Session 9. Exception Handling

Developing applications that are robust is a key requirement to successful software development. Chief among the skills required to develop robust applications is the ability to detect and handle error conditions that may occur throughout the execution of an application. Historically, error detection and handling required a manual approach – the success or fail of any operation was manually checked, then error handling routines had to be explicitly invoked. This approach was often significantly complicated by the structure of the application code. Modern approaches, in particular exception handling, allow error detection and handling mechanisms to be automated somewhat, and the complexity of the task is reduced by the behaviour of exception mechanisms. In this session we examine both approaches, although our focus is on how to detect, throw, and handle exceptions.

Session Objectives

In this session, you will learn:

- Alternatives for error detection and handling;
- How exceptions interrupt the execution of code and can result in program termination;
- How to throw, catch and handle exceptions;
- What exception classes are provided by Microsoft.Net and how to define your own classes to form an exception hierarchy; and
- When and how to apply exceptions in your programs.

Unit Learning Outcomes

- *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*
  Although exceptions are not strictly an object-oriented concept, they are common to most object-oriented programming languages. The development and application of custom exception hierarchies are also closely tied to object-oriented concepts and will assist in the achievement of this ULO.
- *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*
  The application of both manual detection and handling and exception based error handling are key skills for the development of robust applications and for programming in the C# language.
- *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*
  We will not be directly addressing this ULO, however understanding of the C# programming language will contribute to the skills required for this outcome later..
- *ULO 4. Construct object-oriented designs and express them using standard UML notation.*
  We will not be directly addressing this ULO, however custom exception

hierarchies are often used in large and/or complex applications and would form part of the design and modeling.

Required Reading

Additional reading is required from the textbook Deitel, P., and Deitel, H., "Visual C# 2012: How to Program", Fifth Edition, Pearson, 2014, as follows:

Required Reading: Chapter 13

This chapter introduces exceptions including some of the exception types provided by Microsoft.Net and creating your own custom exception classes.

Required Tasks

The following tasks are mandatory for this week:

- Task 9.1. Catching Exceptions
- Task 9.2. Blackjack

9.1. Introduction

Writing the code for an application to provide required functionality is only part of the software development task. An application is not useful if the program terminates unexpectedly during normal operation. Applications terminate unexpectedly due to the occurrence of an errors. Such errors could result from a problem in the code, e.g., a logic error, a problem with the system, e.g., out of memory, or a problem resulting from processing the user's input to the program, e.g., file does not exist or inadequate permissions to access a file. The ***robustness*** of a program refers to the ability of that program to continue operation in the face of errors.

Fundamentally, there are two phases to managing errors in a program: ***error detection*** and ***error handling***. Error detection refers to the ability of a program to identify when an error has occurred, e.g., the program identifies that the user entered a textual value instead of a numeric value. Error handling refers to the ability of a program to correct for that error by performing additional functionality that returns the program to a normal state, e.g., by displaying an error message and prompt for the numeric data again.

9.2. Error Handling Without Exceptions

Historically, the occurrence of an error would usually be indicated by either the return value of the method and/or by the use of ***status codes***. A good example of this approach can be seen in Unix system calls[1], many of which return an integer value. If the value returned by a system call is non-negative, i.e., >= 0, the system call usually completed successfully. However, a return value of –1 usually indicates that an error has occurred. The type of error that occurred is then indicated using a status code,

[1] A system call is a method which is used to directly invoke the functionality of the operating system, e.g., to read or write to a file on disk.

which is a pre-defined value that represents a particular type of error. The status code is stored in a global variable by the name of `errno`[2] (short for error number). The following table lists a subset of status codes that can be set stored in the `errno` variable:

Name	Code Number	Description
ENOERROR	0	(represents no error occurred)
ENOENT	2	No such file or directory
EIO	5	I/O error
ENOMEM	12	Out of memory
EACCESS	13	Permission denied
EINVAL	22	Invalid argument
EMFILE	24	Too many open files
ENOSPC	28	No space left on device
EROFS	30	Read-only file system
ERANGE	34	Math result not representable

The use of these status codes results the following style of code:

```
if (open("someFile.txt", O_RDONLY) == -1)
{
    perror("someFile.txt");
    return -1;
}
```

