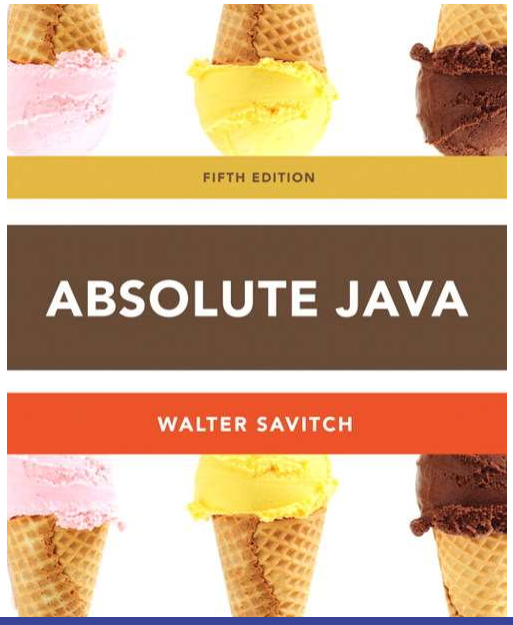



# Chapter 9

## Exception Handling

Slides prepared by Rose Williams, *Binghamton University*

Kenrick Mock, *University of Alaska Anchorage*

PEARSON ALWAYS LEARNING

## Introduction to Exception Handling

- Sometimes the best outcome can be when nothing unusual happens
- However, the case where exceptional things happen must also be prepared for
  - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur
  - Often the exception is some type of error condition

# Introduction to Exception Handling

- Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
  - This is called *throwing an exception*
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called *handling the exception*




# `try-throw-catch` Mechanism

- The basic way of handling exceptions in Java consists of the ***`try-throw-catch`*** trio
- The **`try`** block contains the code for the basic algorithm
  - It tells what to do when everything goes smoothly
- It is called a **`try`** block because it "tries" to execute the case where all goes as planned
  - It can also contain code that throws an exception if something unusual happens

```
try
{
   CodeThatMayThrowAnException
}
```

# try-throw-catch Mechanism

```
throw new
  ExceptionClassName(PossiblySomeArguments);
```

- When an exception is thrown, the execution of the surrounding **try** block is stopped
  - Normally, the flow of control is transferred to another portion of code known as the **catch** block
- The value thrown is the argument to the **throw** operator, and is always an object of some exception class
  - The execution of a **throw** statement is called *throwing an exception*




# try-throw-catch Mechanism

- A **throw** statement is similar to a method call:

```
throw new ExceptionClassName(SomeString);
```

  - In the above example, the object of class ***ExceptionClassName*** is created using a string as its argument
  - This object, which is an argument to the **throw** operator, is the exception object thrown
- Instead of calling a method, a **throw** statement calls a **catch** block

# try-throw-catch Mechanism

- When an exception is thrown, the **catch** block begins execution
  - The **catch** block has one parameter
  - The exception object thrown is plugged in for the **catch** block parameter
- The execution of the **catch** block is called *catching the exception*, or *handling the exception*
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by some **catch** block




# try-throw-catch Mechanism

```
catch(Exception e)
{
  ExceptionHandlingCode
}
```

- A **catch** block looks like a method definition that has a parameter of type **Exception** class
  - It is not really a method definition, however
- A **catch** block is a separate piece of code that is executed when a program encounters and executes a **throw** statement in the preceding **try** block
  - A **catch** block is often referred to as an *exception handler*
  - It can have at most one parameter

# try-throw-catch Mechanism

```
catch(Exception e) { . . . }
```

- The identifier **e** in the above **catch** block heading is called the **catch** block parameter
- The **catch** block parameter does two things:
  1. It specifies the type of thrown exception object that the **catch** block can catch (e.g., an **Exception** class object above)
  2. It provides a name (for the thrown object that is caught) on which it can operate in the **catch** block
     - Note: The identifier **e** is often used by convention, but any non-keyword identifier can be used




