d**, and **e** for characters.

Examples:

```
float waterVolume;
int numberOfStudents;
int j,k;
char c;
```

The following are keywords, and may not be used for variable (or any other) names:

abstract	double	int	strictfp **
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	package	throw
char	for	private	throws
class	goto *	protected	transient
const *	if	public	try
continue	implements	return	void
default	import	short	volatile
do	instanceof	static	while

*Important to note: * indicates a keyword that is not currently used; ** indicates a keyword that was added for Java 2*

The words **false**, **null** and **true** aren't technically reserved words, but for obvious reasons you are advised not to use them as variable names.

Constant naming

The names of variables declared class constants and of ANSI constants should be all upper case with words separated by underscores `_`.¹

Examples:

```
static final int MAX_AGE = 120;
static final String COPYRIGHT_STATEMENT = "Copyright Well House Consultants, 2001";
```

Commenting

First and foremost your code should be well commented. Java includes two comment structures

```
/* through to */
```

which can span over multiple lines of the source code and

```
// through to the line end
```

which allows for single line comments, and comments to the right of live code.

We can't stress enough how important it is to comment your code well; typically, describe each method or major block of code in a comment that stands out from the code, and embed shorter comments in the code to document things which won't be obvious to a maintainer later on.

Sun's conventions include commenting suggestions. It's suggested that `/*` to `*/` comments be structured as follows:

```
/*
 * The comment starter should be on a line on its own
 * and each subsequent line should have a * on the
 * left hand side to indicate that the comment continues.
 *
 * After the end of the comment, the comment terminator
 * should also be on a line on its own as in ...
 */
```

It's further noted that `//` may be used down the left margin to comment out a whole series of lines of code, and that short `//` comments are allowed on the ends of lines. Such comment should be offset to the right so that they're clear of the main body of the code. Where there's more than one such comment within a few lines of code, they should be vertically aligned.

A comment that starts `/**` and ends `*/` is treated by Java itself in exactly the same way as a `/*` to `*/` comment; however, a separate utility supplied with the JSDK (javadoc) extracts all such comments in order to provide an automated documentation facility.

Further coding standards

Sun's suggested conventions go well beyond token naming and the structure of comments; see:

<http://java.sun.com/docs/codeconv>

2.2 The specification of the interface to a class

Whilst it's good practice to use Sun's conventions for variables internal to a class, and private internal method names, it's only the developer or maintainer of the class that will see them later. It becomes much more important to use conventions for the

¹ ANSI constants should be avoided, for ease of debugging

various elements of a class that can be seen and used from outside that class. Not only must these external elements be seen and understood by users at the next level up, but they'll also need to be built into the code at that next level. Unlike internal names, they'll propagate up the tree.

Class naming

Recommendations for class names are as follows:

- Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words, avoiding acronyms and abbreviations.¹
- Interface names should be capitalized like class names.

Examples:

```
class Cat;
class PedigreeCat;
interface Animal;
class FTPConnectionInformation;
```

Method Naming

Recommendation is as follows:

- Methods should be verbs, in mixed case with the first letter lower case, and the first letter of each internal word capitalized.

Examples:

```
traverse();
resolveServerIP();
setColour();
getColour();
initialise();
```

Instance variable naming

Where an instance variable is declared so that it's visible outside of the class in which it's defined, it's important to follow the variable naming convention for regular variables, as already covered.

Bean-able and other standards

Method, class and variable names are just one of many aspects of designing a class. If you're designing a class, you must remember that the externally visible elements will be built into a whole host of other pieces of software, and you should:

- Design the interface in such a way that it will meet the needs of all users
- Design the interface so that it can cope with future expansion requirements
- Design the interface to be expandable if you don't know what future requirements will be
- Avoid providing too many methods or variables as they might increase support costs later
- Be consistent

A consistent interface sticks either to UK or US conventions; for example, you shouldn't write methods called `setColour` and `colorChange` as that would confuse your user. Other considerations on naming consistency.

center	v	centre
prioritise	v	prioritize
disc	v	disk

You should decide if you're going to abbreviate common words. Technically, recommendations are that you should use words like "information" and "initialise" in

¹ unless the abbreviation is much more widely used than the long form, such as URL or HTML

your names, but in practice (even in Sun's standards), you'll find "info" and "init" used.