The above code is been written in the C/C++ programming language (note how similar it is to C#). In this code, the `open` system call has been invoked to open a file named `"someFile.txt"` for reading (`O_RDONLY`). The `open` system call returns a non-negative number upon success and a value of `-1` when an error occurred. Thus, the call is placed in an `if` statement to check for an error result. If an error does occur, the library call `perror` is invoked, which checks the `errno` variable and displays an appropriate message using the `"someFile.txt"` parameter as a prefix for the error message. For example, if the file does not exist, the following error message is produced when the above code is run:

```
someFile.txt: No such file or directory
```

The final statement of the true part of the `if` statement is to return the same error code to the calling method, to “forward” the error message other code in the program which may be better placed to handle the error. Herein lies the disadvantage of this

[2] Actually, the errno variable only gives the appearance of a global variable but it in fact not global. This is due to the need to support multi-threaded applications, however multi-threading is beyond the scope of this unit, hence we do not explore this concept here.

approach to managing errors. Consider a scenario where the error occurs several method calls deep and must be forwarded back out by each of those methods to reach the error handling routine. If one of these methods fails to forward the error correctly, the application is now in an invalid state and will likely fail, i.e., crash.

9.3. Error Handling With Exceptions

Exceptions are quite different to the historic approach to managing errors. When an error is detected, an exception is "thrown", meaning that an object is created containing all of the information that is relevant to the error, which is forwarded directly to the error handling routine. This highlights one of the features of exceptions that is critical to understand – there is no need to forward the exception object, the error handling routine is invoked directly by the action of throwing an exception, passing the object directly to that routine. This also implies that the currently executing code is aborted, however we will examine this further in the coming sections.

At this time, it may appear that handling errors should always be accomplished using exceptions, however this is not the case – there are disadvantages to using exceptions. Firstly, and most importantly, there is usually overhead involved in throwing exceptions, resulting in a degraded performance of your application if they are used too liberally. Secondly, it can become difficult to predict/follow the execution of a program when exceptions are used because the execution jumps from one method to the error handling routine which can be several method calls back (remember this was also the primary advantage!). Thus exceptions must be used judiciously.

9.3.1. Catching Exceptions

As identified above, an exception is an object that is thrown when an error is detected. If a thrown object is not caught, the program is terminated (unhandled exception). To understand this, Figure 9.1 shows how an exception is caught. In the figure, the `Main` method invokes another method (`MethodOne()`), which in turn invokes another method (`MethodTwo()`). During the execution of `MethodTwo()`, an exception is thrown. Any exception that is thrown can only be handled by "catching" the exception object, achieved using a `try`/`catch` block. There is no `try`/`catch` block in `MethodTwo()`, so execution of that method is terminated. There is also no `try`/`catch` block in `MethodOne()`, so it is similarly terminated. Only in the `Main()` method do we see the appearance of a `try`/`catch` block, which catches and handles the exception.

