

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

Exceptions

Your program is affected by the environment around it. Its performance will vary depending on aspects as diverse as user inputs and the availability (or lack) of a network service. Java provides you with exceptions, which are a mechanism through which you trap and handle behaviours that differ from the norm.

<i>"trying" and "catching"</i>	4
<i>"throwing"</i>	9
<i>"finally"</i>	11
<i>Defining your own exceptions</i>	12

By now, you'll certainly have made a few mistakes in the example code you have written.

Some of the errors will be flagged as you compile your program. Something is wrong with the syntax of your program so that the Java compiler does not understand what you want your code to do, or you might be asking for the impossible.

Example:

```
seal% javac corsham.java
corsham.java:12: ';' expected.
    card_7 [] play = new card_7 [26] :
                                ^
1 error
seal%
```

The solution in such circumstances is to correct or modify your code and try again. If it compiles correctly you can rest assured that you have used the right syntax to define your class. It is not a guarantee, however, that your class will perform in the way you intended; you may have made still other errors.

The program examples provided so far have been very careful to check for certain errors as they run. The code has been carefully engineered to ensure that we do not attempt to run off the end of an array, for example.

We'll show a deliberate error. As a demonstration, we have shortened an array in one of the previous examples:

```
seal% javac wadswick.java
seal% java wadswick
Short pack of 20 cards
And a set of cluedo suspect cards!
java.lang.ArrayIndexOutOfBoundsException: 15
    at wadswick.main(Compiled Code)
seal%
```

The compile ran fine; there is nothing wrong with the syntax.

When the program ran, though, we tried to access element number 15 of an array which had elements numbered 0 through 14.

One way to handle such errors is to add extra tests in your program, and to some extent this is done in all programming languages.

Java also provides a mechanism known as **exceptions** for handling such situations.

2.1 "trying" and "catching"

Before we describe how exceptions work, we should make it clear that they are a vital part of the Java language and not a mechanism to catch sloppy programming.

The example we are working with could easily be corrected by resizing an array, but many times **exceptions** are **thrown** for reasons outside the control of the Java programmer:

- A user types in the word "three" when asked for a number
 - A program attempts to open a data file in a directory which the user cannot read
- In practice, the exception mechanism is so much better than adding "hundreds" of checks to your program that you'll even end up throwing your own exceptions.
- A card cannot be described as "the n of xxx" - oops - an exception. The **exception** will be **caught** by the special code that deals with the Joker!

So you think that something might not work in a block of code. For example, you are concerned about trying to hold too many cards in your hand.

You will **try** to pick up a card from the deck, but if there isn't room in your hand, an **exception** will be **thrown**. You will **catch** the **exception** with a piece of code written to handle it.

Let's do that with the example we started with. Here is the original (failing) code:

```
// Well House Consultants 2004.
/** Array too short - shows problem */

public class wadswick{
    public static void main(String[] args){
        System.out.println("Short pack of 20 cards");
        System.out.println
            ("And a set of cluedo suspect cards!");
        card_7 [] play = new card_7 [15] ;
        int cardcount = 0;
        for (int k=0;k<6;k++)
            play[cardcount++] = new suspect_card_7(k);
        for (int k=33;k<53;k++)
            play[cardcount++] = new playing_card_7(k);

// And print out cards!
        for (int k=0;k<26;k++){
            System.out.println(play[k].getcardname());
        }
    }
}
```

Here's the code with the code to handle the exception:

```
// Well House Consultants 2004.
/** First Exception */

public class neston

{
public static void main(String[] args)
{
System.out.println("Short pack of 20 cards");
System.out.println
    ("And a set of cluedo suspect cards!");
card_7 [] play = new card_7 [15] ;
int cardcount = 0;
try
    {
    for (int k=0;k<6;k++)
    play[cardcount++] = new suspect_card_7(k);
    for (int k=33;k<53;k++)
    play[cardcount++] = new playing_card_7(k);
    }
catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println
    ("Hand full. Excess cards dropped");
    cardcount--;
    }

// And print out cards!
    for (int k=0;k<cardcount;k++)
        {
        System.out.println(play[k].getcardname());
        }
}
}
```