Consistency should follow through all the classes that are written or might be used on a project. Other elements of that consistency include:

- Always having a constructor that just sets up an object, and another method to populate the object (or just one method if you prefer that to be your standard)
- Having further conventions of method naming, such as those suggested by Java Beans

It's an excellent idea for you to use the conventions that are provided within the Java Bean API. Briefly, Java Beans provide a standardised convention for naming methods in a class such that the class can be used within a beanbox for development purposes, or within an environment such as JSP (Java Server Pages). Using Java Bean conventions, you'll have

- A constructor that doesn't require any parameters
- Methods with names starting "set" to put values in to an object
- Methods with names starting "get" to read values back from an object
- Methods with names starting "is" to test a boolean property

Even if you're not intending that your code run within an environment that requires it to be "beanified", the naming conventions defined are good ones and many people understand them.

For future-proofing, we cannot stress how important it is that you define your class's public persona in such a way that it will not need to have features removed or changed. It's reasonable to create a class into which extra methods and variables will later be added.¹

Realistically, there's no way of knowing at the start of the life of a class how things will develop, and the maintainer may come to regret the provision of certain methods in the years following its writing. Such methods should not be removed once you have a user base out there using your class, but you may provide additional (and better?) alternative methods, and then describe the original method as "deprecated" in your documentation.

As an example of deprecation, the idea of Java beans hadn't been thought of when Java 1.0 was released, and some of the applet and abstract windowing toolkit classes included methods with names such as "size". When beans were introduced, there was a requirement for members of such classes to be handled by bean containers, and extra methods such as `getSize` were added. The original method (`size`) still exists, but it's deprecated. It can't be used in a beanbox, but it is there in case there's older application code around that needs it.

The alternative of removing the method is too dreadful to think about. It would mean that backwards compatibility would be broken, and that every application would have to be rechecked against the new release of the class.

2.3 Grouping together classes into packages

Having studied naming conventions of classes, let's move on and have a look at their grouping.

A collection of classes which will be used in close association with one another is known as a "package" in Java. Very often, such classes will be related to each other in that some of the classes will be subclassed from others, but that's not necessarily the case. A package can include not only classes, but also other packages.

In Java, packages are arranged into a hierarchy. A tree structure very similar to it is used for directories and files. The separator between each level of the hierarchy is a dot ... so for example, you might write

¹ although any objects that have been saved out via a serialized interface will be made incompatible by you doing such an extension

```

    europe.unitedkingdom.health.emergency.AmbulanceService
to access the AmbulanceService class within the emergency package within the
health package, and so on. Other classes in the same package might include
    europe.unitedkingdom.health.emergency.AccidentDepartment
and    europe.unitedkingdom.health.emergency.Worker
and    europe.unitedkingdom.health.emergency.Paramedic
with this last being a subclass of
    europe.unitedkingdom.health.emergency.Worker
and so on.

```

Naming conventions

Although the naming example we've looked at just above would work, the package names and structure may well turn out to be non-unique on a worldwide basis. That might not matter too much with an organisation's private classes but as computing becomes much more network-bound, packages are going to be shared, and there's now a convention that you should name your packages based on your domain name.

Sun's standard suggests:

- The prefix of a unique package name is always written in all-lower case ASCII letters and should be one of the top-level domain names, currently *com*, *edu*, *gov*, *mil*, *net*, *org*, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981.
- Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.
- No doubt that will be revised soon to include ".info" and other top-level domains. Examples of class names that conform to the suggestions:

```

net.wellho.examples.Wellreader
net.wellho.website.clientside.Tickertape
com.sun.eng.Sparctest
uk.ac.ox.oucs.helpdesk.Call

```

Two top-level "pseudo-domain" packages are also worthy of note:

java. is the package from which standard Java classes available in regular JREs are located - sub packages include:

```

java.lang
java.io
java.util
java.text
java.net

```

javax. is the package which is used for packages which are in the Java 2 Enterprise edition - it stands for java eXtended. Examples:

```

javax.servlet
javax.swing
javax.mail

```

Imports

Fully qualified class names (i.e. class names that include the full package structure) can get very long, and if you have to use a whole series of them in the same class, the coding can get very repetitive.

