Chapter 10

EXCEPTION HANDLING

Exception

An exception is an abnormal condition that arises in a code sequence at run time, it may cause system to crash and affect other programs running in parallel.

How is an Exception represented in Java?

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error
- That method may choose to

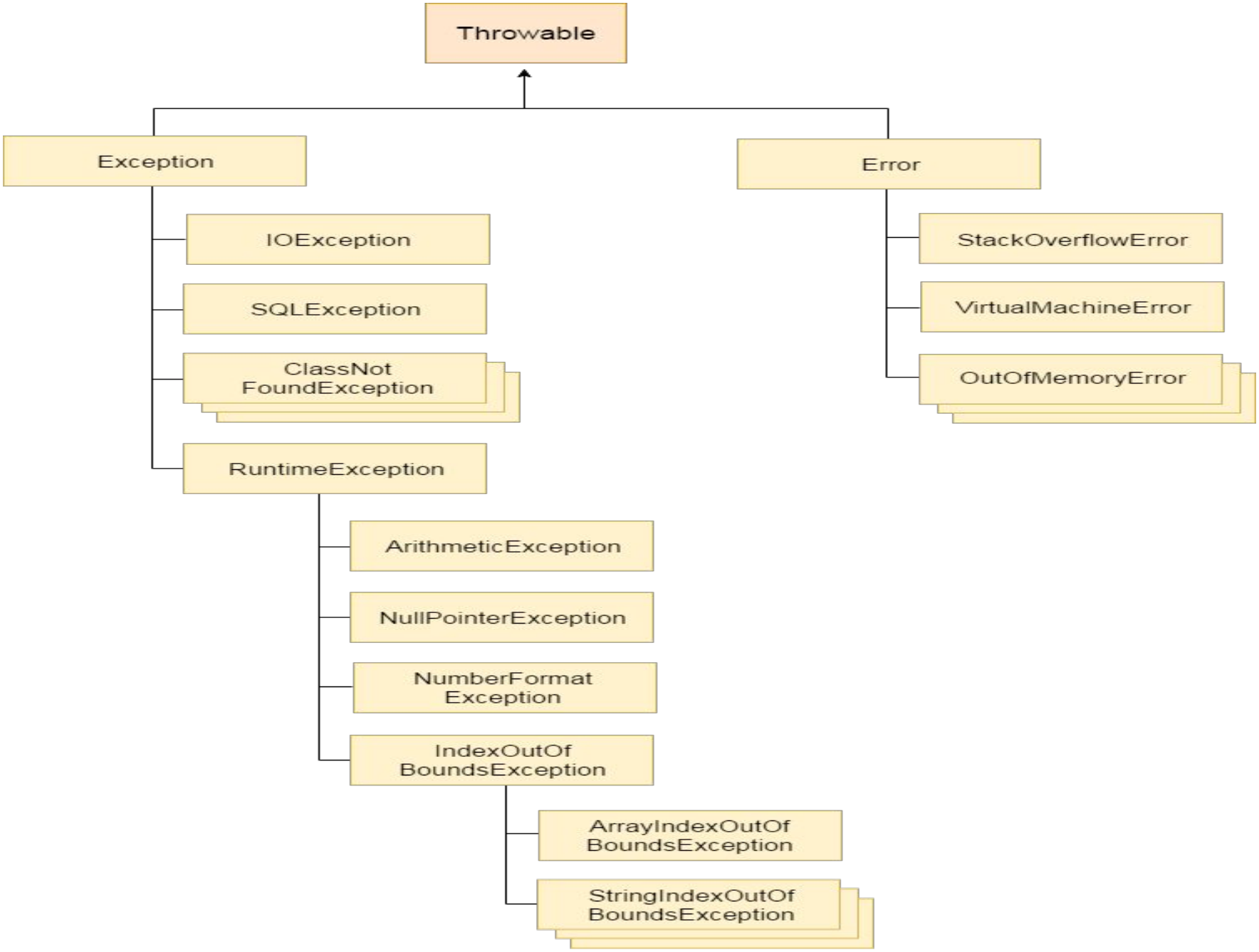
  a) handle it self

  or

  b) pass it on.
- At some point the exception is *caught* and processed

The Exception Hierarchy

- In Java, all exceptions are represented by classes. All exception classes are derived from a class called **Throwable**. Thus, when an exception occurs in a program, an object of some type of exception class is generated.
- There are two direct subclasses of **Throwable**: **Exception** and **Error**.
- Exceptions of type **Error:**
  - Errors that occur in the Java virtual machine itself, and not in the program.
  - Beyond the user control, and the program will not usually deal with them.
- Errors that result from program activity are represented by subclasses of **Exception**.
- For example, divide-by-zero, array boundary, and file errors fall into this category. The program should handle exceptions of these types. An important subclass of **Exception** is **RuntimeException**, which is used to represent various common types of run-time errors.


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

Exception Handling Fundamentals

- Java exception handling is managed via five keywords:
- **try**, **catch**, **throw**, **throws**, and **finally**

Using try and catch

At the core of exception handling are **try** and **catch**. These keywords work together; you have a **catch** without a **try**. Here is the general form of the **try/catch** exception handling

```
try {
  // block of code to monitor for errors
}
catch (ExcepType1 exOb) {
  // handler for ExcepType1
}
catch (ExcepType2 exOb) {
  // handler for ExcepType2
}
```

.

Here, *ExcepType* is the type of exception that has occurred. When an exception is thrown, it is caught by its corresponding **catch** statement, which then processes the exception. As the general form shows, there can be more than one **catch** statement associated with a **try**. The type of the exception determines which **catch** statement is executed. That is, if the exception type specified by a **catch** statement matches that of the exception, then that **catch** statement is executed (and all others are bypassed). When an exception is caught, *exOb* will receive its value.

