

Java Programming, 9e

Chapter 12

Exception Handling





Objectives(1 of 2)

- Describe exceptions
- Try code and catch exceptions
- Throw and catch multiple exceptions
- Use the `finally` block
- Appreciate the advantages of exception handling



Objectives (2 of 2)

- Specify the exceptions that a method can throw
- Trace exceptions through the call stack
- Create your own `Exception` classes
- Use an assertion
- Display a virtual keyboard



Learning About Exceptions (1 of 10)

- **Exceptions**

- Unexpected or error conditions
- Not usual occurrences
- Causes:
 - The program issues a call to a file that does not exist
 - The program attempts to write to a full disk
 - The user enters invalid data
 - The program attempts to divide a value by 0



Learning About Exceptions (2 of 10)

- **Exception handling**
 - Object-oriented techniques used to manage `Exception` errors
 - **Runtime exceptions**
- Exceptions
 - Objects
 - Descend from the `Throwable` class



Learning About Exceptions (3 of 10)

```
java.lang.Object
|
+--java.lang.Throwable
|
+--java.lang.Exception
|   |
|   +--java.io.IOException
|   |
|   +--java.lang.RuntimeException
```

Figure 12-1 The Exception and Error class inheritance hierarchy (*continues*)



Learning About Exceptions (4 of 10)

(continued)



Figure 12-1 The Exception and Error class inheritance hierarchy



Learning About Exceptions (5 of 10)

- **Error class**

- Represents serious errors from which a program usually cannot recover
- Can cause program to **crash**
- `Error` condition
 - A program runs out of memory
 - A program cannot locate a required class



Learning About Exceptions (6 of 10)

- **Exception class**
 - Less serious errors
 - Unusual conditions
 - A program can recover from this type of error
- `Exception` class errors
 - An invalid array subscript
 - Performing illegal arithmetic operations



Learning About Exceptions (7 of 10)

```
import java.util.Scanner;
public class Division
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        int numerator, denominator, result;
        System.out.print("Enter numerator >> ");
        numerator = input.nextInt();
        System.out.print("Enter denominator >> ");
        denominator = input.nextInt();
        result = numerator / denominator;
        System.out.println(numerator + " / " + denominator +
            " = " + result);
    }
}
```

Figure 12-2 The Division class



Learning About Exceptions (8 of 10)

```
Enter numerator >> 12
Enter denominator >> 4
12 / 4 = 3
```

```
Enter numerator >> 12
Enter denominator >> 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Division.main(Division.java:12)
```

Figure 12-3 Two typical executions of the Division application



Learning About Exceptions (9 of 10)

- You do not necessarily have to deal with an exception
 - Let the offending program terminate
 - But doing so is abrupt and unforgiving
- You can write programs without using exception-handling techniques
 - Use a decision to avoid an error
- Exception handling provides a more elegant solution for handling error conditions



Learning About Exceptions (10 of 10)

- **Stack trace**
 - List of methods called prior to crash
- **Mission critical**
 - Any process that is crucial to an organization, e.g., air traffic control
- **Fault-tolerant**
 - Designed to continue to operate when some part of the system fails
- **Robustness**
 - Represents the degree to which a system is resilient to stress



Trying Code and Catching Exceptions (1 of 6)

- **try block**
 - A segment of code in which something might go wrong
 - Attempts to execute
 - Acknowledges an exception might occur
- A `try` block includes:
 - The keyword `try`
 - Opening and closing curly braces
 - Executable statements, which might cause an exception



Trying Code and Catching Exceptions (2 of 6)

- **catch block**

- A segment of code
- Immediately follows a `try` block
- Handles an exception thrown by the `try` block preceding it
- Can “catch” an Object of type `Exception` or an `Exception` child class

- **throw statement**

- Sends an `Exception` object out of a block or method so it can be handled elsewhere



Trying Code and Catching Exceptions (3 of 6)

- A `catch` block includes:
 - The keyword `catch`
 - Opening and closing parentheses
 - An `Exception` type
 - A name for an instance of the `Exception` type
 - Opening and closing curly braces
 - Statements to handle the error condition



Trying Code and Catching Exceptions (4 of 6)

```
returnType methodName(optional arguments)
{
    // optional statements prior to code that is tried
    try
    {
        // statement or statements that might generate an exception
    }
    catch(Exception someException)
    {
        // actions to take if exception occurs
    }
    // optional statements that occur after try,
    // whether catch block executes or not
}
```

Figure 12-6 Format of try...catch pair



Trying Code and Catching Exceptions (5 of 6)