Java's `import` statement allows you to short cut the full naming of a package. Let's say, for example, that you wanted to use a whole series of classes from within the package

```
javax.servlet
```

Whilst you COULD code:

```
public class MyServlet extends javax.servlet.HttpServlet {
protected void doPost (javax.servlet.http.HttpServletRequest request,
javax.servlet.http.HttpServletResponse response)
```

and so on, how much clearer and easier it would be to write:

```
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {
protected void doPost (HttpServletRequest request,
Response response)
```

Import statements "cost" nothing at run time – the statement really creates a list of package names from which classes can be loaded without them being fully qualified. It does NOT load in the whole packages¹ as soon as it's encountered.

You'll find many classes that start with a whole series of imports – even some of the more complex examples in text books have ten or more imports – but you should be careful not to overdo it. If any two of the packages you import define classes of the same name, then at very least you'll be creating a nightmare for later code maintainers who, in any case, will be asking themselves "...and where does *this* class come from then?" with a large number of imports.

The four Ps

There are two philosophies that you can apply with regard to access to classes when you design a programming language. The first is that you assume that your programmers know what they're doing, and you trust them not to intentionally step on each other's namespaces. Languages such as Perl work in this way. For sure, they provide tools that keep namespaces apart, but there's nothing to stop a programmer reaching out from one namespace into another to access a variable / subroutine / method therein.

The second approach is to build a security model around your classes, and provide a very restrictive set of defaults. It's then up to the programmer writing the class to use keywords in his code to open up access as necessary.

There are four access levels in Java code, as follows:

- If something is declared "private", then it's only accessible from within the class in which it is declared.
- If something has no declaration, it's said to have "package" privileges. You don't actually use the word "package" (that keyword is used for something else), but it will be accessible from within the class in which it's declared, and also from within the other classes in the same package.
- The next level of access is "protected". If something is protected, it is accessible
 - from within its own class
 - from within its own packages
 - from within sub-packages of the current package.

It is NOT accessible from other packages apart from sub-packages, i.e. it's protected from above in the tree. We think of this being protected, rather like an umbrella protects you from the rain.

- The final access level is "public". A public declaration means that the item being declared doesn't have accesses to it restricted by the declaration statement, though it is possible that it might not be accessible for other reasons.

¹ or classes from within it, if you specify class by class

So: **private - package - protected - public**

Think now to "real life". Let's take your bank. To all intents and purposes, it's a public place. Anyone can wander up the high street during the day, walk into that public place, have a chat with the staff, and transact their business. But let's say that you walk into your bank one day and find that there's a long queue. Can you simply nip over the counter, help yourself to enough cash your existing account can cover, write an IOU, and leave? No, you can't. In other words, although the bank is a public place by our definition, it also has more private parts.

The same two-tier approach applies to variable and methods within classes. Classes can be declared as public or protected, or default to package. In addition, each variable and method in the class can be declared as public, protected or private, and can default to package.

In order for you to write code to access a method in a class, you must have sufficient a level of access to the method and also to the class itself.

2.4 Deploying multiple classes

Directories and files

The hierarchy of packages and classes is the same as the hierarchy of directories and files as used in modern operating systems. One way to install a set of Java packages and classes is, indeed, to keep them as a series of files and directories under the main application directory, and developers will typically use this approach.

Jars

When it comes to handling a large number of classes, having each in its own separate file isn't very efficient, and if an application (with all its classes) is being distributed over a network, it's preferable to do a single transfer.

For many years, larger distributions have been made in the form of single composite files. The tar file has become something of a standard. "tar" stands for "tape archive", and is good as far as it goes, but it has certain shortcomings.

Java classes and packages are often distributed, not through a tar file, but through a .jar file.

Jar files contain one or more class files (or other files if you wish), held in a jar so they can be distributed as a single entity. The jar file is automatically indexed so that individual items can be extracted quickly, and it's possible to ask the `jar` utility (used to create and extract from jar files) to compress and decompress the individual files. Options to `jar` are similar to options to `tar`.