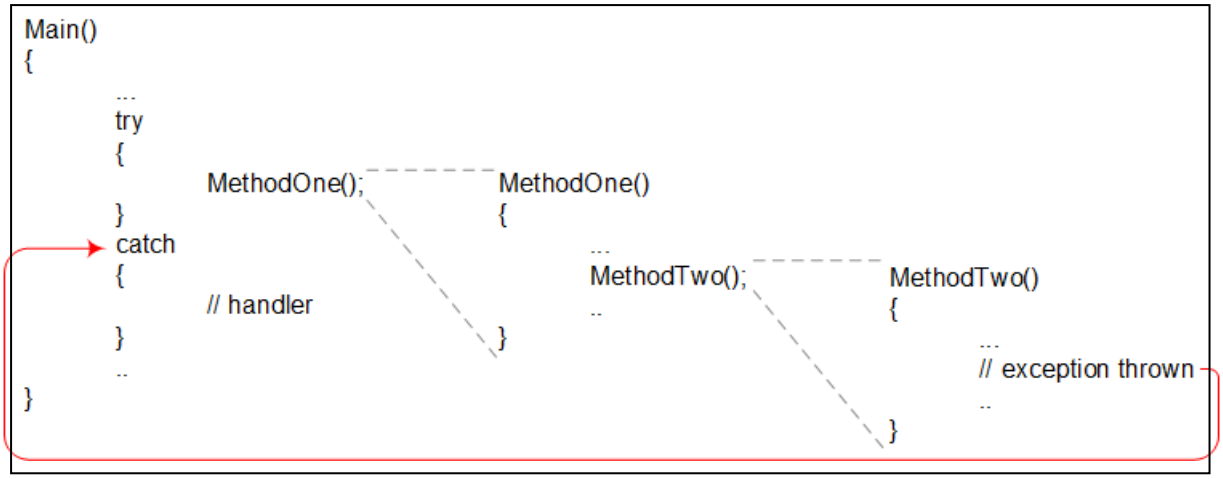

Figure 9.1: How Inheritance Works

From this example, we can see that when an exception is thrown, the code progressively steps out of each method call until a matching `try`/`catch` block is found. If a `try`/`catch` block were present in `MethodOne()`, then only `MethodTwo()` would have been terminated. Similarly, if a `try`/`catch` block were provided in `MethodTwo()`, then that method would never have been terminated either. Alternatively, if there was no `try`/`catch` appearing in the `Main()` method in Figure 9.1, there is one more `try`/`catch` that is generated by the compiler that sits outside of the main `Main()` method – this `try`/`catch` is responsible for terminating the application, and will terminate the application and display basic diagnostic information, including a stack trace[3].

The syntax for a try/catch block is as follows:

```
try
{
        code that may generate an exception
}
[catch (type[  variable_name])
{
        error handling code
}]
[...]
[catch
{
        error handling code
}]
[finally
{
        code always executed after the try/catch
}]
```

Although this appears quite busy, there is in fact three sections: the `try` block, one or more `catch` blocks, and optionally a `finally` block. The `try` block is used to identify the code that may possibly result in an exception being thrown (the call to `MethodOne()` in

[3] Recall that the call stack records the order in which methods were invoked (Section 2.10) – a stack trace just lists those methods from the call stack, allowing you to determine where the exception was thrown.

Figure 9.1 above). The `catch` blocks are then used to provide error handling routines for one or more exception object types (see Section 9.3.4 and Section 9.3.5 below), or to `catch` any type of exception (where catch is specified without ay data type). Where an exception type is specified, a variable name can optionally be specified to allow the object that was thrown to be accessed. Any number of `catch` blocks can be provided (zero or more), and their ordering is important (discussed in Section 9.3.5).

The last element of the syntax is the `finally` block, representing code that is always executed upon the completion of the `try`/`catch` block, regardless of whether an exception occurred or not. The following are possible outcomes of a `try`/`catch` block: normal execution of the `try` block completes without an exception occurring, an exception occurs and is handled completely, or an exception occurs but is not handled/not handled completely. The `finally` block is very useful for making sure that system resources are freed when needed, e.g., files, database, network connections, etc. are closed.

Now that we have seen how to catch an exception, we can show an example that directly compares error handling without exceptions (Section 9.2) and with exceptions. Each of the simple data types of the C# programming language (A) provide two methods for data conversion: `Parse()` and `TryParse()`. These methods can be used as an alternative to the `Convert` object when converting from a `string` data type. The syntax is as follows:

result = *type*.TryParse(*string_input*, out *output_variable*);

and

output_variable = *type*.Parse(*string_input*);

The difference between the two is that the `TryParse()` method returns a boolean result indicating that the value was successfully converted (`true`) or not (`false`), whereas the `Parse()` method returns the converted value when successful and throws an exception when it fails (the exception object type is `FormatException`).

The following are two equivalent examples of how to convert input to an integer, first using the `TryParse()` method:

```
Console.Write("Enter a number: ");
int value = 0;
if (int.TryParse(Console.ReadLine(), out value) == true)
    Console.WriteLine("Thank you.");
else
    Console.WriteLine("That wasn't a number!");
```

and then using the `Parse()` method and exception handling:

```
Console.Write("Enter a number: ");
int value = 0;
try
{
    value = int.Parse(Console.ReadLine());
```

```
        Console.WriteLine("Thank you.");
    }
    catch(FormatException)
    {
        Console.WriteLine("That wasn't a number!");
    }
```