- If no exception occurs within the `try` block, the `catch` block does not execute
- `getMessage()` method
 - Obtains information about the exception
- Within a `catch` block, you might want to add code to correct the error



Trying Code and Catching Exceptions (6 of 6)

```
import java.util.Scanner;
public class DivisionMistakeCaught
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        int numerator, denominator, result;
        System.out.print("Enter numerator >> ");
        numerator = input.nextInt();
        System.out.print("Enter denominator >> ");
        denominator = input.nextInt();
        try
        {
            result = numerator / denominator;
            System.out.println(numerator + " / " + denominator +
                               " = " + result);
        }
        catch(ArithmeticException mistake)
        {
            System.out.println("Arithmetic exception was thrown and caught");
        }
        System.out.println("End of program");
    }
}
```

Figure 12-7 The DivisionMistakeCaught application



Using a `try` Block to Make Programs “Foolproof” (1 of 3)

- It is useful to circumvent data entry errors
 - Handle potential data conversion exceptions caused by careless users
- Using a `nextLine()` call will account for potential remaining characters in the input buffer



Using a `try` Block to Make Programs “Foolproof” (2 of 3)

```
import java.util.Scanner;
public class EnteringIntegers
{
    public static void main(String[] args)
    {
        int[] numberList = {0, 0, 0, 0, 0, 0};
        int x;
        Scanner input = new Scanner(System.in);
        for(x = 0; x < numberList.length; ++x)
        {
            try
            {
                System.out.print("Enter an integer >> ");
                numberList[x] = input.nextInt();
            }
            catch (Exception e)
            {
                System.out.println("Exception occurred");
            }
            // input.nextLine();
        }
        System.out.print("The numbers are : ");
        for(x = 0; x < numberList.length; ++x)
            System.out.print(numberList[x] + " ");
        System.out.println();
    }
}
```

This line is commented out for the execution in Figure 12-13, but not Figure 12-14.

Figure 12-12 The EnteringIntegers program without the extra `nextLine()` call



Using a `try` Block to Make Programs “Foolproof” (3 of 3)

```
Enter an integer >> 22
Enter an integer >> 33
Enter an integer >> ww
Exception occurred
Enter an integer >> 44
Enter an integer >> xx
Exception occurred
Enter an integer >> 55
The numbers are: 22 33 0 44 0 55
```

Figure 12-14 A typical execution of the `EnteringIntegers` program with the extra `nextLine()` call



Declaring and Initializing Variables in `try..catch` Blocks (1 of 3)

- A variable declared within a block is local to that block
 - It goes out of scope when the `try` or `catch` block ends
- If the conversion fails, an exception is thrown



Declaring and Initializing Variables in try..catch Blocks (2 of 3)

```
import java.util.Scanner;
public class UninitializedVariableTest
{
    public static void main(String[] args)
    {
        int x;
        Scanner input = new Scanner(System.in);
        try
        {
            System.out.print("Enter an integer >> ");
            x = input.nextInt();
        }
        catch(Exception e)
        {
            System.out.println("Exception occurred");
        }
        System.out.println("x is " + x);
    }
}
```

Figure 12-15 The UninitializedVariableTest program



Declaring and Initializing Variables in try...catch Blocks (3 of 3)

```
UninitializedVariableTest.java:17: error: variable x might not have been initialized
    System.out.println("x is " + x);
                           ^
1 error
```

Figure 12-16 The error message generated when compiling the UninitializedVariableTest program



Throwing and Catching Multiple Exceptions (1 of 4)

- You can place multiple statements within a `try` block
 - Only the first error-generating statement throws an exception
- When a program contains multiple `catch` blocks:
 - They are examined in sequence until a match is found for the `Exception` type
 - The matching `catch` block executes
 - Each remaining `catch` block is bypassed



Throwing and Catching Multiple Exceptions (2 of 4)

```
import java.util.*;
public class DivisionMistakeCaught3
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        int numerator, denominator, result;
        try
        {
            System.out.print("Enter numerator >> ");
            numerator = input.nextInt();
            System.out.print("Enter denominator >> ");
            denominator = input.nextInt();
            result = numerator / denominator;
            System.out.println(numerator + " / " + denominator +
                               " = " + result);
        }
        catch(ArithmeticException mistake)
        {
            System.out.println(mistake.getMessage());
        }
        catch(InputMismatchException mistake)
        {
            System.out.println("Wrong data type");
        }
        System.out.println("End of program");
    }
}
```

An `InputMismatchException` might be thrown here.

An `ArithmeticException` might be thrown here.

