Exception Handling

Objectives

- To understand exceptions and error handling.
- To use `try` blocks to delimit code in which exceptions may occur.
- To throw exceptions.
- To use `catch` blocks to specify exception handlers.
- To use the `finally` block to release resources.
- To understand the .NET exception-class hierarchy.
- To create programmer-defined exceptions.

"It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something."
Franklin Delano Roosevelt

"O! throw away the worser part of it, And live the purer with the other half."
William Shakespeare

"If they're running and they don't look where they're going I have to come out from somewhere and catch them."
Jerome David Salinger

"And oftentimes excusing of a fault Doth make the fault the worse by the excuse."
William Shakespeare

"I never forget a face, but in your case I'll make an exception."
Groucho (Julius Henry) Marx
11.1 Introduction

In this chapter, we introduce exception handling. An exception is an indication of a problem that occurs during a program’s execution. The name “exception” comes from the fact that although a problem can occur, the problem occurs infrequently—if the “rule” is that a statement normally executes correctly, then the “exception to the rule” is that a problem occurs. Exception handling enables programmers to create applications that can resolve (or handle) exceptions. In many cases, handling an exception allows a program to continue executing as if no problem was encountered. A more severe problem may prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem, then terminate in a controlled manner. The features presented in this chapter enable programmers to write clear, robust and more fault-tolerant programs.

The style and details of exception handling in MC++ are based in part on the work of Andrew Koenig and Bjarne Stroustrup, as presented in their paper, “Exception Handling for C++ (revised).” MC++ designers extended the exception-handling mechanism used in standard C++.

This chapter begins with an overview of exception-handling concepts, then demonstrates basic exception-handling techniques. The chapter continues with an overview of the .NET exception class hierarchy. Programs typically request and release resources (such as files on disk) during program execution. Often, these resources are in limited supply or can be used by only one program at a time. We demonstrate a part of the exception-handling mechanism that enables a program to use a resource, then guarantees that the program releases the resource for use by other programs. The chapter continues with an example that demonstrates several properties of class System::Exception (the base class of all exception classes), followed by an example that shows programmers how to create and use their own exception classes.

11.2 Exception Handling Overview

The logic of the program frequently tests conditions that determine how program execution proceeds. We begin by performing a task. We then test whether that task executed correctly.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
If not, we perform error processing. Otherwise we continue with the next task. Although this form of error-handling logic works, intermixing the logic of the program with the error-handling logic can make the program difficult to read, modify, maintain and debug—especially in large applications. In fact, if many of the potential problems occur infrequently, intermixing program logic and error handling can degrade the performance of the program, because the program must test extra conditions to determine whether the next task can be performed.

Exception handling enables the programmer to remove error-handling code (i.e., the code that resolves the error) from the “main line” of the program’s execution. This improves program clarity and enhances modifiability. Programmers can decide to handle whatever exceptions they choose—all types of exceptions, all exceptions of a certain type or all exceptions of a group of related types. Such flexibility reduces the likelihood that errors will be overlooked and thereby increases a program’s robustness.

**Error-Prevention Tip 11.1**

Exception handling helps improve a program’s fault tolerance. When it is easy to write error-processing code, programmers are more likely to use it.

**Software Engineering Observation 11.1**

Although it is possible to do so, do not use exception handling for conventional flow of control. It is difficult to keep track of a larger number of exception cases, and programs with a large number of exception cases are hard to read and maintain.

Exception handling is designed to process *synchronous errors*—errors that occur during the normal program flow of control. Common examples of these errors are out-of-range array subscripts, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid method parameters and running out of available memory. Exception handling also can process certain *asynchronous* events, such as disk I/O completions.

**Good Programming Practice 11.1**

Avoid using exception handling for purposes other than error handling, because this can reduce program clarity.

With programming languages that do not support exception handling, programmers often delay the writing of error-processing code and sometimes simply forget to include it. This results in less robust software products. MC++ enables the programmer to deal with exception handling easily from the inception of a project. Still, the programmer must put considerable effort into incorporating an exception-handling strategy into software projects.

**Software Engineering Observation 11.2**

Try to incorporate the exception-handling strategy into a system from the inception of the design process. Adding effective exception handling after a system has been implemented can be difficult.

**Software Engineering Observation 11.3**

In the past, programmers used many techniques to implement error-processing code. Exception handling provides a single, uniform technique for processing errors. This helps programmers working on large projects to understand each other’s error-processing code.

The exception-handling mechanism also is useful for processing problems that occur when a program interacts with software elements such as methods, constructors, assemblies and classes. Rather than internally handling problems that occur, such software elements
Chapter 11 Exception Handling

often use exceptions to notify programs when problems occur. This enables programmers to implement customized error handling for each application.

**Common Programming Error 11.1**

Aborting a program component could leave a resource—such as file stream or I/O device—in a state in which other programs are unable to acquire the resource. This is known as a “resource leak.”

**Performance Tip 11.1**

When no exceptions occur, exception-handling code incurs little performance penalties. Thus, programs that implement exception handling operate more efficiently than programs that perform error handling throughout the program logic.

**Performance Tip 11.2**

Exception handling should be used only for problems that occur infrequently. As a “rule of thumb,” if a problem occurs at least 30% of the time when a particular statement executes, the program should test for the error inline; otherwise, the overhead of exception handling will cause the program to execute more slowly.2

**Software Engineering Observation 11.4**

Methods with common error conditions should return a null pointer (or another appropriate value) rather than throwing exceptions. A program calling such a method simply can check the return value to determine success or failure of the method call.3

Complex applications normally consist of predefined software components (such as those defined in the .NET Framework) and components specific to the application that use the predefined components. When a predefined component encounters a problem, that component needs a mechanism to communicate the problem to the application-specific component—the predefined component cannot know in advance how each application will process a problem that occurs. Exception handling simplifies combining software components and having them work together effectively by enabling predefined components to communicate problems that occur to application-specific components, which can then process the problems in an application-specific manner.

Exception handling is geared to situations in which the method or function that detects an error is unable to handle it. Such a method or function throws an exception. There is no guarantee that there will be an exception handler—code that executes when the program detects an exception—to process that kind of exception. If there is, the exception will be caught and handled. The result of an uncaught exception depends on whether the program executes in debug mode or standard execution mode. In debug mode, when the program detects an uncaught exception, a dialog box appears that enables the programmer to view the problem in the debugger or continue program execution by ignoring the problem that occurred. In standard execution mode, a Windows application presents a dialog that enables the user to open the program in the debugger or terminate program execution, and a console application presents a dialog that enables the user to open the program in the debugger or terminate program execution.

---

MC++ uses **try blocks** to enable exception handling. A **try** block consists of keyword **try** followed by braces ({}) that define a block of code in which exceptions may occur. The **try** block encloses statements that might cause exceptions and any statements that should not execute if an exception occurs. Immediately following the **try** block are zero or more **catch blocks** (also called **catch handlers**). Each catch handler specifies in parentheses an exception parameter that represents the type of exception the catch handler can handle. If an exception parameter includes an optional parameter name, the catch handler can use that parameter name to interact with a caught exception object. Optionally, programmers can include a **parameterless catch handler** (also called a “catch-all” handler) that catches all exception types. After the last catch handler, an optional **finally block** contains code that always executes (provided that program execution entered the **try** block), regardless of whether an exception occurs. A **try** block must be followed by at least one **catch** block or a **finally** block.