# try-throw-catch Mechanism

- When a **try** block is executed, two things can happen:
  1. No exception is thrown in the **try** block
     - The code in the **try** block is executed to the end of the block
     - The **catch** block is skipped
     - The execution continues with the code placed after the **catch** block

## `try-throw-catch` Mechanism

2. An exception is thrown in the `try` block and caught in the `catch` block
   - The rest of the code in the `try` block is skipped
   - Control is transferred to a following `catch` block (in simple cases)
   - The thrown object is plugged in for the `catch` block parameter
   - The code in the `catch` block is executed
   - The code that follows that `catch` block is executed (if any)




## Exception Example

- In many cases your own code doesn't throw the exception, but instead it is thrown by an existing Java library
- Example: Input an integer using `nextInt()`
  - What if the user doesn't enter an integer?
  - The `nextInt` method throws an `InputMismatchException`

## Exception Handling with the `Scanner` Class

- If a user enters something other than a well-formed `int` value, an `InputMismatchException` will be thrown
  - Unless this exception is caught, the program will end with an error message
  - If the exception is caught, the `catch` block can give code for some alternative action, such as asking the user to reenter the input




## The `InputMismatchException`

- The `InputMismatchException` is in the standard Java package `java.util`
  - A program that refers to it must use an `import` statement, such as the following:

```
import java.util.InputMismatchException;
```

- It is a descendent class of `RuntimeException`
  - Therefore, it is an unchecked exception and does not have to be caught in a `catch` block or declared in a `throws` clause
  - However, catching it in a `catch` block is allowed, and can sometimes be useful

# Tip: Exception Controlled Loops

- Sometimes it is better to simply loop through an action again when an exception is thrown, as follows:

```
boolean done = false;
while (! done)
{
  try
  {
    CodeThatMayThrowAnException
    done = true;
  }
  catch (SomeExceptionClass e)
  {
    SomeMoreCode
  }
}
```




# Exception Controlled Loop

Display 9.1 An Exception Controlled Loop

```
1  import java.util.Scanner;
2  import java.util.InputMismatchException;

3  public class InputMismatchExceptionDemo
4  {
5      public static void main(String[] args)
6      {
7          Scanner keyboard = new Scanner(System.in);
8          int number = 0; //to keep compiler happy
9          boolean done = false;

10         while (! done)
11         {
12             try
13             {
14               System.out.println("Enter a whole number:");
15               number = keyboard.nextInt();
16               done = true;
17             }
18             catch(InputMismatchException e)
19             {
20               keyboard.nextLine();
21               System.out.println("Not a correctly written whole
                 number.");
22               System.out.println("Try again.");
23             }
24         }

25         System.out.println("You entered " + number);
26     }
27 }
```

*If* `nextInt` *throws an exception, the* `try` *block ends and the* `Boolean` *variable* `done` *is not set to* `true`.

Sample Dialogue

```
Enter a whole number:
forty two
Not a correctly written whole number.
Try again.
Enter a whole number:
Fortytwo
Not a correctly written whole number.
Try again.
Enter a whole number:
42
You entered 42
```

# Exception Classes

- There are more exception classes than just the single class **`Exception`**
  - There are more exception classes in the standard Java libraries
  - New exception classes can be defined like any other class
- All predefined exception classes have the following properties:
  - There is a constructor that takes a single argument of type **`String`**
  - The class has an accessor method **`getMessage`** that can recover the string given as an argument to the constructor when the exception object was created
- All programmer-defined classes should have the same properties




# Exception Classes from Standard Packages

- Numerous predefined exception classes are included in the standard packages that come with Java
  - For example:

```
IOException
NoSuchMethodException
FileNotFoundException
```

  - Many exception classes must be imported in order to use them

```
import java.io.IOException;
```

