# Exception

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

* A user has entered invalid data.
* A file that needs to be opened cannot be found.
* A network connection has been lost in the middle of communications or the JVM has run out of memory.
* Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

# How Exception Handling

The purpose of exception handling mechanism is to provide a system to detect and report an “Exceptional circumstance” so that appropriate actions can be taken.

▪ Exception handling mechanism have following tasks

1. Find the Problem **(Hit the Exception)**
2. Inform that an error has occurred **(Throw the Exception)**
3. Receive the Error information **(Catch the Exception)**
4. Take corrective actions **(Handle the Exception)**

# How to Handle Exception

Java uses try block that contains one or more statements that could generate an exception.

Syntax

**try** **{**

**//statements that cause Exception**

**}**

catch(ExceptionType obj) {

//statements that handle Exception

}

* If any statement generates an Exception, the remaining statements in the block are skipped and execution jumps to the catch block.
* The catch block “catches” the exception “thrown” by the try block and handles it appropriately.
* The catch statement is passed a single parameter, which is reference to the Exception object thrown.
* If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed.

|  |
| --- |
| Exception Demo Arithmetic Exception |
| import java.util.\*;    public class ExceptionDemo\_131  { public static void main(String[] args)  {  Scanner scan = new Scanner(System.in); int age;    try  {  System.out.print("\n\t Enter age :");  age = scan.nextInt();    System.out.println("\n\t Age = " + age); |

Program N0. 131

|  |  |
| --- | --- |
| } | }  catch(InputMismatchException e)  {  System.out.println("\n\t Exception Caught");  System.out.println("\n\t Error : " + e); }  System.out.println("\n\t End of Program"); } |

# Exception Classes

The figure below shows the hierarchy of the Exception classes. The base class for all Exception objects is java.lang.Throwable, together with its two subclasses java.lang.Exception andjava.lang.Error.

A diagram of a error

Description automatically generated

The Error class describes internal system errors (e.g., VirtualMachineError, LinkageError) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.

The Exception class describes the error caused by your program (e.g. FileNotFoundException, IOException). These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

In Java, exceptions are objects. When you throw an exception, you throw an object. You can't throw just any object as an exception, however -- only those objects whose classes descend from Throwable. Throwable serves as the base class for an entire family of classes, declared in java.lang, that your program can instantiate and throw.

Throwable has two direct subclasses, Exception and Error. Exceptions (members of the Exception family) are thrown to signal abnormal conditions that can often be handled by some catcher, though it's possible they may not be caught and therefore could result in a dead thread. Errors (members of the Error family) are usually thrown for more serious problems, such as OutOfMemoryError, that may not be so easy to handle. In general, code you write should throw only exceptions, not errors. Errors are usually thrown by the methods of the Java API, or by the Java virtual machine itself.

In addition to throwing objects whose classes are declared in java.lang, you can throw objects of your own design. To create your own class of throwable objects, you need only declare it as a subclass of some member of

the Throwable family. In general, however, the throwable classes you define should extend class Exception. They should be "exceptions."

Whether you use an existing exception class from java.lang or create one of your own depends upon the situation. In some cases, a class from java.lang will do just fine. For example, if one of your methods is invoked with an invalid argument, you could throw IllegalArgumentException, a subclass of RuntimeException in java.lang.

## Advantages of Exceptions

Now that you know what exceptions are and how to use them, it's time to learn the advantages of using exceptions in your programs.

### Advantage 1: Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

readFile {

open the file; determine its size; allocate that much memory; read the file into memory; close the file;

}

At first glance, this function seems simple enough, but it ignores all the following potential errors.

1. What happens if the file can't be opened?
2. What happens if the length of the file can't be determined?
3. What happens if enough memory can't be allocated?
4. What happens if the read fails?
5. What happens if the file can't be closed?