Note that it is also possible to use the system message provided in the exception object as follows:

```
Console.Write("Enter a number: ");
int value = 0;
try
{
    value = int.Parse(Console.ReadLine());
    Console.WriteLine("Thank you.");
}
catch (FormatException fe)
{
    Console.WriteLine(fe.Message);
}
```

9.3.2. Throwing Exceptions

As discussed above, an exception should be thrown whenever an error condition has been detected. The various classes of the Microsoft.Net hierarchy will throw exceptions wherever relevant – these are documented in the ".Net Framework Class Library" documentation available in MSDN online at:

http://msdn.microsoft.com/en-au/library/ms229335.aspx

However the task of throwing exceptions is not restricted only to the provided class libraries. In building classes that can be reused by many applications you will often come across cases where there is a need for an exception to be thrown – an error state is clearly detectable that cannot be corrected / handled locally and must be referred to the calling code, i.e., the application, to be handled. The syntax for throwing an exception is relatively straightforward, as follows:

```
throw new type([parameter[, ...]]);
```

The type of the object to be thrown must either be of type `Exception` or of a type that is derived from the class `Exception`. These types can either be an exception type provided by the Microsoft.Net framework (Section 9.3.4) or a custom type that you can create yourself (Section 9.3.5). Regardless of the type of exception being thrown, you will usually find at least three overloads for the constructor:

```
throw new type();
throw new type(message);
throw new type(message, inner_exception);
```

In the first example, containing no parameters, an empty object is thrown. It is generally not advisable to use this overload because the object provides no information

to the error handling routine on the state of the program that led to the exception occurring. The second example includes a single parameter (*message*) which is a simple textual string describing the error that occurred. The third example adds another parameter (*inner_exception*), which is when one exception object is contained within another (see Section 9.3.3). Other exception types may provide additional parameters more specific to the type of error they represent. In this unit, for simplicity, we only consider the second of these examples.

9.3.3. Rethrowing Exceptions

Often when an exception has been caught, there is a need to rethrow the exception. This is usually for one of three reasons:

1. After being caught, the exception has been partially handled, changing the nature/type of exception that should be thrown.
2. Before the execution of the method is terminated, it is necessary to tidy up one or more resources which cannot be done using a `finally` code block, i.e., if the code was to complete successfully the resources would not be closed/freed, only if an exception occurs.
3. It is necessary to encapsulate/hide any exception objects thrown by class developed by a third-party which has been delegated functionality, i.e., where delegation is used (see Section 5.4) it is desirable to hide the entire interface of the target class, including any exceptions that are thrown.

Rethrowing an exception occurs in a catch block, either by creating a new object to be thrown (as shown in Section 9.3.2), or by rethrowing the caught object by using the statement:

```
throw;
```

Importantly the rethrowing of the same object in this manner can be optimised by the compiler, and should be favoured whenever it makes sense to do so.

9.3.4. Exceptions Defined by the Microsoft.Net Framework

The Microsoft.Net Framework defines many different types of exceptions that should be used in your program if a programmer using your class would generally expect to see those exception types and there is no additional information that should be provided to simplify the error handling process. The following provides a list of some of those classes which you may encounter or may wish to use in your own programs:

- `AccessViolationException` – attempt to read or write protected memory;
- `ArgumentException` – one or more arguments were invalid;
- `DivideByZeroException` – attempt made to divide by zero[4];
- `FileNotFoundException` – specified file does not exist;
- `IndexOutOfRangeException` – array index is out of range;

[4] Note that a divide by zero exception can only occur when dividing integer (`int`) or decimal (`decimal`) values. Dividing by zero when using floating point data (`float` or `double`) is considered a valid calculation (resulting in positive or negative infinity).

- `InvalidCastException` – a data type casting is not valid (usually because the types are unrelated);
- `InvalidOperationException` – a method call is (currently) invalid;
- `NotImplementedException` – method is not yet implemented;
- `NotSupportedException` – the functionality defined by a method is not supported for that particular object, e.g., invalid reading/writing to a file;
- `NullReferenceException` – when an attempt is made to access an attribute/operation of an object when the reference is set to null;
- `OutOfMemoryException` – the system has run out of memory;
- `OverflowException` – converting a value, such as with the Convert object, results in a loss of data, e.g., attempting to convert the value 123456 to a byte (which has range 0-255);
- `RankException` – attempt to access a dimension of an array that does not exist;
- `StackOverflowException` – the call stack cannot grow any larger;
- `UnauthorizedAccessException` – permission denied;

9.3.5. Creating Custom Exception Hierarchies

The exception types presented in the previous section are often inadequate for an application. This is because the exceptions above represent conditions that relative to the system, and not to the application. For example, which of the Microsoft.Net exception types would you apply to the following error conditions?