### Common Programming Error 11.2

The parameterless catch handler must be the last catch handler following a particular try block; otherwise, a syntax error occurs.

When a method or function called in a program detects an exception or when the CLR (Common Language Runtime) detects a problem, the method/function or CLR throws an exception. The point in the program at which an exception occurs is called the **throw point**—an important location for debugging purposes (as we demonstrate in Section 11.6). Exceptions are objects of classes that extend class **Exception** of namespace **System**. If an exception occurs in a **try** block, the **try** block expires (i.e., terminates immediately) and program control transfers to the first catch handler (if there is one) following the **try** block. MC++ is said to use the **termination model of exception handling**, because the **try** block enclosing a thrown exception expires immediately when that exception occurs.\(^4\) As with any other block of code, when a **try** block terminates, local variables defined in the block go out of scope. Next, the CLR searches for the first catch handler that can process the type of exception that occurred. The CLR locates the matching catch by comparing the thrown exception’s type to each catch’s exception-parameter type until the CLR finds a match. A match occurs if the types are identical or if the thrown exception’s type is a derived class of the exception-parameter type. When a catch handler finishes processing, local variables defined within the catch handler (including the catch parameter) go out of scope. If a match occurs, code contained within the matching catch handler executes. All remaining catch handlers that correspond to the **try** block are ignored and execution resumes at the first line of code after the **try...catch** sequence.

If no exceptions occur in a **try** block, the CLR ignores the exception handlers for that block. Program execution resumes with the next statement after the **try...catch** sequence. If an exception that occurs in a **try** block has no matching catch handler, or if an exception occurs in a statement that is not in a **try** block, the method or function containing that statement terminates immediately and the CLR attempts to locate an enclosing try block in a calling method or function. This process is called **stack unwinding** (discussed in Section 11.6).

---

4. Some languages use the *resumption model of exception handling*, in which, after the handling of the exception, control returns to the point at which the exception was thrown and execution resumes from that point.
11.3 Example: DivideByZeroException

Let us consider a simple example of exception handling. The application in Fig. 11.1 uses `try` and `catch` to specify a block of code that might throw exceptions and to handle those exceptions if they occur. The application prompts the user to enter two integers. The program converts the input values to type `int` and divides the first number (`numerator`) by the second number (`denominator`). Assuming that the user provides integers as input and does not specify 0 as the denominator for the division, line 24 displays the division result. However, if the user inputs a non-integer value or supplies 0 as the denominator, an exception occurs. This program demonstrates how to catch these exceptions.

```cpp
// Fig. 11.1: DivideByZero.cpp
// Divide-by-zero exception handling.

#include "stdafx.h"

#include <mscorlib.dll>
using namespace System;

int _tmain()
{
    try {
        Console::Write( S"Enter an integral numerator: " );
        int numerator = Convert::ToInt32( Console::ReadLine() );
        Console::Write( S"Enter an integral denominator: " );
        int denominator = Convert::ToInt32( Console::ReadLine() );
        // division generates DivideByZeroException if
        // denominator is 0
        int result = numerator / denominator;
        Console::WriteLine( result );
    } // end try

    // process invalid number format
    catch (FormatException *) {
        Console::WriteLine( S"You must enter two integers." );
    } // end catch

    // user attempted to divide by zero
    catch (DivideByZeroException *divideByZeroException ) {
        Console::WriteLine( divideByZeroException->Message );
    } // end catch

    return 0;
} // end _tmain
```

Fig. 11.1 Divide-by-zero exception handling example. (Part 1 of 2.)
Before we discuss the program details, consider the sample outputs in Fig. 11.1. The first output shows a successful calculation in which the user inputs the numerator 100 and the denominator 7. Note that the result (14) is an integer, because integer division always yields integer results. The second output shows the result of inputting a non-integer value—in this case, the user input "hello" at the second prompt. The program attempts to convert the string the user input into an int value with method Convert::ToInt32. If the argument to Convert::ToInt32 is not a valid representation of an integer (in this case a valid string representation of an integer, such as "14"), a FormatException (namespace System) is generated. The program detects the exception and displays an error message, indicating that the user must enter two integers. The last output demonstrates the result after an attempt to divide by zero. In integer arithmetic, the CLR tests for division by zero and generates a DivideByZeroException (namespace System) if the denominator is zero. The program detects the exception and displays an error-message, indicating an attempt to divide by zero.5

Let us consider the user interactions and flow of control that yield the results shown in the sample outputs. Lines 12–25 define a try block that encloses the code that can throw exceptions, as well as the code that should not execute if an exception occurs. For example, the program should not display a new result (line 24) unless the calculation (line 22) completes successfully. Remember that the try block terminates immediately if an exception occurs, so the remaining code in the try block will not execute.

The two statements that read the integers (lines 15 and 18) each call method Convert::ToInt32 to convert strings to int values. This method throws a FormatException if it cannot convert its String * argument to an integer. If lines 15 and 18 properly convert the values (i.e., no exceptions occur), then line 22 divides the numerator by the denominator.

---

5. The CLR allows floating-point division by zero, which produces a positive or negative infinity result, depending on whether the numerator is positive or negative. Dividing zero by zero is a special case that results in a value called “not a number.” Programs can test for these results using constants for positive infinity (PositiveInfinity), negative infinity (NegativeInfinity) and not a number (NaN) that are defined in structures Double (for double calculations) and Single (for float calculations).

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
denominator and assigns the result to variable result. If the denominator is zero, line 22 causes the CLR to throw a DivideByZeroException. If line 22 does not cause an exception, then line 24 displays the result of the division. If no exceptions occur in the try block, the program successfully completes the try block and ignores the catch handlers in lines 28–30 and 33–35—the program execution continues with the first statement following the try...catch sequence.

Immediately following the try block are two catch handlers (also called catch blocks or exception handlers)—lines 28–30 define the exception handler for a FormatException, and lines 33–35 define the catch handler for the DivideByZeroException. Each catch handler begins with keyword catch followed by an exception parameter in parentheses that specifies the type of exception handled by the catch handler. The exception-handling code appears in the catch handler. In general, when an exception occurs in a try block, a catch handler catches the exception and handles it. In Fig. 11.1, the first catch handler specifies that it catches the type FormatExceptions (thrown by method Convert::ToInt32), and the second catch handler specifies that it catches type DivideByZeroExceptions (thrown by the CLR). Only the matching catch handler executes if an exception occurs. Both of the exception handlers in this example display an error-message to the user. When program control reaches the end of a catch handler, the program considers the exception as having been handled, and program control continues with the first statement after the try...catch sequence (the return statement of function _tmain in this example).