Figure 1 Running public class neston

```
seal% java neston
Short pack of 20 cards
And a set of cluedo suspect cards!
Hand full. Excess cards dropped
Col. Mustard
Prof. Plum
Rev. Green
Mrs. White
Miss Scarlett
Mrs. Peacock
10 of Clubs
10 of Diamonds
10 of Hearts
10 of Spades
Jack of Clubs
Jack of Diamonds
Jack of Hearts
Jack of Spades
Queen of Clubs
seal%
```

How does it work?

A block of code where "something may go wrong" is tried so that:

```
for (int k=0;k<6;k++)
    play[cardcount++] = new suspect_card_7(k);
for (int k=33;k<53;k++)
    play[cardcount++] = new playing_card_7(k);
```

became:

```
try
{
for (int k=0;k<6;k++)
    play[cardcount++] = new suspect_card_7(k);
for (int k=33;k<53;k++)
    play[cardcount++] = new playing_card_7(k);
}
```

Directly after the try block, we have added a **catch** block which will be performed if something goes wrong:

```
catch (ArrayIndexOutOfBoundsException e)
{
System.out.println
    "Hand full. Excess cards dropped");
cardcount--;
}
```

Notice:

- The exception has been thrown by Java. We have not had to use a **throw** statement.
- Once the exception was thrown, we jumped out of the **try** block to the **catch** block. We did not attempt to carry on creating more cards and throwing multiple exceptions.
- Upon completion of the execution of the **catch** block, our method continued to run at the following statement.

"Throwing" and "Catching" sounds like a ball game, doesn't it? What type of "thing" is a ball? It's an object. And in just the same way, exceptions are objects.

Like many objects, there are sets and subsets of exceptions.

In our example, we were very specific about the exception we wanted to catch. The hierarchy in this case is:

- **Exception**
- **RuntimeException**
- **ArrayIndexOutOfBoundsException**

And in this case we could have written our **catch** block with any of those declared as the parameter.

We looked for a single specific exception in our first example but we could easily look for several by adding extra **catch** blocks, and/or by catching exceptions at a higher level in the hierarchy.

If you have several **catch** blocks, the order is important. It is pointless to code:

```
try
{
}
catch (Exception e)
{
}
catch (RuntimeException e)
{
}
```

since the first **catch** block always catches all exceptions (including Runtime exceptions).

We've seen how you can handle exceptions within a method. In essence, if you found a problem, you have dealt with it.

2.2 "throwing"

There will be occasions when you want your method to pass back the fact that it encountered a problem, perhaps by passing back the exception.

If you specify that a method can throw one or more exceptions, and then take no action to handle those exceptions within the method, the exceptions will be passed back to the calling method for it to deal with.

As an example, let's add a `getsuitname` method to our `playing_card` class:

```
public String getsuitname()
    throws ArrayIndexOutOfBoundsException
{
    int suit = (value-1) % 4;
    if (value < 1) suit = -1;
    String [] suites = { "Clubs", "Diamonds",
                        "Hearts", "Spades" };
    return suites[suit];
}
```

and notice how this new method is specified as throwing an `ArrayIndexOutOfBoundsException`.

Recall that a Joker was card number 0? For the purpose of this exercise, we have engineered our method to ensure it throws an exception if it finds a Joker.

Our calling method can, in turn, deal with the exception, or choose to throw it on to its calling method. Let's deal with it ...

```
// And print out cards!

for (int k=0;k<cardcount;k++)
{
    try {
        System.out.println(play[k].getsuitname());
    }
    catch (Exception e) {
        System.out.println("Problem finding suit");
    }
}
}
```

```
seal% java atworth
Short pack of 20 cards
Hand full. Excess cards dropped
Problem finding suit
Clubs
Diamonds
Hearts
Spades
Clubs
Diamonds
Problem finding suit
Spades
Clubs
Diamonds
Hearts
Spades
Clubs
Diamonds
seal%
```

Figure 2 Running public class atworth

If the suit is returned correctly, it is printed out. If an exception is thrown (any exception in this example), a problem message is printed instead.