- Attempting to store a birth-date of February 29th in a non-leap year;
- Attempting to exceed the maximum weight of items per order for a web store;
- Attempting to approve a loan for a customer with an inadequate credit rating;

Although the `ArgumentException` or `InvalidOperationException` types may suggest possibilities, they don't provide adequate information regarding the data that causes the error. This is where custom exception classes are useful.

Strictly speaking, we could just use the general `Exception` class for any error condition that occurs in our application. Given this possibility, why bother creating custom exception classes? The key is found in how exceptions are caught. Recall from Section 9.3.1 that we can have several `catch` blocks in a `try`/`catch` block. If we were to use only a single type of object (`Exception`), it would not be easy to differentiate between the different types of exceptions that could be thrown, needing a series of `if` statements to check the contents of the messages contained in the exception objects (very inefficient). By using different types, we can place error handling routines for the different errors into separate `catch` blocks, which are then invoked automatically.

To create a new exception type, the following rules must be followed:

- Create a new class with the suffix `Exception` in the name, i.e., _____`Exception`;

- The class must be derived from the `Exception` class[5];
- A minimum of three constructors must be provided:
  - Parameter-less constructor;
  - Constructor with a single parameter: `string message`;
  - Constructor with two parameters: `string message, Exception inner`;

An example exception class created according to these requirements is as follows:

```
class InvalidBirthdateException : Exception
{
    public InvalidBirthdateException()
    {
    }

    public InvalidBirthdateException(string message)
        : base(message)
    {
    }

    public InvalidBirthdateException(string message, Exception inner)
        : base(message, inner)
    {
    }
}
```

Note that this represents only the minimum required to define a custom exception. Additional attributes, properties, constructors, and so on, can be added to provide additional information regarding the exception that occurred. It is also critical to note that the principles of inheritance must also be considered when using exception classes. In particular, the group of all classes that inherit from the `Exception` class form a hierarchy of classes (referred to as an exception hierarchy), and catching an exception for a particular type will also catch all exception types that derive from that class. Thus, the ordering of `catch` blocks appearing in the code is critical, and should list the most specialised types first and the most generalised types last. This also implies that if the first `catch` block catches the type `Exception`, the remaining `catch` blocks will not be executed – the first `catch` block will catch everything.

9.3.6. Guidelines

The following guidelines are provided by McConnell, S., “Code Complete”, Second Edition, Microsoft Press, 2004:

- Use exceptions to notify other parts of the program about errors that should not be ignored – the nature of how exceptions are thrown and caught means that they cannot be ignored (unlike error handling without exceptions, Section 9.2);
- Throw an exception only for conditions that are truly exceptional – the use of exceptions increases the complexity of the application and also weakens encap-

[5] Originally, the idea was for all exception types defined in Microsoft.Net to be defined from `SystemException`, and for all exception types defined for applications to be derived from `ApplicationException`. However this was found to have negligible value, and was abandoned. Although the `ApplicationException` class remains in the Microsoft.Net framework, the official documentation for the class clearly states that it should no longer be used.