In the second sample output, the user input hello as the denominator. When line 18 executes, Convert::ToInt32 cannot convert this string to an int, so Convert::ToInt32 throws a FormatException object to indicate that the method was unable to convert the String * to an int. When an exception occurs, the try block expires (terminates). Any local variables defined in the try block go out of scope; therefore, those variables (numerator, denominator and result in this example) are not available to the exception handlers. Next, the CLR attempts to locate a matching catch handler, starting with the catch in line 28. The program compares the type of the thrown exception (FormatException) with the type in parentheses following keyword catch (also FormatException). A match occurs, so that exception handler executes and the program ignores all other exception handlers following the corresponding try block. Once the catch handler finishes processing, local variables defined within the catch handler go out of scope. If a match did not occur, the program compares the type of the thrown exception with the next catch handler in sequence and repeats the process until a match is found.

Software Engineering Observation 11.5
Enclose in a try block a significant logical section of the program in which several statements can throw exceptions, rather than using a separate try block for every statement that throws an exception. However, for proper exception-handling granularity, each try block should enclose a section of code small enough that when an exception occurs, the specific context is known and the catch handlers can process the exception properly.

Common Programming Error 11.3
Attempting to access a try block’s local variables in one of that try block’s associated catch handlers is an error. Before a corresponding catch handler can execute, the try block expires, and its local variables go out of scope.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Common Programming Error 11.4

Specifying a comma-separated list of exception parameters in a catch handler is a syntax error. Each catch can have only one exception parameter.

In the third sample output, the user input 0 as the denominator. When line 22 executes, the CLR throws a DivideByZeroException object to indicate an attempt to divide by zero. Once again, the try block terminates immediately upon encountering the exception, and the program attempts to locate a matching catch handler, starting from the catch handler in line 28. The program compares the type of the thrown exception (DivideByZeroException) with the type in parentheses following keyword catch (FormatException). In this case, there is no match, because they are not the same exception types and because FormatException is not a base class of DivideByZeroException. So the program proceeds to line 33 and compares the type of the thrown exception (DivideByZeroException) with the type in parentheses following keyword catch (also DivideByZeroException). A match occurs, so that exception handler executes. Line 34 in this handler uses property Message of class Exception to display the error message to the user. If there were additional catch handlers, the program would ignore them.

Notice that the catch handler that begins in line 33 specifies parameter name divide-ByZeroException. Line 34 uses this parameter (of type DivideByZeroException *) to interact with the caught exception and access its Message property. Meanwhile, the catch handler that begins in line 28 does not specify a parameter name. Thus, this catch handler cannot interact with the caught exception object.

11.4 .NET Exception Hierarchy

The exception-handling mechanism allows only objects of class Exception and its derived classes to be thrown and caught. This section overviews several of the .NET Framework’s exception classes. In addition, we discuss how to determine whether a particular method throws exceptions.

Class Exception of namespace System is the base class of the .NET Framework exception hierarchy. Two of the most important derived classes of Exception are ApplicationException and SystemException. ApplicationException is a base class that programmers can extend to create exception data types that are specific to their applications. We discuss creating programmer-defined exception classes in Section 11.7. Programs can recover from most ApplicationException s and continue execution.

The CLR can generate SystemExceptions at any point during the execution of the program. Many of these exceptions can be avoided by coding properly. These are called runtime exceptions and they derive from class SystemException. For example, if a program attempts to access an out-of-range array subscript, the CLR throws an exception of type IndexOutOfRangeException (a class derived from SystemException). Similarly, a runtime exception occurs when a program attempts to use a null pointer to manipulate an object. Attempting to use such a null pointer causes a NullReferenceException (another

---

6. Actually, it is possible to catch exceptions of types that are not derived from class Exception using the parameterless catch handler. This is useful for handling exceptions from code written in other languages that do not require all exception types to derive from class Exception in the .NET framework.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
type of `SystemException`). For a complete list of derived classes of `Exception`, look up “Exception class” in the Index of the Visual Studio .NET online documentation.

A benefit of using the exception-class hierarchy is that a catch handler can catch exceptions of a particular type or can use a base-class type to catch exceptions in a hierarchy of related exception types. For example, a catch handler that specifies an exception parameter of type `Exception` also can catch exceptions of all classes that extend `Exception`, because `Exception` is the base class of all exception classes. This allows for polymorphic processing of related exceptions. The benefit of the latter approach is that the exception handler can use the exception parameter to manipulate the caught exception. If the exception handler does not need access to the caught exception, the exception parameter may be omitted. If no exception type is specified, the catch handler will catch all exceptions. [Note: In such cases, the catch handler’s parameter is specified as an ellipse (...). You will learn more about the ellipse in Section 11.5.]

Using inheritance with exceptions enables an exception handler to catch related exceptions with a concise notation. An exception handler certainly could catch each derived-class exception type individually, but catching the base-class exception type is more concise. However, this makes sense only if the handling behavior is the same for a base class exception and its derived exception classes. Otherwise, catch each derived-class exception individually.

At this point, we know that there are many different exception types. We also know that methods and the CLR can both throw exceptions. But, how do we determine that an exception could occur in a program? For methods in the .NET Framework classes, we can look at the detailed description of the methods in the online documentation. If a method throws an exception, its description contains a section called “Exceptions” that specifies the types of exceptions thrown by the method and briefly describes potential causes for the exceptions. For example, look up “Convert.ToInt32 method” in the index of the Visual Studio .NET online documentation. In the document that describes the method, click the link “public: static int ToInt32(String *);” In the document that appears, the “Exceptions” section indicates that method `Convert.ToInt32` throws three exception types—`ArgumentException`, `FormatException` and `OverflowException”—and describes the conditions under which each exception type occurs.

**Good Programming Practice 11.2**

Inserting a comment header for each method that lists the explicit exceptions that the method can throw makes future maintenance of the code easier.

**Software Engineering Observation 11.6**

If a method is capable of throwing exceptions, statements that invoke that method should be placed in try blocks and those exceptions should be caught and handled.

### 11.5 __finally Block

Programs frequently request and release resources dynamically (i.e., at execution time). For example, a program that reads a file from disk first attempts to open the file. If that request succeeds, the program reads the contents of the file. Operating systems typically prevent more than one program from manipulating a file at once. Therefore, when a program finishes processing a file, the program normally closes the file (i.e., releases the resource).
This enables other programs to use the file. Closing the file helps prevent the resource leak, in which the file resource is unavailable to other programs because a program using the file never closed it. Programs that obtain certain types of resources (such as files) must return those resources explicitly to the system to avoid resource leaks.

In programming languages, like C and C++, in which the programmer is responsible for dynamic memory management, the most common type of resource leak is a memory leak. This happens when a program allocates memory (as we do with operator new), but does not deallocate the memory when the memory is no longer needed in the program. In MC++, this normally is not an issue, because the CLR performs “garbage collection” of memory no longer needed by an executing program. However, other kinds of resource leaks (such as the unclosed file mentioned previously) can occur in MC++.

**Error-Prevention Tip 11.2**

The CLR does not completely eliminate memory leaks. The CLR will not garbage-collect an object until the program has no more pointers to that object. In addition, the CLR cannot re-claim unmanaged memory. Thus, memory leaks can occur if programmers erroneously keep pointers to unwanted objects.