Here is an important point: If no exception is thrown, then a **try** block ends normally, and all of its **catch** statements are bypassed. Execution resumes with the first statement following the last **catch**. Thus, **catch** statements are executed only if an exception is thrown.

NOTE

Beginning with JDK 7, there is another form of the **try** statement that supports *automatic resource management*. This form of **try** is called ***try*-with-resources**. It is described in Chapter 10, in the context of managing I/O streams (such as those connected to a file) because streams are some of the most commonly used resources.

A Simple Exception Example
Using
try and catch blocks

```
class ExcDemo1 {
  public static void main(String args[]) {
    int nums[] = new int[4];

  try {
    System.out.println("Before exception is generated.");

    // Generate an index out-of-bounds exception.
    nums[7] = 10;
    System.out.println("this won't be displayed");
  }
  catch (ArrayIndexOutOfBoundsException exc) {
    // catch the exception
    System.out.println("Index out-of-bounds!");
  }
  System.out.println("After catch statement.");
 }
}
```

Create a try block.

Attempt to index past nums boundary.

Catch array boundary errors.

This program displays the following output:

```
Before exception is generated.
Index out-of-bounds!
After catch statement.
```

bounds index will never execute. After the **catch** statement executes, program control continues with the statements following the **catch**. Thus, it is the job of your exception handler to remedy the problem that caused the exception so that program execution can continue normally.

Remember, if no exception is thrown by a **try** block, no **catch** statements will be executed and program control resumes after the **catch** statement. To confirm this, in the preceding program, change the line

```
nums[7] = 10;
```

to

```
nums[0] = 10;
```

Now, no exception is generated, and the **catch** block is not executed.

```
/* An exception can be generated by one
   method and caught by another. */

class ExcTest {
  // Generate an exception.
  static void genException() {
    int nums[] = new int[4];

    System.out.println("Before exception is generated.");

    // generate an index out-of-bounds exception
    nums[7] = 10;
    System.out.println("this won't be displayed");
  }
}
```

Exception generated here.

```
class ExcDemo2 {
  public static void main(String args[]) {

    try {
      ExcTest.genException();
    } catch (ArrayIndexOutOfBoundsException exc) {
      // catch the exception
      System.out.println("Index out-of-bounds!");
    }
    System.out.println("After catch statement.");
  }
}
```

