



7: Exception Handling

IT1406 - Introduction to Programming

Level I - Semester 1

7. Exception Handling

Content

- 7.1. Exception handling fundamentals
- 7.2. Exception types
- 7.3. Uncaught exceptions
- 7.4. Using exceptions
 - Try
 - Catch
 - Throw
 - Throws
 - Finally
 - Java's built in Exceptions
 - Creating one's own exception sub classes

7.1. Exception handling fundamentals

7.1. Exception handling fundamentals

- 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 handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

7.1. Exception handling fundamentals (contd.)

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

7.1. Exception handling fundamentals (contd.)

- This is the general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors  
}  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1  
}  
catch (ExceptionType2 exOb) {  
    // exception handler for ExceptionType2  
}  
// ...  
finally {  
    // block of code to be executed after try block ends  
}
```

- Here, *ExceptionType* is the type of exception that has occurred.

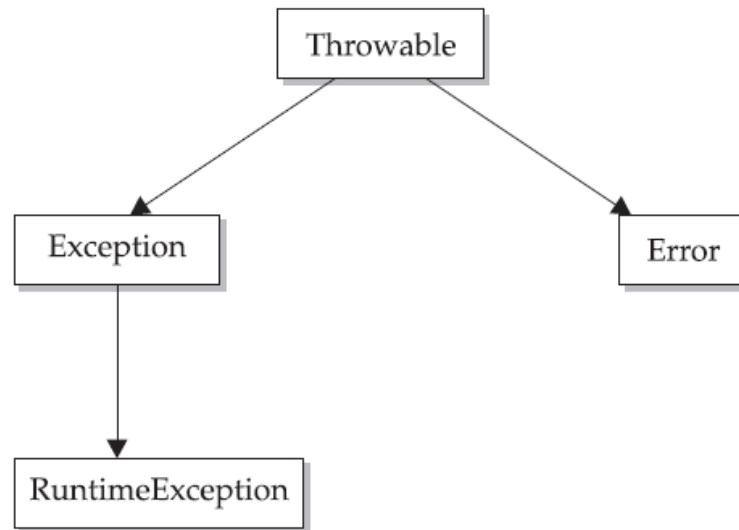
7.2. Exception types

7.2. Exception types

- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.
- There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

7.2. Exception types (contd.)

- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.
- The top-level exception hierarchy is shown here:



7.3. Uncaught exceptions

7.3. Uncaught exceptions

- This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;  
    }  
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.

7.3. Uncaught exceptions (contd.)

- This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.
- Any exception that I
- s not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
- Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero  
at Exc0.main(Exc0.java:4)
```

7.3. Uncaught exceptions (contd.)

- Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened.
- The preceding program that introduces the same error but in a method separate from **main()**:

```
class Exc1 {  
    static void subroutine() {  
        int d = 0;  
        int a = 10 / d;  
    }  
    public static void main(String args[]) {  
        Exc1.subroutine();  
    }  
}
```

7.3. Uncaught exceptions (contd.)

- The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero  
at Exc1.subroutine(Exc1.java:4)  
at Exc1.main(Exc1.java:7)
```

- Here, the bottom of the stack is main's line 7, which is the call to subroutine(), which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

7.4. Using exceptions

Try and catch

- Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits.
 - First, it allows you to fix the error.
 - Second, it prevents the program from automatically terminating.
- Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.
- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause that processes the **ArithmeticException** generated by the division-by-zero error:

Try and catch (contd.)

```
class Exc2 {  
    public static void main(String args[]) {  
        int d, a;  
        try { // monitor a block of code.  
            d = 0;  
            a = 42 / d;  
            System.out.println("This will not be printed.");  
        } catch (ArithmeticException e) { // catch divide-by-zero error  
            System.out.println("Division by zero.");  
        }  
        System.out.println("After catch statement.");  
    }  
}
```

- This program generates the following output:

Division by zero.

After catch statement.

Try and catch (contd.)

- Notice that the call to **println()** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**.
- A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements).
- The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement.

Displaying a Description of an Exception

- **Throwable** overrides the **toString()** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println()** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {  
    System.out.println("Exception: " + e);  
    a = 0; // set a to zero and continue  
}
```

- When this version is substituted in the program, and the program is run, each divide-byzero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

Multiple catch Clauses

- In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try / catch** block.
- The following example traps two different exception types:

Multiple catch Clauses (contd.)

```
// Demonstrate multiple catch statements.
class MultipleCatches {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;
        } catch(ArithmeticException e) {
            System.out.println("Divide by 0: " + e);
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
    }
}
```

Multiple catch Clauses (contd.0

- This program will cause a division-by-zero exception if it is started with no commandline arguments, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero.
- But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**. program attempts to assign a value to **c[42]**.
- Here is the output generated by running it both ways:

```
C:\>java MultipleCatches
```

```
a = 0
```

```
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
After try/catch blocks.
```

```
C:\>java MultipleCatches TestArg
```

```
a = 1
```

```
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
```

```
After try/catch blocks.
```

Multiple catch Clauses (contd.)