Figure 12-18 The `DivisionMistakeCaught3` class



Throwing and Catching Multiple Exceptions (3 of 4)

- “Catch-all” block
 - Accepts a more generic `Exception` argument type:
`catch (Exception e)`
- Unreachable code
 - Program statements that can never execute under any circumstances
- A `catch` block can be written to catch multiple `Exception` types
- Considered poor style for a method to throw more than three or four `Exception` types



Throwing and Catching Multiple Exceptions

(4 of 4)

```
import java.util.*;
public class DivisionMistakeCaught4
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        int numerator, denominator, result;
        try
        {
            System.out.print("Enter numerator >> ");
            numerator = input.nextInt();
            System.out.print("Enter denominator >> ");
            denominator = input.nextInt();
            result = numerator / denominator;
            System.out.println(numerator + " / " + denominator +
                               " = " + result);
        }
        catch(Exception mistake)
        {
            System.out.println("Operation unsuccessful");
        }
        System.out.println("End of program");
    }
}
```

This catch block
can handle all
Exception types.

Figure 12-20 The DivisionMistakeCaught4 application



Using the `finally` Block (1 of 5)

- **`finally` block**
 - Use for actions you must perform at the end of a `try...catch` sequence
 - Use to perform cleanup tasks
 - Executes regardless of whether the preceding `try` block identifies an exception



Using the `finally` Block (2 of 5)

```
try
{
    // statements to try
}
catch(Exception e)
{
    // actions that occur if exception was thrown
}
finally
{
    // actions that occur whether catch block executed or not
}
```

Figure 12-24 Format of `try...catch...finally` sequence



Using the `finally` Block (3 of 5)

- When the `try` code fails:
 - It throws an exception
 - The `Exception` object is caught
 - The `catch` block executes
 - Control passes to statements at the end of the method



Using the `finally` Block (4 of 5)

- Reasons the final set of statements might never execute:
 - An unplanned exception might occur
 - The `try` or `catch` block might contain a `System.exit()` ; statement
- The `try` block might throw an `Exception` object for which you did not provide a `catch` block
 - Program execution stops immediately
 - The exception is sent to the operating system for handling
 - The current method is abandoned



Using the `finally` Block (5 of 5)

- When the `finally` block is used, `finally` statements execute before the method is abandoned
- The `finally` block executes no matter what outcome of the `try` block occurs
 - The `try` ends normally
 - The `catch` executes
 - The exception causes the method to abandon prematurely



Understanding the Advantages of Exception Handling (1 of 4)

- Before object-oriented programming languages, errors were handled with confusing, error-prone methods
 - When any method fails, the program sets appropriate error code
 - Difficult to follow
 - The application's purpose and intended outcome are lost in a maze of `if` statements
 - Coding mistakes are made because of complicated nesting



Understanding the Advantages of Exception Handling (2 of 4)

```
call methodA()
if methodA() worked
{
    call methodB()
    if methodB() worked
    {
        call methodC()
        if methodC() worked
            everything's okay, so display finalResult
        else
            set errorCode to 'C'
    }
    else
        set errorCode to 'B'
}
else
    set errorCode to 'A'
```

Figure 12-26 Pseudocode representing traditional error checking



Understanding the Advantages of Exception Handling (3 of 4)


```
try
{
    call methodA() and maybe throw an exception
    call methodB() and maybe throw an exception
    call methodC() and maybe throw an exception
    everything's okay, so display finalResult
}
catch(methodA()'s error)
{
    set errorCode to "A"
}
catch(methodB()'s error)
{
    set errorCode to "B"
}
catch(methodC()'s error)
{
    set errorCode to "C"
}
```

Figure 12-27 Pseudocode representing object-oriented exception handling



Understanding the Advantages of Exception Handling (4 of 4)

- Java's object-oriented, error-handling technique
 - Statements of the program that do the “real” work are placed together, where their logic is easy to follow
 - Unusual, exceptional events are grouped and moved out of the way
- An advantage to object-oriented exception handling is flexibility in handling of error situations
- Appropriately deal with exceptions as you decide how to handle them



Specifying the Exceptions That a Method Can Throw (1 of 3)

- If a method throws an exception that it will not catch but will be caught by a different method, use the keyword throws followed by the Exception type in the method header
- **Exception specification**
 - Lists exceptions that a method may throw
- Every Java method has the potential to throw an exception
 - For most Java methods, do not use the throws clause
 - Let Java handle any exception by shutting down the program
 - Most exceptions never have to be explicitly thrown or caught



Specifying the Exceptions That a Method Can Throw (2 of 3)