Most resources that require explicit release have potential exceptions associated with the processing of the resource. For example, a program that processes a file might receive IOExceptions during the processing. For this reason, file-processing code normally appears in a try block. Regardless of whether a program successfully processes a file, the program should close the file when the file is no longer needed. Suppose that a program places all resource-request and resource-release code in a try block. If no exceptions occur, the try block executes normally and releases the resources after using them. However, if an exception occurs, the try block may expire before the resource-release code can execute. We could duplicate all resource-release code in the catch handlers, but this makes the code more difficult to modify and maintain.

The exception-handling mechanism provides the __finally block, which is guaranteed to execute if program control enters the corresponding try block. The __finally block executes regardless of whether that try block executes successfully or an exception occurs. This guarantee makes the __finally block an ideal location to place resource deallocation code for resources acquired and manipulated in the corresponding try block. If the try block executes successfully, the __finally block executes immediately after the try block terminates. If an exception occurs in the try block, the __finally block executes immediately after a catch handler completes exception handling. If the exception is not caught by a catch handler associated with that try block or if a catch handler associated with that try block throws an exception, the __finally block executes, then the exception is processed by the next enclosing try block (if there is one).

**Error-Prevention Tip 11.3**

A __finally block typically contains code to release resources acquired in the corresponding try block; this makes the __finally block an effective way to eliminate resource leaks.

**Error-Prevention Tip 11.4**

The only reason a __finally block will not execute if program control entered the corresponding try block is that the application terminates before __finally can execute.
Performance Tip 11.3
As a rule, resources should be released as soon as it is apparent that they are no longer needed in a program, to make those resources immediately available for reuse, thus enhancing resource utilization in the program.

If one or more catch handlers follow a try block, the __finally block is optional. If no catch handlers follow a try block, a __finally block must appear immediately after the try block. If any catch handlers follow a try block, the __finally block appears after the last catch. Only white space and comments can separate the blocks in a try...catch...__finally sequence.

Common Programming Error 11.5
Placing the __finally block before a catch handler is a syntax error.

The MC++ application in Fig. 11.2 demonstrates that the __finally block always executes, even if no exception occurs in the corresponding try block. The program contains four functions that _tmain invokes to demonstrate __finally—DoesNotThrowException (lines 62–81), ThrowExceptionWithCatch (lines 84–107), ThrowExceptionWithoutCatch (lines 110–128) and ThrowExceptionCatchRe-throw (lines 131–160).

```c++
// Fig. 11.2: UsingExceptionTest.cpp
// Demonstrating __finally blocks.

#include "stdafx.h"

using namespace System;

void DoesNotThrowException();
void ThrowExceptionWithCatch();
void ThrowExceptionWithoutCatch();
void ThrowExceptionCatchRethrow();

// main entry point for application
int _tmain()
{
  // Case 1: no exceptions occur in called function
  Console::WriteLine( S"Calling DoesNotThrowException" );
  DoesNotThrowException();

  // Case 2: exception occurs and is caught
  Console::WriteLine( S"\nCalling ThrowExceptionWithCatch" );
  ThrowExceptionWithCatch();

  // Case 3: exception occurs, but not caught
  // in called function, because no catch handlers
```

Fig. 11.2  Demonstrating that __finally blocks always execute regardless of whether an exception occurs. (Part 1 of 4.)

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
try {
    Console::WriteLine("nCalling ThrowExceptionWithoutCatch");
    try {
        ThrowExceptionWithoutCatch();
    } // end try

    // calls ThrowExceptionWithoutCatch
    catch (...)
    {
        Console::WriteLine("Caught exception from: 
    "S"ThrowExceptionWithoutCatch in _tmain"); 
    } // end catch

    // process exception returned from ThrowExceptionWithoutCatch
    catch (...)
    {
        Console::WriteLine("Caught exception from: 
    "S"ThrowExceptionWithoutCatch in _tmain"); 
    } // end catch

    // Case 4: exception occurs and is caught
    // in called function, then rethrown to caller
    Console::WriteLine("nCalling ThrowExceptionCatchRethrow");

    try {
        ThrowExceptionCatchRethrow();
    } // end try

    // call ThrowExceptionCatchRethrow
    catch (...)
    {
        Console::WriteLine(String::Concat ( S"Caught exception from ", 
                S"ThrowExceptionCatchRethrow in _tmain");
    } // end catch

    return 0;
} // end _tmain

// no exception thrown
void DoesNotThrowException()
{
    // try block does not throw any exceptions
    try {
        Console::WriteLine("In DoesNotThrowException"); 
    } // end try

    // this catch never executes
    catch (...)
    {
        Console::WriteLine("This catch never executes"); 
    } // end catch

    // __finally executes because corresponding try executed
    __finally
    {
        Console::WriteLine("Finally executed in DoesNotThrowException"); 
    } // end finally

Fig. 11.2 Demonstrating that __finally blocks always execute regardless of whether an exception occurs. (Part 2 of 4.)
Chapter 11
Exception Handling

Fig. 11.2 Demonstrating that __finally blocks always execute regardless of whether an exception occurs. (Part 3 of 4.)
// throws exception, catches it and rethrows it
void ThrowExceptionCatchRethrow()
{
  // try block throws exception
  try {
    Console::WriteLine(S"In ThrowExceptionCatchRethrow");
    throw new Exception(S"Exception in ThrowExceptionCatchRethrow");
  } // end try

  // catch any exception, place in object error
  catch ( Exception *error ) {
    Console::WriteLine(String::Concat(S"Message: ", error->Message));

    // rethrow exception for further processing
    throw error;

    // unreachable code; would generate logic error
  } // end catch

  // __finally executes because of corresponding try executed
  finally {
    Console::WriteLine(String::Concat(S"Finally executed in ",
      S"ThrowExceptionCatchRethrow"));
  } // end finally

// unreachable code; would generate logic error
Console::WriteLine(S"This will never be printed");
} // end function ThrowExceptionCatchRethrow

Calling DoesNotThrowException
In DoesNotThrowException
End of DoesNotThrowException

Calling ThrowExceptionWithCatch
In ThrowExceptionWithCatch
Message: Exception in ThrowExceptionWithCatch
Finally executed in ThrowExceptionWithCatch
End of ThrowExceptionWithCatch

Calling ThrowExceptionWithoutCatch
In ThrowExceptionWithoutCatch
Finally executed in ThrowExceptionWithoutCatch
Caught exception from: ThrowExceptionWithoutCatch in _tmain

Calling ThrowExceptionCatchRethrow
In ThrowExceptionCatchRethrow
Message: Exception in ThrowExceptionCatchRethrow
Finally executed in ThrowExceptionCatchRethrow
Caught exception from ThrowExceptionCatchRethrow in _tmain

Fig. 11.2 Demonstrating that __finally blocks always execute regardless of whether an exception occurs. (Part 4 of 4.)
Line 20 invokes function **DoesNotThrowException** (lines 62–81). The try block (lines 65–67) begins by outputting a message (line 66). The try block does not throw any exceptions, so program control reaches the closing brace of the try block. The program skips the catch handler (lines 70–72), because no exception is thrown; instead, the program executes the __finally block (lines 75–78), which outputs a message. At this point, program control continues with the first statement after the __finally block (line 80), which outputs a message indicating that the end of the function has been reached. Then, program control returns to _tmain. Notice that line 70 specifies the parameterless catch handler by placing an ellipsis (...) within the parentheses following keyword catch. Recall that the parameterless catch handler matches any exception.