- When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses.
- Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error
- For example, consider the following program.

Multiple catch Clauses (contd.)

/* This program contains an error. A subclass must come before its superclass in a series of catch statements. If not, unreachable code will be created and a compile-time error will result.*/

```
class SuperSubCatch {  
    public static void main(String args[]) {  
        try {  
            int a = 0;  
            int b = 42 / a;  
        } catch(Exception e) {  
            System.out.println("Generic Exception catch.");  
        }  
        //This catch is never reached because ArithmeticException is a subclass of  
        //Exception.  
        catch(ArithmeticException e) { // ERROR – unreachable  
            System.out.println("This is never reached.");  
        }  
    }  
}
```


Multiple catch Clauses (contd.)

- If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught.
- Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute.
- To fix the problem, reverse the order of the **catch** statements.

Nested try Statements

- The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.
- Here is an example that uses nested **try** statements:

Nested try Statements (contd.)

```
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* If no command-line args are present, the following statement
            will generate a divide-by-zero exception. */
            int b = 42 / a;

            System.out.println("a = " + a);
            try { // nested try block
                /* If one command-line arg is used, then a divide-by-zero
                exception will be generated by the following code. */
                if(a==1) a = a/(a-a); // division by zero

                /* If two command-line args are used, then generate an
                out-of-bounds exception. */
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            }
        }
    }
}
```

Nested try Statements (contd.)

```
        } catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println("Array index out-of-bounds: " + e);  
        }  
    } catch(ArithmeticException e) {  
        System.out.println("Divide by 0: " + e);  
    }  
}  
}
```

- When you execute the program with no command-line arguments, a divide-by zero exception is generated by the outer **try** block.
- Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled.

Nested try Statements (contd.)

- If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner **try** block. Here are sample runs that illustrate each case:

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

Multiple-catch features

- Beginning with JDK 7, three interesting and useful features have been added to the exception system.
 - The first 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 the **try** statement called ***try-with-resources***.
 - The second feature is called *multi-catch*,
 - And the third is sometimes referred to as *final rethrow* or *more precise rethrow*.
- The multi-catch feature allows two or more exceptions to be caught by the same catch clause.

Multiple-catch features (contd.)

- It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions.
- Instead of having to catch each exception type individually, you can use a single **catch** clause to handle all of the exceptions without code duplication.
- To use a multi-catch, separate each exception type in the **catch** clause with the OR operator. Each multi-catch parameter is implicitly **final**. (You can explicitly specify **final**, if desired, but it is not necessary.) Because each multi-catch parameter is implicitly **final**, it can't be assigned a new value.
- Here is a **catch** statement that uses the multi-catch feature to catch both **ArithmeticException** and **ArrayIndexOutOfBoundsException**:

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException  
e) {
```

Multiple-catch features (contd.)

- The following program shows the multi-catch feature in action:

```
// Demonstrate the multi-catch feature.
class MultiCatch {
    public static void main(String args[]) {
        int a=10, b=0;
        int vals[] = { 1, 2, 3 };

        try {
            int result = a / b; // generate an ArithmeticException
            // vals[10] = 19; // 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.");
    }
}
```


Multiple-catch features (contd.)

- The program will generate an **ArithmeticException** when the division by zero is attempted.
- If you comment out the division statement and remove the comment symbol from the next line, an **ArrayIndexOutOfBoundsException** is generated.
- Both exceptions are caught by the single **catch** statement.

Throw

- It is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

throw ThrowableInstance;

- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- There are two ways you can obtain a **Throwable** object: using a parameter in a **catch** clause or creating one with the **new** operator.

Throw (contd.)

- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
- The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.
- Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

Throw (contd.)

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }
    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

Throw (contd.)

- This program gets two chances to deal with the same error. First, **main()** sets up an exception context and then calls **demoproc()**.
- The **demoproc()** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown.
- Here is the resulting output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

Throw (contd.)

- The program also illustrates how to create one of Java's standard exception objects. Consider about this line:

```
throw new NullPointerException("demo");
```

- Here, **new** is used to construct an instance of **NullPointerException**.
- Many of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception.
- This string is displayed when the object is used as an argument to **print()** or **println()**. It can also be obtained by a call to **getMessage()**, which is defined by **Throwable**.

Throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration.
- A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

Throws (contd.)

- This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws  
exception-list  
{  
    // body of method  
}
```

- Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Throws (contd.)

- Example:

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws
    IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Throws (contd.)

- Here is the output generated by running this example program:

inside throwOne

caught java.lang.IllegalAccessException: demo

finally

- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Finally (contd.)

- Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
    // Throw an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("procA's finally");
        }
    }
    // Return from within a try block.
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        } finally {
            System.out.println("procB's finally");
        }
    }
}
```

Finally (contd.)