Let's see some examples:

Create a jar archive from all the .class files in a directory.

```
bash-2.04$ cd java/answers
bash-2.04$ jar cvf ~/answers.jar *.class
added manifest
adding: alvechurch.class(in = 2043) (out= 1197)(deflated 41%)
adding: another.class(in = 2724) (out= 1466)(deflated 46%)
adding: bagpipes.class(in = 628) (out= 437)(deflated 30%)
adding: banjo.class(in = 1595) (out= 926)(deflated 41%)
(etc)
adding: xylophone.class(in = 1517) (out= 968)(deflated 36%)
adding: zither.class(in = 1683) (out= 1005)(deflated 40%)
bash-2.04$
```

That example was, by default, compressed. Certain tools only work directly from a jar file if it isn't compressed, and that can be created using an additional 0 (zero) option:

```
bash-2.04$ jar 0cvf ~/answersfull.jar *.class
added manifest
adding: alvechurch.class(in = 2043) (out= 2043)(stored 0%)
adding: another.class(in = 2724) (out= 2724)(stored 0%)
adding: bagpipes.class(in = 628) (out= 628)(stored 0%)
adding: banjo.class(in = 1595) (out= 1595)(stored 0%)
(etc)
adding: xylophone.class(in = 1517) (out= 1517)(stored 0%)
adding: zither.class(in = 1683) (out= 1683)(stored 0%)
bash-2.04$
```

which does result in a larger file:

```
bash-2.04$ ls -l ans*
-rw-rw-r-- 1 trainee graham 55389 Oct 21 15:56 answers.jar
-rw-rw-r-- 1 trainee graham 86876 Oct 21 15:57 answersfull.jar
bash-2.04$
```

The key options to jar are:

- c** create (i.e. write a file)
- t** tell (a.k.a tabulate; look, but don't extract)
- x** extract

You'll normally specify the **f** option as well as to give a file name rather than use the default (which is to use **stdin** or **stdout**).

Let's see what's in our jar:

```
bash-2.04$ jar tvf ../answers.jar
 0 Sun Oct 21 15:56:52 BST 2001 META-INF/
68 Sun Oct 21 15:56:52 BST 2001 META-INF/MANIFEST.MF
2043 Sun Jul 09 15:33:36 BST 2000 alvechurch.class
2724 Sun Jul 09 15:33:36 BST 2000 another.class
 628 Sun Jul 09 15:33:38 BST 2000 bagpipes.class
1595 Sun Jul 09 15:33:38 BST 2000 banjo.class
(etc)
1517 Sun Jul 09 15:33:38 BST 2000 xylophone.class
1683 Sun Jul 09 15:33:38 BST 2000 zither.class
bash-2.04$ ls
```

You've noticed the manifest, no doubt? Let's extract the whole archive:

```
bash-2.04$ tar xvf ../answers.jar
tar: This does not look like a tar archive
tar: Skipping to next header
tar: 93 garbage bytes ignored at end of archive
tar: Error exit delayed from previous errors
bash-2.04$ jar xvf ../answers.tar
java.io.FileNotFoundException: ../answers.tar (No such file or directory)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:64)
  at sun.tools.jar.Main.run(Main.java:186)
  at sun.tools.jar.Main.main(Main.java:904)
bash-2.04$
```

Oops, **tar** and **jar** are not the same! Let's try again:

```
bash-2.04$ jar xvf ../answers.jar
  created: META-INF/
  extracted: META-INF/MANIFEST.MF
  extracted: alvechurch.class
  extracted: another.class
  extracted: bagpipes.class
  extracted: banjo.class
  (etc)
  extracted: xylophone.class
  extracted: zither.class
bash-2.04$
```

And to prove that those class files "work":

```
bash-2.04$ java warwick
Reading URLs
Got http://seal/moredata.txt
Length in bytes 172
Length in lines 4
Got http://seal/index.html
Length in bytes 2472
Length in lines 57
URLs Grabbed
lengths 2472 and 172
index.html is larger
bash-2.04$
```

2.5 Accessing shared classes

The Java Interpreter (i.e. the Java Virtual Machine "Java") knows where to find system classes and also, from release 1.2, where to find extensions in the **javax** directory. It needs to be told, though, where to find other classes that comprise the application that's to be run.