Our *atworth.java* main method includes a line of code which replaces one of the cards with a Joker, and one card is a suspect card. We'll look at that next.

We had to engineer our method to cause it to fail so that the exception was thrown. Can we throw our own exceptions?

Yes, we can. We'll illustrate that in the *getsuitname* method that we have provided for the *suspect_card* class.

Clearly, suspect cards never have a suit, so we always want this method to throw an exception.

Here's the method:

```
public String getsuitname()
    throws ArrayIndexOutOfBoundsException
{
    ArrayIndexOutOfBoundsException e =
        new ArrayIndexOutOfBoundsException();
    throw e;
}
```

Easy really, just create an exception object ... and throw it!

As well as throwing a new exception, you can elect to **rethrow** an exception within a catch block. You would do this if you need to tidy up (an error) in your method, but also tell the calling method that it needs to tidy up.

2.3 "finally"

Recall in the handling of exceptions:

```
try
catch
etc ...
```

On some occasions, you may have code that you want to be executed before a method exits no matter what exceptions were thrown. For example, closing a file or releasing a connection to a remote system.

You may put any such code in a **finally** block, after the last of your **catch** blocks. Such a block of code will be executed when the execution of the **try** block is completed, regardless of whether any exceptions were thrown, or a **return** or **break** statement was executed.

The following example shows a **finally** block which is run whether or not an exception was caught, and is even run if the exception returns from the method.

```
// And print out cards!
int probcount=0;
for (int k=0;k<cardcount;k++)
{
    try {
        System.out.print(play[k].getsuitname());
    }
    catch (Exception e) {
        System.out.print("Problem finding suit");
        if (++probcount > 1) return;
    }
    finally
    {
        System.out.print(".\n");
    }
}
}
```

```
seal% java whitley
Short pack of 20 cards
Hand full. Excess cards dropped
Problem finding suit.
Clubs.
Diamonds.
Hearts.
Spades.
Clubs.
Diamonds.
Problem finding suit.
seal%
```

Figure 3 Running public class whitley

2.4 Defining your own exceptions

You can define your own exceptions and exception classes if you wish. Indeed, that would have been a good idea in our suit example rather than for us to take over one of the built-in exceptions classes!

As all exceptions are derived from a "throwable" base class, a number of methods are available to you, including:

getMessage() which returns the message describing the current exception
printStackTrace() which outputs to standard error the stack trace where the error message was generated

And remember that these methods are available on standard exceptions, your own exceptions, and any exceptions provided by other classes you have elected to use.

Exercise

Use the `read_word` method in our `wellreader` class to read a string of text. Call the static `parseInt` method to extract an integer value from the line. `parseInt` is a static method in the `java.lang.Integer` class. It takes a `String` parameter, and returns an `int` primitive.

- Deal with any `NumberFormatException` that might be thrown ... give the user another chance!

If you are new to programming, you might like to do this exercise in two stages ... firstly write a program to prompt for and read a `String` and extract an integer from it, **IGNORING** any errors in the user's entry. Such a program might read:

```
public class E1 {
    public static void main (String [] args) {
        System.out.print ("Please enter an integer ... ");
        String Said = WellHouseInput.readLine();
        int j = Integer.parseInt(Said);
        System.out.println ("That was "+j);
    }
}
```

Then extend the program to trap the error and reprompt if necessary.

Our example answer is E2

```
seal% java E2
give me a number please 23.5
Expecting an integer. Try again: seven
Expecting an integer. Try again: 73
I got 73
seal% java E2
give me a number please: -34
I got -34
seal%
```

For Advanced Students

Returning to the weather station example, can you make the `Field Station` class return an exception if the calling method asks for the pressure? (You may recall this was a non-existent variable in the sub-class, and temporarily we returned a "-1" value to indicate an error.)

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 _____

License Ends.