- **Unchecked exceptions**

- Inherit from the `Error` class or the `RuntimeException` class
- You are not required to handle these exceptions
 - You can simply let the program terminate
 - An example is dividing by zero

- **Checked exceptions**

- Programmers should anticipate checked exceptions
- Programs should be able to recover from them



Specifying the Exceptions That a Method Can Throw (3 of 3)

- If you throw a checked exception from a method, you must do one of the following:
 - Catch it
 - Declare the exception in the method header's `throws` clause
- To use a method to its full potential, you must know:
 - Method's name
 - Method's `return` type
 - Type and number of arguments the method requires
 - Type and number of exceptions the method throws



Tracing Exceptions Through the Call Stack

(1 of 3)

- **Call stack**

- The memory location where the computer stores the list of method locations to which the system must return
- When a method throws an exception:
 - The exception is thrown to the next method up the call stack
 - Methods are allowed to handle exceptions wherever the programmer has decided it is most appropriate
 - Including allowing the operating system to handle the error

Tracing Exceptions Through the Call Stack (2 of 3)

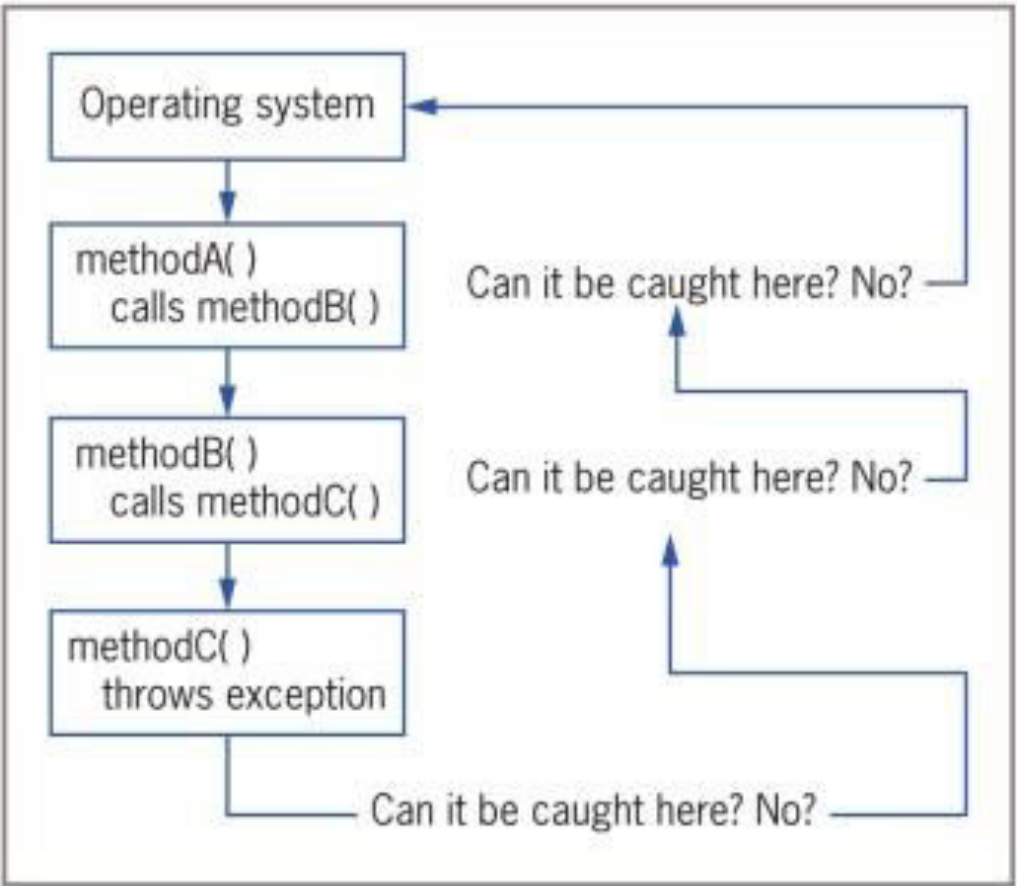


Figure 12-32 Cycling through the call stack



Tracing Exceptions Through the Call Stack

(3 of 3)

- `printStackTrace()` method
 - Displays a list of methods in the call stack
 - Determines the location of an exception
 - Is not placed in a finished program
 - Most useful for diagnosing problems



Creating Your Own Exception Classes

- Java provides over 40 categories of `Exceptions`
- Java allows you to create your own `Exception` classes
 - Extend a subclass of `Throwable`
- `Exception` class constructors:
 - `Exception()`
 - `Exception(String message)`
 - `Exception(String message, Throwable cause)`
 - `Exception(Throwable cause)`