## Exception Classes from Standard Packages

- The predefined exception class **`Exception`** is the root class for all exceptions
  - Every exception class is a descendent class of the class **`Exception`**
  - Although the **`Exception`** class can be used directly in a class or program, it is most often used to define a derived class
  - The class **`Exception`** is in the **`java.lang`** package, and so requires no **`import`** statement




## Using the `getMessage` Method

```
. . . // method code
try
{
  . . .
  throw new Exception(StringArgument);
  . . .
}
catch(Exception e)
{
  String message = e.getMessage();
  System.out.println(message);
  System.exit(0);
}  . . .
```

## Using the `getMessage` Method

- Every exception has a `String` instance variable that contains some message
  - This string typically identifies the reason for the exception
- In the previous example, `StringArgument` is an argument to the `Exception` constructor
- This is the string used for the value of the string instance variable of exception `e`
  - Therefore, the method call `e.getMessage()` returns this string




## Defining Exception Classes

- A `throw` statement can throw an exception object of any exception class
- Instead of using a predefined class, exception classes can be programmer-defined
  - These can be tailored to carry the precise kinds of information needed in the `catch` block
  - A different type of exception can be defined to identify each different exceptional situation

# Defining Exception Classes

- Every exception class to be defined must be a derived class of some already defined exception class
  - It can be a derived class of any exception class in the standard Java libraries, or of any programmer defined exception class
- Constructors are the most important members to define in an exception class
  - They must behave appropriately with respect to the variables and methods inherited from the base class
  - Often, there are no other members, except those inherited from the base class
- The following exception class performs these basic tasks only




# A Programmer-Defined Exception Class

Display 9.4 A Programmer-Defined Exception Class

```
1  public class DivisionByZeroException extends Exception
2  {
3      public DivisionByZeroException()
4      {
5          super("Division by Zero!");
6      }

7      public DivisionByZeroException(String message)
8      {
9          super(message);
10     }
11 }
```

*You can do more in an exception constructor, but this form is common.*

**super** *is an invocation of the constructor for the base class* **Exception**.

# Using our own Exception Class (1 of 3)

Display 9.5 Using a Programmer-Defined Exception Class

```
1  import java.util.Scanner;

2  public class DivisionDemoFirstVersion
3  {

4      public static void main(String[] args)
5      {
6          try
7          {
8              Scanner keyboard = new Scanner(System.in);

9              System.out.println("Enter numerator:");
10             int numerator = keyboard.nextInt();
11             System.out.println("Enter denominator:");
12             int denominator = keyboard.nextInt();
```




# Using our own Exception Class (2 of 3)

```
13             if (denominator == 0)
14                 throw new DivisionByZeroException();

15             double quotient = numerator/(double)denominator;
16             System.out.println(numerator + "/"
17                                + denominator
18                                + " = " + quotient);
19         }
20         catch (DivisionByZeroException e)
21         {
22             System.out.println(e.getMessage());
23             secondChance();
24         }

25         System.out.println("End of program.");
26     }
```

# Using our own Exception Class (3 of 3)

```
27      public static void secondChance()
28      {
29          Scanner keyboard = new Scanner(System.in);
30          System.out.println("Try again:");
31          System.out.println("Enter numerator:");
32          int numerator = keyboard.nextInt();
33          System.out.println("Enter denominator:");
34          System.out.println("Be sure the denominator is not zero.");
35          int denominator = keyboard.nextInt();
36                                                  Sometimes it is better to handle
37          if (denominator == 0)                   an exceptional case without throwing
38          {                                       an exception.
39              System.out.println("I cannot do division by zero.");
40              System.out.println("Aborting program.");
41              System.exit(0);
42          }

43          double quotient = ((double)numerator)/denominator;
44          System.out.println(numerator + "/"
45                                       + denominator
46                                       + " = " + quotient);
47      }
48  }
```