Line 24 of _tmain invokes function **ThrowExceptionWithCatch** (lines 84–107), which begins in its try block (lines 87–92) by outputting a message. Next, the try block creates a new Exception object and uses a throw statement to throw the exception object (lines 90–91). The string passed to the constructor becomes the exception object’s error message. When a throw statement executes in a try block, the try block expires immediately, and program control continues at the first catch (lines 95–98) following this try block. In this example, the type thrown (Exception) matches the type specified in the catch, so lines 96–97 output a message indicating the exception that occurred. Then, the finally block (lines 101–104) executes and outputs a message. At this point, program control returns to _tmain. Before control returns to _tmain, the __finally block (lines 121–124) executes and outputs a message. At this point, program control continues at the first statement after the __finally block (line 125), which outputs a message indicating that the end of the function has been reached, then program control returns to _tmain. Notice that, in line 97, we use the exception object’s Message property to access the error message associated with the exception—the (message passed to the Exception constructor). Section 11.6 discusses several properties of class Exception.

**Common Programming Error 11.6**

*In MC++, the expression in a throw statement—an exception object—must be of either class Exception or one of its derived classes.*

Lines 32–34 of _tmain define a try block in which _tmain invokes function **ThrowExceptionWithoutCatch** (lines 110–128). The try block enables _tmain to catch any exceptions thrown by ThrowExceptionWithoutCatch. The try block in lines 113–118 of ThrowExceptionWithoutCatch begins by outputting a message. Next, the try block throws an Exception (lines 116–117), and the try block expires immediately. Normally, program control would continue at the first catch following the try block. However, this try block does not have any corresponding catch handlers. Therefore, the exception is not caught in function ThrowExceptionWithoutCatch. Normal program control cannot continue until that exception is caught and processed. Thus, the CLR will terminate ThrowExceptionWithoutCatch and program control will return to _tmain. Before control returns to _tmain, the __finally block (lines 121–124) executes and outputs a message. At this point, program control returns to _tmain—any statements appearing after the __finally block would not execute. In this example, because the exception thrown in lines 116–117 is not caught, function ThrowExceptionWithoutCatch always terminates after the __finally block executes. In _tmain, the catch handler in lines 37–40 catches the exception and displays a message indicating that the exception was caught in _tmain.

Lines 48–50 of _tmain define a try block in which _tmain invokes function **ThrowExceptionCatchRethrow** (lines 131–160). The try block enables _tmain to catch any
exceptions thrown by ThrowExceptionCatchRethrow. The try block in lines 134–139 of ThrowExceptionCatchRethrow begins by outputting a message (line 135), then throwing an Exception (lines 137–138). The try block expires immediately, and program control continues at the first catch (lines 142–150) following the try block. In this example, the type thrown (Exception) matches the type specified in the catch, so lines 143–144 output a message indicating the exception that occurred. Line 147 uses the throw statement to rethrow the exception. This indicates that the catch handler performed partial processing (or no processing) of the exception and is now passing the exception back to the calling function (in this case _tmain) for further processing. Note that the expression in the throw statement is the pointer to the exception that was caught. When rethrowing the original exception, you can also use the statement

throw;

with no expression. Section 11.6 discusses the throw statement with an expression. Such a throw statement enables programmers to catch an exception, create an exception object, then throw a different type of exception from the catch handler. Class library designers often do this to customize the exception types thrown from methods in their class libraries or to provide additional debugging information.

Software Engineering Observation 11.7
Whenever possible, a method should handle exceptions that are thrown in that method, rather than passing the exceptions to another region of the program.

Software Engineering Observation 11.8
Before throwing an exception to a calling method, the method that throws the exception should release any resources acquired within the method before the exception occurred.

The exception handling in function ThrowExceptionCatchRethrow did not complete, because the program cannot run code in the catch handler placed after the invocation of the throw statement (line 147). Therefore, function ThrowExceptionCatchRethrow will terminate and return control to _tmain. Once again, the __finally block (lines 153–156) will execute and output a message before control returns to __tmain. When control returns to __tmain, the catch handler in lines 53–56 catches the exception and displays a message indicating that the exception was caught.

Note that the point at which program control continues after the __finally block executes depends on the exception-handling state. If the try block successfully completes or if a catch handler catches and handles an exception, control continues with the next statement after the __finally block. If an exception is not caught or if a catch handler rethrows an exception, program control continues in the next enclosing try block. The enclosing try may be in the calling function or one of its callers. Nesting a try...catch sequence in a try block is also possible, in which case the outer try block’s catch handlers would process any exceptions that were not caught in the inner try...catch sequence. If a try block has a corresponding __finally block, the __finally block executes even if the try block terminates due to a return statement; then the return occurs.

---


©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
Common Programming Error 11.7

Throwing an exception from a __finally block can be dangerous. If an uncaught exception is awaiting processing when the __finally block executes and the __finally block throws a new exception that is not caught in the __finally block, the first exception is lost, and the new exception is the one passed to the next enclosing try block.

Error-Prevention Tip 11.5

When placing code that can throw an exception in a __finally block, always enclose that code in a try…catch sequence that catches the appropriate exception types. This prevents losing uncaught and rethrown exceptions that occur before the __finally block executes.

Performance Tip 11.4

Adding an excessive number of try-catch-__finally blocks in your code can result in reduced performance.

11.6 Exception Properties

As we discussed in Section 11.4, exception data types derive from class Exception, which has several properties. These properties frequently are used to formulate error messages for a caught exception. Two important properties are Message and StackTrace. Property Message stores the error message associated with an Exception object. This message may be a default message associated with the exception type or a customized message passed to an exception object’s constructor when the exception object is constructed. Property StackTrace contains a string that represents the method call stack. The runtime environment keeps a list of method calls that have been made up to a given moment. The StackTrace string represents this sequential list of methods that had not finished processing at the time the exception occurred. The exact location at which the exception occurs in the program is called the exception’s throw point.

Error-Prevention Tip 11.6

A stack trace shows the complete method call stack at the time an exception occurred. This lets the programmer view the series of method calls that led to the exception. Information in the stack trace includes names of the methods on the call stack at the time of the exception, names of the classes in which those methods are defined, names of the namespaces in which those classes are defined and line numbers. The first line number in the stack trace indicates the throw point. Subsequent line numbers indicate the locations from which each method in the stack trace was called.