Exception caught here.

```
Before exception is generated.
Index out-of-bounds!
After catch statement.
```

The Consequences of an Uncaught Exception

```
// Let JVM handle the error.
class NotHandled {
  public static void main(String args[]) {
    int nums[] = new int[4];

    System.out.println("Before exception is generated.");

    // generate an index out-of-bounds exception
    nums[7] = 10;
  }
}
```

If your program does not catch an exception, then it will be caught by the JVM. The JVM's default exception handler terminates execution and displays a stack trace and error message

When the array index error occurs, execution is halted, and the following error message is displayed.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
        at NotHandled.main(NotHandled.java:9)
```

```
// This won't work!
class ExcTypeMismatch {
  public static void main(String args[]) {
    int nums[] = new int[4];

    try {
      System.out.println("Before exception is generated.");

      //generate an index out-of-bounds exception
      nums[7] = 10;
      System.out.println("this won't be displayed");
    }

    /* Can't catch an array boundary error with an
       ArithmeticException. */
```

This throws an ArrayIndexOutOfBoundsException.

```
    catch (ArithmeticException exc) {
      // catch the exception
      System.out.println("Index out-of-bounds!");
    }
    System.out.println("After catch statement.");
  }
}
```

This tries to catch it with an **ArithmeticException**.

The output is shown here.

```
Before exception is generated.
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
        at ExcTypeMismatch.main(ExcTypeMismatch.java:10)
```

As the output demonstrates, a **catch** for **ArithmeticException** won't catch an **ArrayIndexOutOfBoundsException**.

Exceptions Enable You to Handle Errors Gracefully

```
// Handle error gracefully and continue.
class ExcDemo3 {
  public static void main(String args[]) {
    int numer[] = { 4, 8, 16, 32, 64, 128 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.length; i++) {
      try {
        System.out.println(numer[i] + " / " +
                           denom[i] + " is " +
                           numer[i]/denom[i]);
      }
      catch (ArithmeticException exc) {
        // catch the exception
        System.out.println("Can't divide by Zero!");
      }
    }
  }
}
```

The output from the program is shown here:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
```

Using Multiple catch Statements

```
// Use multiple catch statements.
class ExcDemo4 {
  public static void main(String args[]) {
    // Here, numer is longer than denom.
    int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.length; i++) {
      try {
        System.out.println(numer[i] + " / " +
                           denom[i] + " is " +
                           numer[i]/denom[i]);
      }
      catch (ArithmeticException exc) {
        // catch the exception
        System.out.println("Can't divide by Zero!");
      }
```

Multiple **catch** statements

```
        catch (ArrayIndexOutOfBoundsException exc) {
          // catch the exception
          System.out.println("No matching element found.");
        }
      }
    }
}
```

This program produces the following output:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
No matching element found.
```

Catching Subclass Exceptions

- If you want to catch exceptions of both a superclass type and a subclass type, put the subclass first in the **catch** sequence.

- Putting the superclass first causes unreachable code to be created, since the subclass **catch** clause can never execute

```
class ExcDemo5 {
  public static void main(String args[]) {
    // Here, numer is longer than denom.
    int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.length; i++) {
      try {
        System.out.println(numer[i] + " / " +
                           denom[i] + " is " +
                           numer[i]/denom[i]);
      }
      catch (ArrayIndexOutOfBoundsException exc) {
        // catch the exception
        System.out.println("No matching element found.");
      }
```

← Catch subclass

```
      catch (Throwable exc) {
        System.out.println("Some exception occurred.");
      }
    }
  }
}
```

Catch superclass

The output from the program is shown here:

```
4 / 2 is 2
Some exception occurred.
16 / 4 is 4
32 / 4 is 8
Some exception occurred.
128 / 8 is 16
No matching element found.
No matching element found.
```

Try Blocks Can Be Nested