# Tip: An Exception Class Can Carry a Message of Any Type: int Message

- An exception class constructor can be defined that takes an argument of another type
  - It would stores its value in an instance variable
  - It would need to define accessor methods for this instance variable

## An Exception Class with an `int` Message

**An Exception Class with an `int` Message**

```
1  public class BadNumberException extends Exception
2  {
3      private int badNumber;

4      public BadNumberException(int number)
5      {
6          super("BadNumberException");
7          badNumber = number;
8      }

9      public BadNumberException()
10     {
11         super("BadNumberException");
12     }

13     public BadNumberException(String message)
14     {
15         super(message);
16     }

17     public int getBadNumber()
18     {
19         return badNumber;
20     }
21 }
```




## Exception Object Characteristics

- The two most important things about an exception object are its type (i.e., exception class) and the message it carries
  - The message is sent along with the exception object as an instance variable
  - This message can be recovered with the accessor method **`getMessage`**, so that the catch block can use the message

## Programmer-Defined Exception Class Guidelines

- Exception classes may be programmer-defined, but every such class must be a derived class of an already existing exception class
- The class **`Exception`** can be used as the base class, unless another exception class would be more suitable
- At least two constructors should be defined, sometimes more
- The exception class should allow for the fact that the method **`getMessage`** is inherited




## Preserve `getMessage`

- For all predefined exception classes, **`getMessage`** returns the string that is passed to its constructor as an argument
  - Or it will return a default string if no argument is used with the constructor
- This behavior must be preserved in all programmer-defined exception class
  - A constructor must be included having a string parameter whose body begins with a call to **`super`**
  - The call to **`super`** must use the parameter as its argument
  - A no-argument constructor must also be included whose body begins with a call to **`super`**
  - This call to **`super`** must use a default string as its argument

## Multiple `catch` Blocks

- A `try` block can potentially throw any number of exception values, and they can be of differing types
  - In any one execution of a `try` block, at most one exception can be thrown (since a throw statement ends the execution of the `try` block)
  - However, different types of exception values can be thrown on different executions of the `try` block




## Multiple `catch` Blocks

- Each `catch` block can only catch values of the exception class type given in the `catch` block heading
- Different types of exceptions can be caught by placing more than one `catch` block after a `try` block
  - Any number of `catch` blocks can be included, but they must be placed in the correct order

## Pitfall: Catch the More Specific Exception First

- When catching multiple exceptions, the order of the **`catch`** blocks is important
  - When an exception is thrown in a **`try`** block, the **`catch`** blocks are examined in order
  - The first one that matches the type of the exception thrown is the one that is executed




## Pitfall: Catch the More Specific Exception First

```
catch (Exception e)
{ . . . }
catch (NegativeNumberException e)
{ . . . }
```

- Because a **`NegativeNumberException`** is a type of **`Exception`**, all **`NegativeNumberExceptions`** will be caught by the first **`catch`** block before ever reaching the second block
  - The catch block for **`NegativeNumberException`** will never be used!
- For the correct ordering, simply reverse the two blocks

## Throwing an Exception in a Method

- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
  - Some programs that use a method should just end if an exception is thrown, and other programs should do something else
  - In such cases, the program using the method should enclose the method invocation in a **`try`** block, and catch the exception in a **`catch`** block that follows
- In this case, the method itself would not include **`try`** and **`catch`** blocks
  - However, it would have to include a ***`throws`*** *clause*




## Declaring Exceptions in a **`throws`** Clause

- If a method can throw an exception but does not catch it, it must provide a warning
  - This warning is called a ***`throws`*** *clause*
  - The process of including an exception class in a throws clause is called *declaring the exception*

```
throws AnException  //throws clause
```

  - The following states that an invocation of **`aMethod`** could throw **`AnException`**

```
public void aMethod() throws AnException
```

## Declaring Exceptions in a `throws` Clause

- If a method can throw more than one type of exception, then separate the exception types by commas

```
public void aMethod() throws
  AnException, AnotherException
```