CLASSPATH and CLASSDIR

The **CLASSPATH** environment variable tells Java where to look for classes in addition to the system directories which (in more recent versions) are already defined in the installation. It's set and changed as you would set any other environment variable on your system (systems dependent command); our example uses bash under Linux.

First example, using a regular setting that we happen to have in use, looking for a class that's nowhere on that path:

```
bash-2.04$ echo $CLASSPATH
.: /home/javatools/jdk1.3/lib/tools.jar:/home/javatools/j2sdskeel1.3/lib/
j2ee.jar
bash-2.04$ java fife
Exception in thread "main" java.lang.NoClassDefFoundError: fife
bash-2.04$ HOLD=$CLASSPATH
```

now adding in a directory ...

```
bash-2.04$ export CLASSPATH=$CLASSPATH:myjavadir
bash-2.04$ java fife
First Temperature = 20
Second Temperature = 30
Final Temperature = 27
Average Temperature: 25.666666666666668
bash-2.04$
```

Loading in classes from a .jar file ...

```

bash-2.04$ export CLASSPATH=$HOLD
bash-2.04$ java fife
Exception in thread "main" java.lang.NoClassDefFoundError: fife
bash-2.04$ export CLASSPATH=$CLASSPATH:answersfull.jar
bash-2.04$ java fife
First Temperature = 22
Second Temperature = 21
Final Temperature = 17
Average Temperature: 20.0
bash-2.04$ export CLASSPATH=$HOLD
bash-2.04$ java fife
Exception in thread "main" java.lang.NoClassDefFoundError: fife
bash-2.04$ export CLASSPATH=$CLASSPATH:answers.jar
bash-2.04$ java fife
First Temperature = 22
Second Temperature = 32
Final Temperature = 21
Average Temperature: 25.0
bash-2.04$

```

On Windows systems, the command to set the CLASSPATH is slightly different; you use a semicolon rather than a colon between elements. Here's an example:

```
set CLASSPATH=uni.jar;C:\gje\extra.jar;C:\gje\javaliblocal
```

java has changed since it was first released. Our examples use a modern (1.2 or later) version. In earlier versions, the variable in which extra directories were to be found was called **CLASSDIR** rather than **CLASSPATH**, and also in earlier versions, **java** could only read classes from jars if the jar was uncompressed.

You'll typically have a whole series of directories, jars, etc., in which you hold your classes, and want to access them from many applications. Thus, it's the norm to set up your CLASSPATH in your initialisation files. For example, we use *.bashrc*. If, exceptionally, you want to make a "single shot" test using classes that are off your normal path, you can use the **-classpath** option to the Java Virtual Machine, which can also be shortened to **-cp**; these options are available from **java** 1.2.

Other Virtual Machines

For users of other virtual machines, the rules differ. In a browser, for example, the **CLASSPATH** doesn't apply. Instead, the class will be loaded using information specified within the HTML. The recommended tag is the **OBJECT** tag, although the **APPLET** tag which is now deprecated remains popular and is similar in operation.

Within **OBJECT**, you can specify:

- | | |
|-----------------|--|
| CLASSID | to give the name of the class to be used from the server. The URL can be relative (to the current server directory) or absolute. |
| or | |
| CODEBASE | Specify the base URL relative from which applet code is to be loaded |
| CODE | Specify the class name required |
| or | |
| ARCHIVE | Specify a list of URLs that are to be downloaded before the object (applet in our example) is rendered. |

Archive is typically used for performance reasons to save the need to download a whole series of separate classes each under a separate connection.



Exercise

Take an application from a previous exercise and bundle all the necessary files for it to run into a jar.

Create a new directory and copy the jar file to that new directory.

Set your CLASSPATH so that your Java run time environment looks at the newly relocated jar.

Run the application from the new location. Delete some of the original .class files if you want to make sure it really is running from that new location.

License

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log
Original Version, Well House Consultants, 2004

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

License Ends.