```
// Use a nested try block.
class NestTrys {
  public static void main(String args[]) {
    // Here, numer is longer than denom.
    int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };

    try { // outer try
      for(int i=0; i<numer.length; i++) {
        try { // nested try
          System.out.println(numer[i] + " / " +
                             denom[i] + " is " +
                             numer[i]/denom[i]);
        }
        catch (ArithmeticException exc) {
          // catch the exception
          System.out.println("Can't divide by Zero!");
        }
      }
    }
```

Nested try blocks

```
    catch (ArrayIndexOutOfBoundsException exc) {
      // catch the exception
      System.out.println("No matching element found.");
      System.out.println("Fatal error - program terminated.")
    }
  }
}
```

The output from the program is shown here:

```
4 / 2 is 2
Can't divide by Zero!
16 / 4 is 4
32 / 4 is 8
Can't divide by Zero!
128 / 8 is 16
No matching element found.
Fatal error - program terminated.
```

In this example, an exception that can be handled by the inner **try**—in this case, a divide-by-zero error—allows the program to continue. However, an array boundary error is caught by the outer **try**, which causes the program to terminate.

Throwing an Exception

- It is possible to manually throw an exception by using the **throw** statement. Its general form is shown here:
- throw *exceptOb*;
- Here, *exceptOb* must be an object of an exception class derived from **Throwable**.
- Here is an example that illustrates the **throw** statement by manually throwing an **ArithmeticException**:

throw Demo

```
// Manually throw an exception.
class ThrowDemo {
  public static void main(String args[]) {
    try {
      System.out.println("Before throw.");
      throw new ArithmeticException();
    }
    catch (ArithmeticException exc) {
      // catch the exception
      System.out.println("Exception caught.");
    }
    System.out.println("After try/catch block.");
  }
}
```

Throw an exception.

The output from the program is shown here:

```
Before throw.
Exception caught.
After try/catch block.
```

Rethrowing an Exception

An exception caught by one **catch** statement can be rethrown so that it can be caught by an outer **catch**. The most likely reason for rethrowing this way is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception, and a second handler copes with another aspect. Remember, when you rethrow an exception, it will not be recaught by the same **catch** statement. It will propagate to the next **catch** statement. The following program illustrates rethrowing an exception:

```
// Rethrow an exception.
class Rethrow {
  public static void genException() {
    // here, numer is longer than denom
    int numer[] = { 4, 8, 16, 32, 64, 128, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };
```

```
for(int i=0; i<numer.length; i++) {
  try {
    System.out.println(numer[i] + " / " +
                       denom[i] + " is " +
                       numer[i]/denom[i]);
  }
  catch (ArithmeticException exc) {
    // catch the exception
    System.out.println("Can't divide by Zero!");
  }
  catch (ArrayIndexOutOfBoundsException exc) {
    // catch the exception
    System.out.println("No matching element found.");
    throw exc; // rethrow the exception
  }
}
```

Rethrow the exception.

```
  }
}

class RethrowDemo {
  public static void main(String args[]) {
    try {
      Rethrow.genException();
    }
    catch(ArrayIndexOutOfBoundsException exc) {  <-- Catch rethrown exception.
      // recatch exception
      System.out.println("Fatal error - " +
                         "program terminated.");
    }
  }
}
```

In this program, divide-by-zero errors are handled locally, by **genException( )**, but an array boundary error is rethrown. In this case, it is caught by **main( )**.

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
String toString( )	Returns a **String** object containing a complete description of the exception. This method is called by **println**( ) when outputting a **Throwable** object.

Table 9-1 Commonly Used Methods Defined by **Throwable**

Of the methods defined by **Throwable**, two of the most interesting are **printStackTrace( )** and **toString( )**. You can display the standard error message plus a record of the method calls that lead up to the exception by calling **printStackTrace( )**. You can use **toString( )** to retrieve the standard error message. The **toString( )** method is also called when an exception is used as an argument to **println( )**. The following program demonstrates these methods:

```
// Using the Throwable methods.