- If a method throws an exception and does not catch it, then the method invocation ends immediately




## The Catch or Declare Rule

- Most ordinary exceptions that might be thrown within a method must be accounted for in one of two ways:
  1. The code that can throw an exception is placed within a `try` block, and the possible exception is caught in a `catch` block within the same method
  2. The possible exception can be declared at the start of the method definition by placing the exception class name in a `throws` clause

# The Catch or Declare Rule

- The first technique handles an exception in a `catch` block
- The second technique is a way to shift the exception handling responsibility to the method that invoked the exception throwing method
- The invoking method must handle the exception, unless it too uses the same technique to "pass the buck"
- Ultimately, every exception that is thrown should eventually be caught by a `catch` block in some method that does not just declare the exception class in a `throws` clause




# The Catch or Declare Rule

- In any one method, both techniques can be mixed
  - Some exceptions may be caught, and others may be declared in a `throws` clause
- However, these techniques must be used consistently with a given exception
  - If an exception is not declared, then it must be handled within the method
  - If an exception is declared, then the responsibility for handling it is shifted to some other calling method
  - Note that if a method definition encloses an invocation of a second method, and the second method can throw an exception and does not catch it, then the first method must catch or declare it

## Checked and Unchecked Exceptions

- Exceptions that are subject to the catch or declare rule are called *checked* exceptions
  - The compiler checks to see if they are accounted for with either a catch block or a throws clause
  - The classes **`Throwable`**, **`Exception`**, and all descendants of the class **`Exception`** are checked exceptions
- All other exceptions are *unchecked* exceptions
- The class **`Error`** and all its descendant classes are called *error classes*
  - Error classes are *not* subject to the Catch or Declare Rule

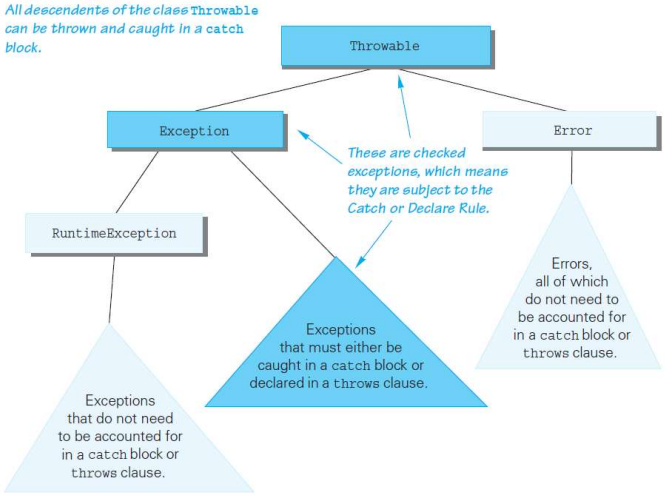



## Exceptions to the Catch or Declare Rule

- Checked exceptions must follow the Catch or Declare Rule
  - Programs in which these exceptions can be thrown will not compile until they are handled properly
- Unchecked exceptions are exempt from the Catch or Declare Rule
  - Programs in which these exceptions are thrown simply need to be corrected, as they result from some sort of error

# Hierarchy of Throwable Objects

Display 9.11 Hierarchy of `Throwable` Objects







## The **`throws`** Clause in Derived Classes

- When a method in a derived class is overridden, it should have the same exception classes listed in its **`throws`** clause that it had in the base class
  - Or it should have a subset of them
- A derived class may not add any exceptions to the **`throws`** clause
  - But it can delete some

## What Happens If an Exception is Never Caught?

- If every method up to and including the main method simply includes a **`throws`** clause for an exception, that exception may be thrown but never caught
  - In a GUI program (i.e., a program with a windowing interface), nothing happens - but the user may be left in an unexplained situation, and the program may be no longer be reliable
  - In non-GUI programs, this causes the program to terminate with an error message giving the name of the exception class
- Every well-written program should eventually catch every exception by a **`catch`** block in some method




