9. Creating And Using Exceptions

Introduction

If the reader has written Java programs that make use of the file handling facilities they will have written code to catch exceptions i.e. they will have used try\catch blocks. This chapter explains the importance of creating your own exceptions and shows how to do this by extending the Exception Class and using the 'Throw' mechanism.

Objectives

By the end of this chapter you will be able to….

- Appreciate the importance of exceptions
- Understand how to create your own exceptions and
- Understand how to throw these exceptions.

This chapter consists of six sections :-

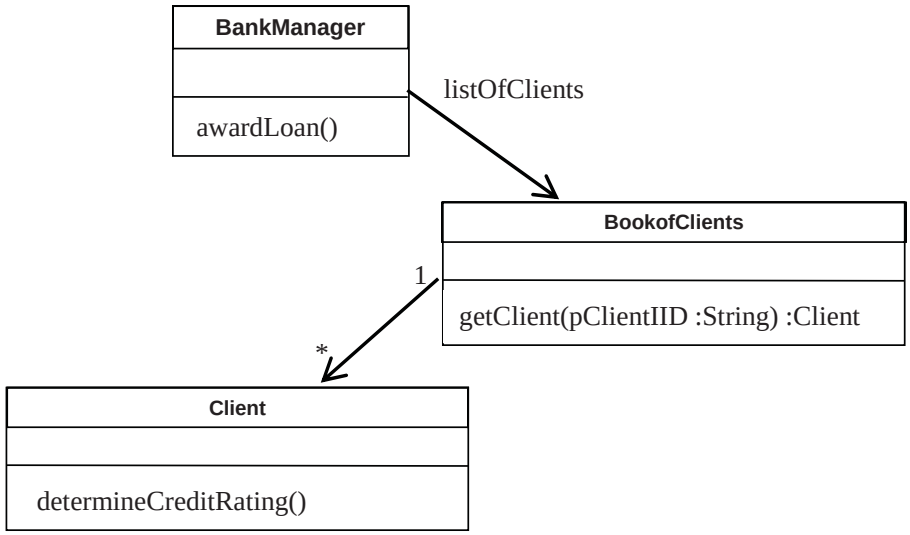
1) Understanding the Importance of Exceptions
2) Kinds of Exception
3) Extending the Exception Class
4) Throwing Exceptions
5) Catching Exceptions
6) Summary

9.1 Understanding the Importance of Exceptions

Exception handling is a critical part of writing Java programs. The authors of the file handling classes within the Java language knew this and created routines that made use of Java exception handling facilities – but are these really important? and do these facilities matter to programmers who write their own applications using Java?

Activity 1

Imagine part of a banking program made up of three classes, and three methods as shown below…



The system shown above is driven by the BankManager class. The awardLoan() method is invoked, either via the interface or from another method. This method is intended to accept or reject a loan application.

The BookofClients class maintains a set of account holders…people are added to this set if they open an account and of course they can be removed. However the only method of interest to us is the getClient() method. This method requires a string parameter (a client ID) and either returns a client object (if the client has an account at that bank) – or returns NULL (if the client does not exist).

The Client class has only one method of interest determineCreditRating(). This method is invoked to determine a clients credit rating – this is used by the BankManager class to decide if a loan should be approved or not.

Considering the scenario above look at the snippet of code below …

```
Client c = listOfClients.getClient(clientID) ;
c.determineCreditRating();
```

This fragment of code would exist in the awardLoan() method. Firstly it would invoke the getClient() method, passing a client ID as a parameter. This method would return the appropriate client object (assuming of course that a client with this ID exists) which is then stored in a local variable ‘c’. Having obtained a client the determineCreditRating() method would be invoked on this client.

Look at these two lines of code. Can you identify any potential problems with them?

Feedback 1

If a client with the specified ID exists this code above will work. However if a client does not exist with the specified ID the getClient() method will return NULL.

The second line of code would then cause a run time error (specifically a NullPointerException) as it tries to invoke the determineCreditRating() method on a non existent client and the program would crash at this point.

Activity 2

Consider the following amendment to this code and decide if this would fix the problem.

```
Client c = listOfClients.getClient(pClientID) ;
If (c !=NULL) {
      c.determineCreditRating();
}
```

Feedback 2

If the code was amended to allow for the possible NULL value returned it would work – however this protection is insecure as it relies on the programmer to spot this potential critical error.

When writing the getClient() method the author was fully aware that a client may not be found and in this case decided to return a NULL value. However this relies on every programmer who ever uses this method to recognise and protect against this eventuality.

If any programmer using this method failed to protect against a NULL return then their program could crash – potentially in this case losing the bank large sums of money. Of course in other applications, such as an aircraft control system, a program crash could have life threatening results.

A more secure programming method is required to ensure that that a potential crash situation is always dealt with!

Such a mechanism exists - it is a mechanism called ‘exceptions’.

By using this mechanism we can ensure that other programmers who use our code will be alerted to potential crash situations and the compiler will ensure that these programmers deal with the ‘issue’. Thus we can ensure that no such situation is ‘forgotten’. How they are dealt with remains a choice with a programmer who uses our methods but the compiler will ensure that they at least recognise a potential crash situation.

In the situation above rather than return a NULL value the getClient() method should generate an exception. By generating an exception the Java compiler will ensure that this situation is dealt with.

9.2 Kinds of Exception

In order to generate meaningful exceptions we need to extend the Exception classes built into the Java language – there are two of these (normal exceptions and run time exceptions).

Subclasses of java.lang.Exception are used for anticipated problems which need to be managed. They must be declared in the originating method's **throws** clause and a call to method must be placed in **try**/**catch** block.