class ExcTest {
  static void genException() {
    int nums[] = new int[4];

    System.out.println("Before exception is generated.");

    // generate an index out-of-bounds exception
```

```
    // generate an index out-of-bounds exception
    nums[7] = 10;
    System.out.println("this won't be displayed");
  }
}

class UseThrowableMethods {
  public static void main(String args[]) {

    try {
      ExcTest.genException();
    }
    catch (ArrayIndexOutOfBoundsException exc) {
      // catch the exception
      System.out.println("Standard message is: ");
      System.out.println(exc);
      System.out.println("\nStack trace: ");
```

```
      exc.printStackTrace();
    }
    System.out.println("After catch statement.");
  }
}
```

The output from this program is shown here:

```
Before exception is generated.
Standard message is:
java.lang.ArrayIndexOutOfBoundsException: 7

Stack trace:
java.lang.ArrayIndexOutOfBoundsException: 7
    at ExcTest.genException(UseThrowableMethods.java:10)
    at UseThrowableMethods.main(UseThrowableMethods.java:19)
After catch statement.
```

Using finally

Sometimes you will want to define a block of code that will execute when a **try/catch** block is left. For example, an exception might cause an error that terminates the current method, causing its premature return. However, that method may have opened a file or a network connection that needs to be closed. Such types of circumstances are common in programming, and Java provides a convenient way to handle them: **finally**.

To specify a block of code to execute when a **try/catch** block is exited, include a **finally** block at the end of a **try/catch** sequence. The general form of a **try/catch** that includes **finally** is shown here.

```
try {
  // block of code to monitor for errors
}
catch (ExcepType1 exOb) {
  // handler for ExcepType1
}
catch (ExcepType2 exOb) {
  // handler for ExcepType2
}
//...
finally {
  // finally code
}
```

The **finally** block will be executed whenever execution leaves a **try/catch** block, no matter what conditions cause it. That is, whether the **try** block ends normally, or because of an exception, the last code executed is that defined by **finally**. The **finally** block is also executed if any code within the **try** block or any of its **catch** statements return from the method.

Here is an example of **finally**:

```
// Use finally.
class UseFinally {
  public static void genException(int what) {
    int t;
    int nums[] = new int[2];

    System.out.println("Receiving " + what);
```

```
try {
  switch(what) {
    case 0:
      t = 10 / what; // generate div-by-zero error
      break;
    case 1:
      nums[4] = 4; // generate array index error.
      break;
    case 2:
      return; // return from try block
  }
}
catch (ArithmeticException exc) {
  // catch the exception
  System.out.println("Can't divide by Zero!");
  return; // return from catch
}
```

```
    catch (ArrayIndexOutOfBoundsException exc) {
      // catch the exception
      System.out.println("No matching element found.");
    }
    finally {
      System.out.println("Leaving try.");
    }
  }
}

class FinallyDemo {
  public static void main(String args[]) {
      for(int i=0; i < 3; i++) {
        UseFinally.genException(i);
        System.out.println();
      }
    }
```

This is executed on the way out of **try/catch** blocks.

Here is the output produced by the program:

```
Receiving 0
Can't divide by Zero!
Leaving try.

Receiving 1
No matching element found.
Leaving try.

Receiving 2
Leaving try.
```

As the output shows, no matter how the **try** block is exited, the **finally** block is executed.

Using throws

- In some cases, if a method generates an exception that it does not handle, it must declare that exception in a **throws** clause. Here is the general form of a method that includes a **throws** clause:
- *ret-type methName*(*param-list*) throws *except-list* { // body }

- Here, *except-list* is a comma-separated list of exceptions that the method might throw outside of itself.

- Exceptions that are subclasses of **Error** or **RuntimeException** don't need to be specified in a **throws** list. Java simply assumes that a method may throw one. All other types of exceptions *do* need to be declared. Failure to do so causes a compile-time error.