To handle such cases, the readFile function must have more code to do error detection, reporting, and handling. Here is an example of what the function might look like.

errorCodeType readFile { initialize errorCode = 0;

open the file; if (theFileIsOpen) {

determine the length of the file; if (gotTheFileLength) { allocate that much memory; if (gotEnoughMemory) { read the file into memory; if (readFailed) { errorCode = -1;

}

} else { errorCode = -2;

} } else { errorCode = -3;

}

close the file;

if (theFileDidntClose && errorCode == 0) { errorCode = -4;

} else {

errorCode = errorCode and -4;

} } else { errorCode = -5;

}

return errorCode;

}

There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. Worse yet, the logical flow of the code has also been lost, thus making it difficult to tell whether the code is doing the right thing: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing when you modify the method three months after writing it. Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the readFile function used exceptions instead of traditional error-management techniques, it would look more like the following.

readFile { try {

open the file; determine its size; allocate that much memory; read the file into memory; close the file; } catch (fileOpenFailed) { doSomething;

} catch (sizeDeterminationFailed) { doSomething;

} catch (memoryAllocationFailed) { doSomething;

} catch (readFailed) {

doSomething;

} catch (fileCloseFailed) { doSomething;

}

}

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

### Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the readFile method is the fourth method in a series of nested method calls made by the main program: method1 calls method2, which calls method3, which finally calls readFile.

method1 { call method2;

} method2 { call method3;

} method3 {

call readFile; }

Suppose also that method1 is the only method interested in the errors that might occur within readFile. Traditional error-notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1—the only method that is interested in them.

method1 {

errorCodeType error; error = call method2; if (error) doErrorProcessing; else proceed;

}

errorCodeType method2 { errorCodeType error; error = call method3; if (error) return error; else proceed;

}

errorCodeType method3 { errorCodeType error; error = call readFile; if (error) return error; else proceed;

}

Recall that the Java runtime environment searches backward through the call stack to find any methods that are interested in handling a particular exception. A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors.

method1 { try { call method2; } catch (exception e) { doErrorProcessing;

} }

method2 throws exception {

call method3;

}

method3 throws exception { call readFile; }

However, as the pseudocode shows, ducking an exception requires some effort on the part of the middleman methods. Any checked exceptions that can be thrown within a method must be specified in its throws clause.

### Advantage 3: Grouping and Differentiating Error Types

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy. An example of a group of related exception classes in the Java platform are those defined in java.io —IOException and its descendants. IOException is the most general and represents any type

of error that can occur when performing I/O. Its descendants represent more specific errors. For example, FileNotFoundException means that a file could not be located on disk.

A method can write specific handlers that can handle a very specific exception. The FileNotFoundException class has no descendants, so the following handler can handle only one type of exception.

catch (FileNotFoundException e) { ...

}

A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement. For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an IOException argument.

catch (IOException e) { ...

}

This handler will be able to catch all I/O exceptions, including FileNotFoundException, EOFException, and so on. You can find details about what occurred by querying the argument passed to the exception handler. For example, use the following to print the stack trace.

catch (IOException e) {

// Output goes to System.err.

e.printStackTrace(); // Send trace to stdout.

e.printStackTrace(System.out); }

You could even set up an exception handler that handles any Exception with the handler here.

// A (too) general exception handler catch (Exception e) { ...

}

The Exception class is close to the top of the Throwable class hierarchy. Therefore, this handler will catch many other exceptions in addition to those that the handler is intended to catch. You may want to handle exceptions this way if all you want your program to do, for example, is print out an error message for the user and then exit.

In most situations, however, you want exception handlers to be as specific as possible. The reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy. In effect, by not catching specific errors, the handler must accommodate any possibility. Exception handlers that are too general can make code more error-prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.

As noted, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

## Checked vs. Unchecked Exceptions

A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. These exceptions cannot simply be ignored at the time of compilation. These exceptions are explicitly handled in code itself with the help of try – catch blocks

Unchecked Exceptions are not essentially handled in the program code, instead the JVM handles such Exceptions. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

A diagram of a flowchart

Description automatically generated

As illustrated, the subclasses of Error and RuntimeException are known as unchecked exceptions. These exceptions are not checked by the compiler, and hence, need not be caught or declared to be thrown in your program.

This is because there is not much you can do with these exceptions. For example, a "divide by 0" triggers an ArithmeticException, array index out-of-bound triggers an ArrayIndexOutOfBoundException, which are really programming logical errors that shall be been fixed in compiled-time, rather than leaving it to runtime exception handling.