Subclasses of java.lang.RuntimeException are used for situations which lead to runtime failure and where it may not be possible to take any sensible remedial actions. They do not need to be declared in **throws** clause and a call need not be in **try**/**catch** block (but can be).

Thus we have the choice as to whether the Java compiler should force us to explicitly deal with a particular kind of exception.

Exception subclasses are appropriate for things which we know might go wrong and where we can take sensible recovery action – e.g. IO errors.

RuntimeException subclasses are appropriate for things which should not happen at all and where there is probably nothing we can do to recover the situation, e.g. an out of memory error or discovering that the system is in an inconsistent state which should never be able to arise.

9.3 Extending the Exception Class

When writing our own methods we should look for potential failure situations (e.g. value that cannot be returned, errors that may occur in calculation etc). When a potential error occurs we should generate an 'Exception' object i.e. an object of the Exception class. However it is best to first define a subclass of the general Exception i.e. to create a specialised class and throw an object of this subtype.

A new exception is just like any new class in this case it is a subclass of **java.lang.Exception**

In the case above an error could occur if no client is found with a specified ID. Therefore we could create a new exception class called 'UnknownClientException'.

The parameter to the constructor for the Exception requires a Sting thus the constructor for UnknownClientException also requires a String. This string is used to give details of the problem that may generate an exception.

The code to create this new class is given below…..

```
import java.lang.Exception;

/*******************************************************
 * Exception thrown when attempting to get an non-existent client ID
 *
 * @author Simon Kendal
 * @version 1.0 (11th July 2009)
 *******************************************************/
class UnknownClientException extends Exception
{
    /**
    * Constructor
    *
    * @param pMessage description of exception
    */
    UnknownClientException (String pMessage)
    {
        super(pMessage);
    }
}
```

In some respects this looks rather odd. Here we are creating a subclass of Exception but our subclass does not contain any new methods – nor does it override any existing methods. Thus its functionality is identical to the superclass – however it is a subtype with a meaningful and descriptive name.

If sublasses of Exception did not exist we would only be able to catch the most general type of exception i.e an Exception object. Thus we would only be able to write a catch block that would catch every single type of exception.

Having defined a subclass we instead have a choice… a) we could define a catch block to catch objects of the general type 'Exception' i.e. it would catch ALL exceptions or b) we could define a catch block that would catch UnknownClientExceptions but would ignore other types of exception.

By looking at the online API we can see that many predefined subclass of exception already exist. There are many of these including :-

- IOException
  - CharConversionException
  - EOFException
  - FileNotFoundException
  - ObjectStreamException
- NullPointerException
- PrinterException
- SQLexception

Thus we could write a catch block that would react to any type of exception, or we could limited it to input \ output exceptions or we could be even more specific and limit it to FileNotFound exceptions.

9.4 Throwing Exceptions

Having defined our own exception we must then instruct the getClient() method to throw this exception (assuming a client has not been found with the specified ID).

To do this we must first tell the compiler that this class may generate an exception – the complier will then ensure that any future programmer who makes use of this method catches this exception.

To tell the compiler this method throws an exception we add the following statement to the methods signature 'throws UnknownClientException'**.**

```
public Client getClient(String pClientID)
                                    throws UnknownClientException
```

We must create a new instance of the UnknownClientException class and apply the throw keyword to this newly created object.

We use the keyword 'throw' to throw an exception at the appropriate point within the body of the method.

```
if (foundClient != null)
{
    return foundClient;
}
else
{
    throw new UnknownClientException("BookOfClients.getClient():
                                     unknown client ID:" + pClientID);
}
```

In the example above if a client is found the method will return the client object. However it will no longer return a NULL value. Instead if a client has not been found the constructor for UnknownClientException is invoked, using ‘new’. This constructor requires a String parameter – and the string we are passing here is an error message that is trying to be informative and helpful. The message is specifying :-

- the class which generated the exception (i.e. BookOfClients),
- the method within this class (i.e. getClient()),
- some text which explains what caused the exception and
- the value of the parameter for which a client could not be found.

By defining an UnkownClientException and using the throw clause in the header of the getClient() method we are providing a warning to all methods calling this one that an exception may be thrown. This enables the Java compiler to make sure a try/catch block is provided where required.

By doing this we are preventing potentially critical errors from going unnoticed!

9.5 Catching Exceptions

Having specified to the compiler that this method may generate an exception we are forcing other programmers to protect against critical errors by placing calls to this method within a try / catch block. The code in the try block will be terminated if an exception is generated and the code in the catch block will be initiated instead.

Thus in the example above the awardLoan() method can decide what to do if no client with the specified ID is found…..

```
try
{
     Client c = listOfClients.getClient(clientID) ;
     c.determineCreditRating();

      // add code to award or reject a loan application based on this
         credit rating

}

catch (UnknownClientException uce)
{
    System.out.println("INTERNAL ERROR IN BankManager.awardLoan()\n"
                                      + "Exception details: " + uce);

}
```

Now, instead of crashing with a NullPointerException if the client ID is not found, the UnknownClientException we have deliberately thrown will be handled by the Java Virtual Machine which will terminate the code in the **try** clause and invoke the code in the **catch** clause, which in this case will display a message warning the user about the problem.

9.6 Summary

Java exceptions provide a mechanism to deal with abnormal situations which occur during program execution.

By making use of the Java exception mechanism we are protecting against potentially life threatening program failure.

The exception mechanism will ensure other programmers who use our methods recognise and deal with error situations.

When exceptions are generated the code in a catch block will be initiated – this code could take remedial action or terminate the program generating an appropriate error message. In either case at least the program doesn’t just ‘stop’.