Another property used frequently by class library programmers is InnerException. Typically, programmers use this property to “wrap” exception objects caught in their code, then throw new exception types that are specific to their libraries. For example, a programmer implementing an accounting system might have some account-number processing code in which account numbers are input as strings, but represented with integers in the code. As you know, a program can convert strings to Int32 values with Convert::ToInt32, which throws a FormatException when it encounters an invalid number format. When an invalid account-number format occurs, the accounting-system programmer might wish either to indicate an error message different from the default one supplied by FormatException or to indicate a new exception type, such as InvalidAccountNumberFormatException. In these cases, the programmer would provide code to catch the FormatException, then
create an exception object in thecatchhandler, passing the original exception as one of the
constructor arguments. The original exception object becomes theInnerException of the
new exception object. When anInvalidAccountNumberFormatExceptionoccurs in
code that uses the accounting-system library, thecatch handler that catches the exception
can view the original exception via the propertyInnerException. Thus, the exception indi-
cates that an invalid account number was specified and that the particular problem was an
invalid number format.

Our next example (Fig. 11.3–Fig. 11.5) demonstrates propertiesMessage, Stack-
Trace andInnerException, as well as methodToString. In addition, this example
demonstrates stack unwinding—the process that attempts to locate an appropriatecatch
handler for an uncaught exception. As we discuss this example, we keep track of the
methods on the call stack, so we can discuss propertyStackTrace and the stack-
unwinding mechanism.

```cpp
// Fig. 11.3: Properties.h
// Stack unwinding and Exception class properties.
#pragma once

using namespace System;

// demonstrates using the Message, StackTrace and
// InnerException properties
public __gc class Properties
{
  public:
    static void Method1();
    static void Method2();
    static void Method3();
}; // end class Properties
```

Fig. 11.3 Exception properties and stack unwinding.

```cpp
// Fig. 11.4: Properties.cpp
// Method definitions for class Properties.

#include "stdafx.h"
#include "Properties.h"

// calls Method2
void Properties::Method2()
{
  Method2();
} // end method Method2

// calls Method3
void Properties::Method3()
{
```
Method3();
} // end method Method2
// throws an Exception containing an InnerException
void Properties::Method3()
{
    // attempt to convert non-integer string to int
    try {
        Convert::ToInt32( S"Not an integer" );
    } // end try
    // catch FormatException and wrap it in new Exception
    catch ( FormatException *error ) {
        throw new Exception( S"Exception occurred in Method3", error );
    } // end try
} // end method Method3

// Fig. 11.5: PropertiesTest.cpp
// PropertiesTest demonstrates stack unwinding. (Part 1 of 2.)
#include "stdafx.h"
#include "Properties.h"

using namespace System;

int _tmain()
{
    // calls Method1, any Exception it generates will be
    // caught in the catch handler that follows
    try {
        Properties::Method1();
    } // end try

    // output string representation of Exception, then
    // output values of InnerException, Message,
    // and StackTrace properties
    catch( Exception *exception ) {
        Console::WriteLine( S"exception->ToString(): 
{0}
", exception->ToString() );
        Console::WriteLine( S"exception->Message: 
{0}
", exception->Message );
        Console::WriteLine( S"exception->StackTrace: 
{0}\n", exception->StackTrace );
    } // end catch
} // end _tmain

// Fig. 11.5: PropertiesTest.cpp
// PropertiesTest demonstrates stack unwinding. (Part 1 of 2.)
Program execution begins with the invocation of _tmain (Fig. 11.5), which becomes the first method on the method call stack (_tmain is actually a function, but both functions and methods can be added to the method call stack). Line 17 of the try block in _tmain invokes Method1 (lines 8–11 of Fig. 11.4), which becomes the second method on the stack. If Method1 throws an exception, the catch handler in lines 23–35 (Fig. 11.5) handles the exception.
exception and outputs information about the exception that occurred. Line 10 of Method1 (Fig. 11.4) invokes Method2 (lines 14–17), which becomes the third method on the stack. Then line 16 of Method2 invokes Method3 (defined in lines 20–31), which becomes the fourth method on the stack.

**Error-Prevention Tip 11.7**
When reading a stack trace, start from the top of the stack trace and read the error message first. Then, read the remainder of the stack trace, looking for the first line that indicates code that you wrote in your program. Normally, this is the location that caused the exception.

At this point, the method call stack for the program is

```
Method3
Method2
Method1
_tmain
```

with the last method called (Method3) at the top and the first method called (_tmain) at the bottom. The try block (lines 23–25 of Fig. 11.4) in Method3 invokes method Convert::ToInt32 (line 24) and attempts to convert a string to an int. At this point, Convert::ToInt32 becomes the fifth and final method on the call stack.

The argument to Convert::ToInt32 is not in integer format, so line 24 of Fig. 11.4 throws a FormatException that is caught in line 28 in Method3. The exception terminates the call to Convert::ToInt32, so the method is removed from the method-call stack. The catch handler creates an Exception object, then throws it. The first argument to the Exception constructor is the custom error message for our example, "Exception occurred in Method3". The second argument is the InnerException object—the FormatException that was caught. Note that the StackTrace for this new exception object will reflect the point at which the exception was thrown (line 29). Now, Method3 terminates, because the exception thrown in the catch handler is not caught in the method body. Thus, control will be returned to the statement that invoked Method3 in the prior method in the call stack (Method2). This unwinds (removes) Method3 from the method-call stack.

**Good Programming Practice 11.3**

When catching and rethrowing an exception, provide additional debugging information in the rethrown exception. To do so, create an Exception object with more specific debugging information and pass the original caught exception to the new exception object's constructor to initialize the InnerException property.8

When control returns to line 16 in Method2, the CLR determines that line 16 is not in a try block. Therefore, the exception cannot be caught in Method2, and Method2 terminates. This unwinds Method2 from the method-call stack and returns control to line 10 in Method1. Here again, line 10 is not in a try block, so the exception cannot be caught in Method1. The method terminates and unwinds from the call stack, returning control to line 17 in _tmain (Fig. 11.5), which is in a try block. The try block in _tmain expires, and

---

the catch handler in lines 23–35 catches the exception. The catch handler uses method
ToString and properties Message, StackTrace and InnerException to produce the
output. Note that stack unwinding continues until either a catch handler catches the exception or the program terminates.

The first block of output (reformatted for readability) in Fig. 11.5 shows the exception’s
string representation returned from method ToString. This begins with the name of the
exception class followed by the Message property value. The next eight lines show the
string representation of the InnerException object. The remainder of that block of output
shows the StackTrace for the exception thrown in Method3. Note that the StackTrace
represents the state of the method-call stack at the throw point of the exception, not at the
point where the exception eventually is caught. Each of the StackTrace lines that begins
with "at" represents a method on the call stack. These lines indicate the method in which
the exception occurred, the file in which that method resides and the line number in the file.
Also, note that the stack trace includes the inner-exception stack trace. [Note: File and line
number information is not shown for FCL classes and methods.]

The next block of output (two lines) simply displays the Message property (Exception
occurred in Method3) of the exception thrown in Method3.

The third block of output displays the StackTrace property of the exception thrown in Method3. Note that the StackTrace property includes the stack trace starting from line
29 in Method3, because that is the point at which the Exception object was created and
thrown. The stack trace always begins from the exception’s throw point.