sulation (to use a class you must be aware of the exceptions it throws, when, and why);
- Don't use an exception to pass the buck – don't throw an exception if the error condition can be handled at the point of detection;
- Avoid throwing exceptions in constructors and destructors unless you catch them in the same place – the way exceptions behave and their impact on memory management varies significantly when thrown in constructors and destructors, problems which are better avoided;
- Throw exceptions at the right level of abstraction – exception objects that are thrown from your class should be defined in the same library[6];
- Include in the exception message all the information that led to the exception – as discussed above, it is important to provide all relevant information in an exception object (exception message is referring to the object in this guideline);
- Avoid empty catch blocks – an empty catch block suggests either an exception that has been incorrectly thrown or an exception being incorrectly caught/handled;
- Know the exceptions your library code throws – some languages (such as C#) don't require the exception types they can throw to be documented, be careful to make sure you catch all possible exceptions from the library code otherwise your program can crash unexpectedly;
- Consider building a centralized exception reporter – the functionality for catching and handling exceptions will always be distributed throughout an application, which can lead to errors being reported to the user inconsistently;
- Standardize your project's use of exceptions – create a set of coding rules that govern how exceptions are thrown and handled within your code to keep them consistent and thus easier to understand;
- Consider alternatives to exceptions – always keep in mind that there is no rule that requires exceptions to be used, so consider carefully whether they need to be used at all in a particular program.

Task 9.1. Catching Exceptions

Objective: The objective of this task is to familiarise you with how to catch and handle exceptions in client code. It is important to master this concept if you are to use exceptions competently in your own programs.

Modify your solution (or the provided solution) to Task 3.3 to store the following information for a student: student ID, name, age (as date of birth), telephone number, and results for four units. Your solution must satisfy the following requirements:

- Must allow the user to enter all relevant information before displaying it on the screen;
- Must follow appropriate object-oriented design principles;

[6] Note that this guideline may appear to conflict with the general practice of throwing the exception types defined by Microsoft.Net when they are relevant. However the context of the operation throwing an exception can clarify what type of exception should be thrown, e.g., an operation such as `LoadFromFile()` may sensibly throw exception objects of type `FileNotFoundException` or `UnauthorizedAccessException`, however if the use of files is hidden from client objects these exception types would be unexpected.

- Only the `Main()` method may use the Console object, other classes may not; and
- Validate numeric data by using the relevant `Parse()` method (not `TryParse()`) and handling any exceptions appropriately.

Task 9.2. Blackjack

Objective: Understanding how when exceptions should be thrown and caught is critical to successfully developing software in modern object-oriented frameworks which often apply exceptions. This task, although not a realistic/appropriate use of exceptions, does require that you master exceptions in order to solve the problem correctly.

In Blackjack, two playing cards are initially dealt to the player and two cards to the dealer. The player can then request the dealer to "hit" them with additional cards (deal additional cards to the player's hand), or can elect to "stay" with the cards they have. Once the player stays, the dealer similarly deals additional cards to the dealer's hand in an attempt to beat the player. The winner of a round is whomever has the highest score without exceeding a score of 21, where the player/dealer is deemed to have "busted". Code is provided in CloudDeakin that provides a very simple working version of the game for a single player's hand (no dealers hand), and only dealing extra cards (no "stays"). Your task is to identify where modifications will be needed to support two changes:

a. The provided code cannot handle the same card being selected to be dealt from the deck. Identify the changes required to extend the game to handle this. Ignore the possibility that there may be no cards left in the deck for the moment.
b. The provided code cannot handle more than one hand being dealt (and busted). Identify the changes required to extend the game to handle this. Again, ignore the possibility of the deck running out of cards.

To become a successful programmer/software developer, you need to establish an attitude where you choose to explore programming wherever you see the opportunity to expand or improve your knowledge. For example, in this task, you could also complete the following points (not required):

c. Extend the game to use a dealer shoe of eight decks of cards, rather than a single deck. Note that the only modification that should be needed to existing code is to change the reference to the Deck in the Dealer class to a reference to a Shoe.
d. Extend the game to give the player the option to "stay".
e. Extend the game to handle the dealer's hand as well. The initial deal is done in the order {player, dealer, player, dealer}, and the dealer's first card remains faced down until the player elects to stay. The dealer then deals cards to their hand until they either (a) beat the player's score, (b) are busted, or (c) reach a specified minimum score e.g., a score of 15.
f. Extend the game to handle multiple players (you will need player names, etc.). The initial deal is done in the order {player 1, player 2, ..., player n, dealer, p.1,

p.2, ..., p.n, d.}. Additional cards are then dealt to those players who wish to "hit" in turn {p.1, p.2., ..., p.n, p.1, p.2, etc.}.

Note: No solution will be provided for these additional tasks (tasks c-f), use your own knowledge to test your answers. Similarly, tasks c-f do not need to be submitted.