```
    }  
}  
// Execute a try block normally.  
static void procC() {  
    try {  
        System.out.println("inside procC");  
    } finally {  
        System.out.println("procC's finally");  
    }  
}  
}  
public static void main(String args[]) {  
    try {  
        procA();  
    } catch (Exception e) {  
        System.out.println("Exception caught");  
    }  
    procB();  
    procC();  
}  
}
```

Finally (contd.)

- In this example, **procA()** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB()**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB()** returns. In **procC()**, the **try** statement executes normally, without error. However, the **finally** block is still executed.
- **REMEMBER** If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.
- Here is the output generated by the preceding program:

```
inside procA  
procA's finally  
Exception caught  
inside procB  
procB's finally  
inside procC  
procC's finally
```

Java's Built-in Exceptions

- Inside the standard package **java.lang**, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**.
- These exceptions need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.
- The unchecked exceptions defined in **java.lang** are listed in the following table.

Exception	Meaning
ArithmeticException	Arithmetic error, such as 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.

Java's Built-in Exceptions (contd.)

- Next table lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself.
- These are called *checked exceptions*. In addition to the exceptions in **java.lang**, Java defines several more that relate to its other standard packages.

Java's Built-in Exceptions (contd.)

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.

Creating one's own exception sub classes

- To create own exception: Just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). The subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. They are shown in following Table. You may also wish to override one or more of these methods in exception classes that you create.

Method	Description
final void addSuppressed(Throwable <i>exc</i>)	Adds <i>exc</i> to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the try -with-resources statement.
Throwable fillInStackTrace()	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause()	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
StackTraceElement[] getStackTrace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement . The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.

<code>final Throwable[] getSuppressed()</code>	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the try-with-resources statement.
<code>Throwable initCause(Throwable <i>causeExc</i>)</code>	Associates <i>causeExc</i> with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
<code>void printStackTrace()</code>	Displays the stack trace.
<code>void printStackTrace(PrintStream <i>stream</i>)</code>	Sends the stack trace to the specified stream.
<code>void printStackTrace(PrintWriter <i>stream</i>)</code>	Sends the stack trace to the specified stream.
<code>void setStackTrace(StackTraceElement <i>elements</i>[])</code>	Sets the stack trace to the elements passed in <i>elements</i> . This method is for specialized applications, not normal use.
<code>String toString()</code>	Returns a String object containing a description of the exception. This method is called by println() when outputting a Throwable object.

- **Exception** defines four public constructors. Two support chained exceptions, described in the next section. The other two are shown here:

`Exception()`

`Exception(String msg)`

- The first form creates an exception that has no description. The second form lets you specify a description of the exception.
- Although specifying a description when an exception is created is often useful, sometimes it is better to override **toString()**. Here's why: The version of **toString()** defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString()**, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

Chained Exceptions

- Beginning with JDK 1.4, a feature was incorporated into the exception subsystem: *chained exceptions*.
- The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception.
- For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an **ArithmeticException**, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

Chained Exceptions (contd.)

- To allow chained exceptions, two constructors and two methods were added to **Throwable**. The constructors are shown here:

`Throwable(Throwable causeExc)`

`Throwable(String msg, Throwable causeExc)`

- In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception.
- These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

Chained Exceptions (contd.)

- The chained exception methods supported by `Throwable` are `getCause()` and `initCause()`. These methods are shown in Table 10-3 and are repeated here for the sake of discussion.

`Throwable getCause()`

`Throwable initCause(Throwable causeExc)`

- The **`getCause()`** method returns the exception that underlies the current exception. If there is no underlying exception, **`null`** is returned. The **`initCause()`** method associates *causeExc* with the invoking exception and returns a reference to the exception.
- Thus, you can associate a cause with an exception after the exception has been created.

Chained Exceptions (contd.)

- However, the cause exception can be set only once. Thus, you can call **initCause()** only once for each exception object.
- Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause()**. In general, **initCause()** is used to set a cause for legacy exception classes that don't support the two additional constructors described earlier.
- In the following example, the top-level exception is **NullPointerException**. To it is added a cause exception, **ArithmeticException**. When the exception is thrown out of **demoproc()**, it is caught by **main()**. There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling **getCause()**.
- Here is the example that illustrates the mechanics of handling chained exceptions:

```
// Demonstrate exception chaining.
class ChainExcDemo {
    static void demoproc() {

        // create an exception
        NullPointerException e = new NullPointerException("top
layer");

        // add a cause
        e.initCause(new ArithmeticException("cause"));
        throw e;
    }
    public static void main(String args[]) {
        try {
            demoproc();
        } catch (NullPointerException e) {
            // display top level exception
            System.out.println("Caught: " + e);

            // display cause exception
            System.out.println("Original cause: " + e.getCause());
        }
    }
}
```

Chained Exceptions (contd.)

- The output from the program is shown here:

```
Caught: java.lang.NullPointerException: top layer  
Original cause: java.lang.ArithmeticException:  
cause
```

- Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design.
- Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.