```
// Use throws.
class ThrowsDemo {
  public static char prompt(String str)
    throws java.io.IOException {

    System.out.print(str + ": ");
    return (char) System.in.read();
  }

  public static void main(String args[]) {
    char ch;

    try {
      ch = prompt("Enter a letter");
    }
    catch(java.io.IOException exc) {
```

Notice the **throws** clause.

Since **prompt( )** might throw an exception, a call to it must be enclosed within a **try** block.

```
    catch(java.io.IOException exc) {
      System.out.println("I/O exception occurred.");
      ch = 'X';
    }

    System.out.println("You pressed " + ch);
  }
}
```

On a related point, notice that **IOException** is fully qualified by its package name **java.io**. As you will learn in Chapter 10, Java's I/O system is contained in the **java.io** package. Thus, the **IOException** is also contained there. It would also have been possible to import **java.io** and then refer to **IOException** directly.

Three Recently Added Exception Features

- The first supports *automatic resource management,* which automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of **try**, called the ***try****-with-resources* statement.
- The second is multi-catch allows two or more exceptions to be caught by the same **catch** clause.
- The third is

  final rethrow or *more precise rethrow*

Multi catch

```
// Use the multi-catch feature.  Note: This code requires JDK 7 or
// later to compile.
class MultiCatch {
  public static void main(String args[]) {
    int a=88, b=0;
    int result;
    char chrs[] = { 'A', 'B', 'C' };

    for(int i=0; i < 2; i++) {
      try {
        if(i == 0)
          result = a / b; // generate an ArithmeticException
        else
          chrs[5] = 'X'; // generate an ArrayIndexOutOfBoundsException

      // This catch clause catches both exceptions.
      }
      catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception caught: " + e);
      }
    }

    System.out.println("After multi-catch.");
  }
}
```

More precise rethrow

- The more precise rethrow feature restricts the type of exceptions that can be rethrown to only those checked exceptions that the associated **try** block throws, that are not handled by a preceding **catch** clause, and that are a subtype or supertype of the parameter.

Java's Built-in Exceptions

Unchecked exceptions:

- The compiler does not check to see if a method handles or throws these exceptions.
- They need not be included in any method's **throws** list

Checked exceptions:

- Must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself.

Exception	Meaning
ArithmeticException	Arithmetic error, such as integer divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 9-2 The Unchecked Exceptions Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 9-3 The Checked Exceptions Defined in **java.lang**

Creating Exception Subclasses

```
// Use a custom exception.

// Create an exception.
class NonIntResultException extends Exception {
  int n;
  int d;

  NonIntResultException(int i, int j) {
    n = i;
    d = j;
  }

  public String toString() {
    return "Result of " + n + " / " + d +
           " is non-integer.";
  }
}
```

```
class CustomExceptDemo {
  public static void main(String args[]) {

    // Here, numer contains some odd values.
    int numer[] = { 4, 8, 15, 32, 64, 127, 256, 512 };
    int denom[] = { 2, 0, 4, 4, 0, 8 };

    for(int i=0; i<numer.length; i++) {
      try {
        if((numer[i]%2) != 0)
          throw new
            NonIntResultException(numer[i], denom[i]);

        System.out.println(numer[i] + " / " +
                           denom[i] + " is " +
                           numer[i]/denom[i]);
      }
```

```
      catch (ArithmeticException exc) {
        // catch the exception
        System.out.println("Can't divide by Zero!");
      }
      catch (ArrayIndexOutOfBoundsException exc) {
        // catch the exception
        System.out.println("No matching element found.");
      }
      catch (NonIntResultException exc) {
        System.out.println(exc);
      }
    }
  }
}
```

The output from the program is shown here:

```
4 / 2 is 2
Can't divide by Zero!
Result of 15 / 4 is non-integer.
32 / 4 is 8
Can't divide by Zero!
Result of 127 / 8 is non-integer.
No matching element found.
No matching element found.
```

Questions

- What class is at the top of the exception hierarchy?
- What are checked and unchecked exceptions?
- Differentiate between throw and throws keywords.
- What is the significance of finally keyword? Explain.