## When to Use Exceptions

- Exceptions should be reserved for situations where a method encounters *an unusual or unexpected case that cannot be handled easily in some other way*
- When exception handling must be used, here are some basic guidelines:
  - Include **`throw`** statements and list the exception classes in a **`throws`** clause within a method definition
  - Place the **`try`** and **`catch`** blocks in a different method

# When to Use Exceptions

- Here is an example of a method from which the exception originates:

```
public void someMethod()
                    throws SomeException
{
   . . .
   throw new
           SomeException(SomeArgument);
   . . .
}
```




# When to Use Exceptions

- When `someMethod` is used by an `otherMethod`, the `otherMethod` must then deal with the exception:

```
public void otherMethod()
{
   try
   {
     someMethod();
     . . .
   }
   catch (SomeException e)
   {
     CodeToHandleException
   }
   . . .
}
```

# Event Driven Programming

- Exception handling is an example of a programming methodology known as *event-driven programming*
- When using event-driven programming, objects are defined so that they send events to other objects that handle the events
  - An event is an object also
  - Sending an event is called *firing an event*




# Event Driven Programming

- In exception handling, the event objects are the exception objects
  - They are fired (thrown) by an object when the object invokes a method that throws the exception
  - An exception event is sent to a `catch` block, where it is handled

# Pitfall: Nested `try-catch` Blocks

- It is possible to place a **`try`** block and its following catch blocks inside a larger **`try`** block, or inside a larger **`catch`** block
  - If a set of **`try-catch`** blocks are placed inside a larger **`catch`** block, different names must be used for the **`catch`** block parameters in the inner and outer blocks, just like any other set of nested blocks
  - If a set of **`try-catch`** blocks are placed inside a larger **`try`** block, and an exception is thrown in the inner **`try`** block that is not caught, then the exception is thrown to the outer **`try`** block for processing, and may be caught in one of its **`catch`** blocks




# The `finally` Block

- The **`finally`** block contains code to be executed whether or not an exception is thrown in a **`try`** block
  - If it is used, a **`finally`** block is placed after a **`try`** block and its following **`catch`** blocks

```
try
{  . . .  }
catch(ExceptionClass1 e)
{  . . .  }
  . . .
catch(ExceptionClassN e)
{  . . .  }
finally
{
  CodeToBeExecutedInAllCases
}
```

# The `finally` Block

- If the `try-catch-finally` blocks are inside a method definition, there are three possibilities when the code is run:
  1. The `try` block runs to the end, no exception is thrown, and the finally block is executed
  2. An exception is thrown in the `try` block, caught in one of the `catch` blocks, and the `finally` block is executed
  3. An exception is thrown in the `try` block, there is no matching `catch` block in the method, the `finally` block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method




# Rethrowing an Exception

- A `catch` block can contain code that throws an exception
  - Sometimes it is useful to catch an exception and then, depending on the string produced by `getMessage` (or perhaps something else), throw the same or a different exception for handling further up the chain of exception handling blocks

# The `AssertionError` Class

- When a program contains an assertion check, and the assertion check fails, an object of the class **`AssertionError`** is thrown
  - This causes the program to end with an error message
- The class **`AssertionError`** is derived from the class **`Error`**, and therefore is an unchecked exception
  - In order to prevent the program from ending, it could be handled, but this is not required




# `ArrayIndexOutOfBoundsException`

- An **`ArrayIndexOutOfBoundsException`** is thrown whenever a program attempts to use an array index that is out of bounds
  - This normally causes the program to end
- Like all other descendents of the class **`RuntimeException`**, it is an unchecked exception
  - There is no requirement to handle it
- When this exception is thrown, it is an indication that the program contains an error
  - Instead of attempting to handle the exception, the program should simply be fixed