Using Assertions (1 of 2)

- **Assertion**

- A Java language feature
- Detects logic errors
- Debugs programs

- **assert statement**

- Creates an assertion, such as:

```
assert booleanExpression : optionalErrorMessage
```

- The Boolean expression in the `assert` statement should always be `true` if the program is working correctly



Using Assertions (2 of 2)

- An `AssertionError` is thrown when a condition is `false`
- To enable an assertion, you must use the `-ea` option when executing a program



Displaying the Virtual Keyboard (1 of 3)

- A **virtual keyboard** is a computer keyboard that appears on the screen
- Allows user to point to and click keys with a mouse
 - With a touch screen user can touch keys with finger or stylus



Displaying the Virtual Keyboard (2 of 3)

```
import java.util.Scanner;
import java.io.IOException;
public class VirtualKeyboardDemo
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        try
        {
            Process proc = Runtime.getRuntime().exec
                ("cmd /c C:\\Windows\\System32\\osk.exe");
        }
        catch(IOException e)
        {
            System.out.println(e.getMessage());
        }
        String name;
        System.out.print("Enter name >> ");
        name = input.nextLine();
        System.out.println("Hello, " + name + "!");
    }
}
```

Figure 12-48 The VirtualKeyboardDemo application



Displaying the Virtual Keyboard (3 of 3)

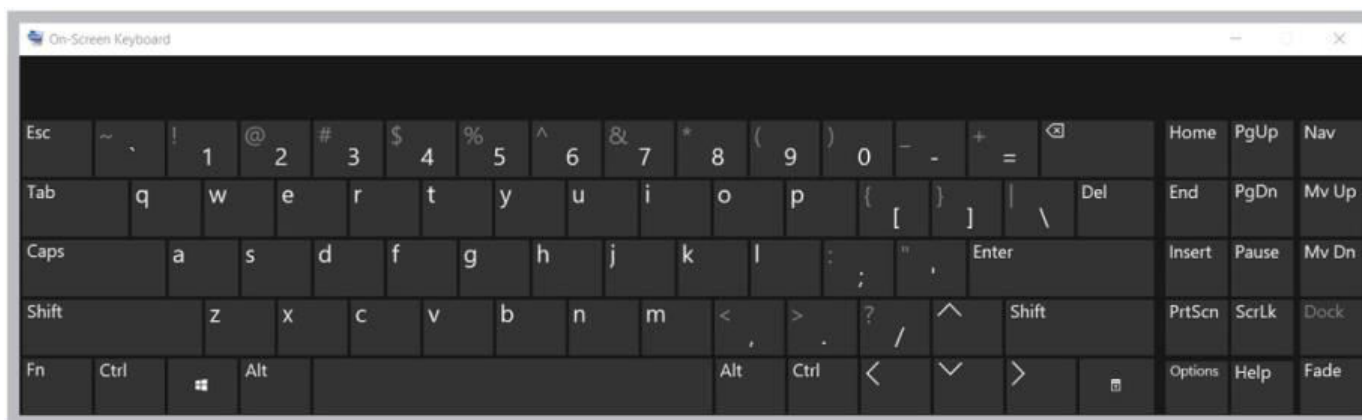


Figure 12-49 The VirtualKeyboardDemo program during execution



Don't Do It (1 of 2)

- Don't forget that all the statements in a `try` block might not execute
- Don't forget you might need a `nextLine()` method call after an attempt to read numeric data from the keyboard throws an exception
- Don't forget that a variable declared in a `try` block goes out of scope at the end of the block
- Don't forget that when a variables get its usable value within a `try` block, you must ensure that it has a valid value before attempting to use it



Don't Do It (2 of 2)

- Don't forget to place more specific `catch` blocks before more general ones
- Don't forget to write a `throws` clause for a method that throws an exception but does not handle it
- Don't forget to handle any checked exception thrown to your method



Summary (1 of 2)

- Exception
 - An unexpected or error condition
- Exception handling
 - Object-oriented techniques to manage errors
- Basic classes of errors: `Error` and `Exception`
- Exception–handling code
 - `try` block
 - `catch` block
 - `finally` block



Summary (2 of 2)

- Use the `throws <name>Exception` clause after the method header
 - Indicate the type of exception that might be thrown
- Call stack
 - A list of method locations to which the system must return
- Java provides over 40 categories of `Exceptions`
 - Create your own `Exception` classes
- Assertion
 - State a condition that should be `true`
 - Java throws an `AssertionError` when it is not
- You can call up Windows operating system programs such as the virtual keyboard using the `exec()` method in the `Runtime` class