Finally, the last block of output displays the ToString representation of the Inner-
Exception property, which includes the namespace and class names of that exception
object, its Message property and its StackTrace property.

### 11.7 Programmer-Defined Exception Classes

In many cases, programmers can use existing exception classes from the .NET Framework
to indicate exceptions that occur in their programs. However, in some cases, programmers
may wish to create exception types that are more specific to the problems that occur in their
programs. **Programmer-defined exception classes** should derive directly or indirectly from
class ApplicationException of namespace System.

**Good Programming Practice 11.4**

*Associating each type of malfunction with an appropriately named exception class improves program clarity.*

**Software Engineering Observation 11.9**

*Before creating programmer-defined exception classes, investigate the existing exception classes in the .NET Framework to determine whether an appropriate exception type already exists.*

**Software Engineering Observation 11.10**

*Programmers should create exception classes only if they need to catch and handle the new exceptions differently from other existing exception types.*

**Software Engineering Observation 11.11**

*Always catch any exception class that you create and throw.*
Good Programming Practice 11.5

Only handle one problem per exception class. Never overload an exception class to deal with several exceptions.

Good Programming Practice 11.6

Use significant and meaningful names for exception classes.

Figures 11.6–11.8 demonstrate defining and using a programmer-defined exception class. Class NegativeNumberException (Fig. 11.6–Fig. 11.7) is a programmer-defined exception class representing exceptions that occur when a program performs an illegal operation on a negative number, such as the square root of a negative number.

```cpp
// Fig. 11.6: NegativeNumberException.h
// NegativeNumberException represents exceptions caused by illegal
// operations performed on negative numbers.
#pragma once

using namespace System;

// NegativeNumberException represents exceptions caused by
// illegal operations performed on negative numbers
public __gc class NegativeNumberException : public ApplicationException
{

public:
    NegativeNumberException();
    NegativeNumberException( String * );
    NegativeNumberException( String *, Exception * );
}; // end class NegativeNumberException
```

Fig. 11.6  
ApplicationException subclass thrown when a program performs illegal operations on negative numbers.

```cpp
// Fig. 11.7: NegativeNumberException.cpp
// Method definitions for class NegativeNumberException.
#include "stdafx.h"
#include "NegativeNumberException.h"

// default constructor
NegativeNumberException::NegativeNumberException()
    : ApplicationException( $"Illegal operation for a negative number" )
{
}
```

Fig. 11.7  
NegativeNumberException class method definitions. (Part 1 of 2)
Programmer-defined exceptions should extend class `ApplicationException`, should have a class name that ends with "Exception" and should define three constructors—a default constructor, a constructor that receives a string argument (the error message) and a constructor that receives a string argument and an `Exception` argument (the error message and the inner-exception object).

NegativeNumberExceptions most likely occur during arithmetic operations, so it seems logical to derive class `NegativeNumberException` from class `ArithmeticException`. However, class `ArithmeticException` derives from class `SystemException`—the category of exceptions thrown by the CLR. `ApplicationException` specifically is the base class for exceptions thrown by a user program, not by the CLR.

Figure 11.8 demonstrates our programmer-defined exception class. The application enables the user to input a numeric value, then invokes function `FindSquareRoot` (lines 41–50) to calculate the square root of that value. For this purpose, `FindSquareRoot` invokes class Math’s `Sqrt` method, which receives a positive double value as its argument. If the argument is negative, method `Sqrt` normally returns constant `NaN`—“not a number”—from class `Double`. In this program, we would like to prevent the user from calculating the square root of a negative number. If the numeric value received from the user is negative, `FindSquareRoot` throws a `NegativeNumberException` (lines 45–46). Otherwise, `FindSquareRoot` invokes class Math’s `Sqrt` method to compute the square root.

```cpp
// Fig. 11.7: NegativeNumberException class method definitions. (Part 2 of 2)

NegativeNumberException::NegativeNumberException( String *message )
    : ApplicationException( message )
{
}

NegativeNumberException::NegativeNumberException( String *message,
    Exception *inner ) : ApplicationException( message, inner )
{
}
```

```
1 // Fig. 11.8: SquareRoot.cpp
2 // Demonstrating a programmer-defined exception class.
3 #include "stdafx.h"
4 #include "mscorlib.dll"
5 #include "SystemException".
6 #include <mscorlib.dll>
7 using namespace System;

Fig. 11.8 FindSquareRoot function throws exception if error occurs when calculating square root. (Part 1 of 2)


©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
```csharp
double FindSquareRoot( double );

int _tmain()
{
    // catch any NegativeNumberExceptions thrown
    try {
        Console::Write( S"Please enter a number: " );

        double result =
            FindSquareRoot( Double::Parse( Console::ReadLine() ) );

        Console::WriteLine( result );
    } // end try

    // process invalid number format
    catch ( FormatException *notInteger ) {
        Console::WriteLine( notInteger->Message );
    } // end catch

    // display message if negative number input
    catch ( NegativeNumberException *error ) {
        Console::WriteLine( error->Message );
    } // end catch

    return 0;
} // end _tmain

// computes the square root of its parameter; throws
// NegativeNumberException if parameter is negative
double FindSquareRoot( double operand )
{
    // if negative operand, throw NegativeNumberException
    if ( operand < 0 )
        throw new NegativeNumberException(
            S"Square root of negative number not permitted." );

    // compute the square root
    return Math::Sqrt( operand );
} // end function FindSquareRoot
```

Please enter a number: 33
5.74456264653803

Please enter a number: hello
Input string was not in a correct format.

Please enter a number: -12.45
Square root of negative number not permitted.

Fig. 11.8 FindSquareRoot function throws exception if error occurs when calculating square root. (Part 2 of 2.)
The try block (lines 17–24) attempts to invoke FindSquareRoot with the value input by the user. If the user input is not a valid number, a FormatException occurs, and the catch handler in lines 27–29 processes the exception. If the user inputs a negative number, function FindSquareRoot throws a NegativeNumberException (lines 45–46). The catch handler in lines 32–33 catches and handles that exception.

**SUMMARY**

- An exception is an indication of a problem that occurs during a program’s execution.
- Exception handling enables programmers to create applications that can resolve exceptions, often allowing a program to continue execution as if no problems were encountered.
- Exception handling enables programmers to write clear, robust and more fault-tolerant programs.
- Exception handling enables programmers to remove error-handling code from the “main line” of the program’s execution. This improves program clarity and enhances modifiability.
- Exception handling is designed to process synchronous errors, such as out-of-range array subscripts, arithmetic overflow, division by zero, invalid method parameters and memory exhaustion, and asynchronous events, such as disk I/O completion.
- When a method called in a program or the CLR detects a problem, the method or CLR throws an exception. The point in the program at which an exception occurs is called the throw point.
- There is no guarantee that there will be an exception handler to process that kind of exception. If there is, the exception will be caught and handled.
- A try block consists of keyword try followed by braces ({} that delimit a block of code in which exceptions could occur.
- Immediately following the try block are zero or more catch handlers. Each catch specifies in parentheses an exception parameter representing the exception type the catch can handle.
- If an exception parameter includes an optional parameter name, the catch handler can use that parameter name to interact with a caught exception object.
- There can be one parameterless catch handler that catches all exception types.
- After the last catch handler, an optional __finally block contains code that always executes, regardless of whether an exception occurs.
- MC++ uses the termination model of exception handling. If an exception occurs in a try block, the block expires and program control transfers to the first catch handler following the try block that can process the type of exception that occurred.
- The appropriate handler is the first one in which the thrown exception’s type matches, or is derived from, the exception type specified by the catch handler’s exception parameter.
- If no exceptions occur in a try block, the CLR ignores the exception handlers for that block.
- If no exceptions occur or if an exception is caught and handled, the program resumes execution with the next statement after the try...catch...__finally sequence.
- If an exception occurs in a statement that is not in a try block, the method containing that statement terminates immediately—a process called stack unwinding.
- The MC++ exception-handling mechanism allows only objects of class Exception and its derived classes to be thrown and caught. Class Exception of namespace System is the base class of the .NET Framework exception hierarchy.
- Exceptions are objects of classes that inherit directly or indirectly from class Exception.
**TERMINOLOGY**

<table>
<thead>
<tr>
<th>ApplicationException class</th>
<th>ArgumentOutOfRangeException</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic overflow</td>
<td>out-of-range array subscript</td>
</tr>
<tr>
<td>call stack</td>
<td>overflow</td>
</tr>
<tr>
<td>catch all exception types</td>
<td>OverflowException class</td>
</tr>
<tr>
<td>catch block (or handler)</td>
<td>polymorphic processing of related exceptions</td>
</tr>
<tr>
<td>disk I/O completion</td>
<td>positive infinity</td>
</tr>
<tr>
<td>divide by zero</td>
<td>programmer-defined exception classes</td>
</tr>
<tr>
<td>DivideByZeroException class</td>
<td>release a resource</td>
</tr>
<tr>
<td>Double class</td>
<td>resource leak</td>
</tr>
<tr>
<td>eliminate resource leaks</td>
<td>result of an uncaught exception</td>
</tr>
<tr>
<td>error-processing code</td>
<td>resumption model of exception handling</td>
</tr>
<tr>
<td>exception</td>
<td>rthrow an exception</td>
</tr>
<tr>
<td>Exception class</td>
<td>runtime exception</td>
</tr>
<tr>
<td>exception handler</td>
<td>stack unwinding</td>
</tr>
<tr>
<td>fault-tolerant program</td>
<td>StackTrace property of Exception</td>
</tr>
<tr>
<td>__finally block</td>
<td>synchronous error</td>
</tr>
<tr>
<td>FormatException class</td>
<td>SystemException class</td>
</tr>
<tr>
<td>IndexOutOfRangeException class</td>
<td>termination model of exception handling</td>
</tr>
<tr>
<td>inheritance with exceptions</td>
<td>throw an exception</td>
</tr>
<tr>
<td>InnerException property of Exception</td>
<td>throw point</td>
</tr>
<tr>
<td>memory leak</td>
<td>throw statement</td>
</tr>
<tr>
<td>Message property of class Exception</td>
<td>ToInt32 method of Convert</td>
</tr>
<tr>
<td>method call stack</td>
<td>try block</td>
</tr>
<tr>
<td>NaN constant of class Double</td>
<td>uncaught exception</td>
</tr>
<tr>
<td>negative infinity</td>
<td></td>
</tr>
</tbody>
</table>

**SELF-REVIEW EXERCISES**

11.1 Fill in the blanks in each of the following statements:

a) A method _______ an exception when that method detects that a problem occurred.
b) The _______ block associated with a try block always executes.
c) Exception classes in MC++ are derived from class _______.
d) The statement that throws an exception is called the _______ of the exception.
e) A(n) _______ block encloses code that might throw an exception and code that should not execute if an exception occurs.
f) If the catch-all exception handler is declared before another exception handler, a(n) _______ occurs.
g) An uncaught exception in a method causes that method to _______ from the method-call stack.
h) Method `Convert::ToInt32` can throw a(n) _______ exception if its argument is not a valid integer value.
i) Runtime exceptions derive from class _______.
j) The _______ property of class `Exception` represents the state of the method-call stack at the throw point of an exception.

11.2 State whether each of the following is true or false. If false, explain why.

a) Exceptions always are handled in the method that initially detects the exception.
b) Programmer-defined exception classes should extend class `SystemException`.
c) Accessing an out-of-bounds array subscript causes the CLR to throw an exception.
d) A __finally block is optional after a try block.

©2004 Deitel & Associates, Inc. and Pearson Education. All Rights Reserved.
e) If a __finally block appears in a method, that __finally block is guaranteed to execute.
f) Returning to the throw point of an exception using keyword return is possible.
g) Exceptions can be rethrown.
h) MC++ exceptions are objects of classes that inherit directly or indirectly from class Exception.
i) Property Message of class Exception returns a String * indicating the method from which the exception was thrown.
j) Exceptions can be thrown only by methods explicitly called in a try block.

**ANSWERS TO SELF-REVIEW EXERCISES**

11.1  a) throws. b) __finally. c) Exception. d) throw point. e) try. f) syntax error. g) unwind. 
     h) FormatException. i) SystemException. j) StackTrace.

11.2  a) False. Exceptions are handled by the first matching catch handler, which could follow an enclosing try block. b) False. Programmer-defined exception classes should extend class ApplicationException. c) True. d) False. The __finally block is an option only if there is at least one catch handler. If there are no catch handlers, the __finally block is required. e) False. The __finally block will execute only if program control enters the corresponding try block. f) False. Keyword return causes control to return to the calling method. g) True. h) True. i) False. Property Message returns a String * representing the error message. j) False. Exceptions can be thrown by any method, called from a try block or not. Also, the CLR can throw exceptions.

**EXERCISES**

11.3 Use inheritance to create an exception base class and various exception-derived classes. Write a program to demonstrate that the catch specifying the base class catches derived-class exceptions.

11.4 Write an MC++ program that demonstrates how various exceptions are caught with

```cpp
    catch ( Exception *exception )
```

11.5 Write an MC++ program that shows the importance of the order of exception handlers. Write two programs: One with the correct order of catch handlers, and one with an incorrect order (i.e., place the base class exception handler before the derived-class exception handlers). Show that if you attempt to catch a base-class exception type before a derived-class exception type, the derived-class exceptions are not invoked (which potentially yield logical errors in routine). Explain why these errors occur.

11.6 Exceptions can be used to indicate problems that occur when an object is being constructed. Write an MC++ program that shows a constructor passing information about constructor failure to an exception handler that occurs after a try block. The exception thrown also should contain the arguments sent to the constructor.

11.7 Write an MC++ program that demonstrates rethrowing an exception.

11.8 Write an MC++ program that shows that a method with its own try block does not have to catch every possible exception that occurs within the try block. Some exceptions can slip through to, and be handled in, other scopes.