All the other exceptions are called checked exceptions. They are checked by the compiler and must be caught or declared to be thrown.

## Printing Exception

### System.out.println(ex)

Prints the string representation of exception object. Prints the exception class name (java.lang.ArithmeticException) and also the exception message

System.out.println(ex.getMessage())

Prints exception message (Ex. / by zero) only indicating the cause of exception

### e.printStackTrace()

Prints the exception class name with the message particulars and also the line number where the problem arises (traces the actual problem).

## Common Exception Classes

### NumberFormatException

* It is an unchecked exception thrown by **parseXXX()** methods when they are unable to format (convert) a string into a number.
* Sometimes, in Java coding, we get input (like from command-line arguments and text field) from the user in the form of string. To use the string in arithmetic operations, it must be converted (parsed) into data types. This is done by parseXXX() methods of wrapper classes.
* Object –> Throwable –> Exception –> RuntimeException –> NumberFormatException

|  |
| --- |
| **Exception Demo** |
| public class ExceptionDemo  {  public static void main(String[] args)  {  String str = "onetwothree";  int no;    try  {  no = Integer.parseInt(str);  System.out.println("\n\t no = " + no);  }  catch(NumberFormatException e)  {  System.out.println("\n\t Error :" + e.getMessage());  }  System.out.println("\n\t End of Program");  }  } |

Program N0.

### ArrayIndexOutOfBoundsException

Thrown by JVM when your code uses an array index, which is outside the array's bounds. For example,

int[] anArray = new int[3];

System.out.println(anArray[3]);

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3

### NullPointerException

Thrown by the JVM when your code attempts to use a null reference where an object reference is required. For example,

String[] strs = new String[3];

System.out.println(strs[0].length());

Exception in thread "main" java.lang.NullPointerException

### ClassCastException

Thrown by JVM when an attempt is made to cast an object reference fails. For example,

Object o = new Object();

Integer i = (Integer)o;

Exception in thread "main" java.lang.ClassCastException: java.lang.Object cannot be cast to java.lang.Integer

### IllegalArgumentException

Thrown programmatically to indicate that a method has been passed an illegal or inappropriate argument. You could reuse this exception for your own methods.

### IllegalStateException

Thrown programmatically when a method is invoked and the program is not in an appropriate state for that method to perform its task. This typically happens when a method is invoked out of sequence, or perhaps a method is only allowed to be invoked once and an attempt is made to invoke it again.

### NoClassDefFoundError

Thrown by the JVM or class loader when the definition of a class cannot be found. Prior to JDK 1.7, you will see this exception call stack trace if you try to run a non-existent class. JDK 1.7 simplifies the error message to "Error: Could not find or load main class xxx".

## Multiple Catch Blocks

It is possible to have more than one catch statement in the try-catch block

|  |
| --- |
| **Exception Demo** |
| import java.util.\*;    public class ExceptionDemo  { public static void main(String[] args)  {  Scanner scan = new Scanner(System.in);  int n1, n2, ans;    try  {  System.out.print("\n\t Enter Number 1 :"); n1 = scan.nextInt(); |

Program N0.

|  |  |
| --- | --- |
| } | System.out.print("\n\t Enter Number 2 :");  n2 = scan.nextInt();  ans = n1/n2;  System.out.println("\n\t Answer = " + ans);  }  catch(InputMismatchException e)  {  System.out.println("\n\t Exception Caught"); e.printStackTrace();  }  catch(ArithmeticException e)  {  System.out.println("\n\t Can not / by zero");  }  catch(Exception e)  {  }  System.out.println("\n\t End of Program"); } |

When an exception in the try block is generated, the Java treats the multiple catch Statements like cases in a switch statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.

**Note:** Java does not require any processing of the exception at all. We can simply have a catch statement with an empty block to avoid program abortion

## Handling exceptions – Three styles

1. Using try-catch block; the robust way
2. Using throws in place of try-catch, not a robust way
3. To throw the exception object to the system using throw keyword, not a robust way

## Exception Handling Operations

Five keywords are used in exception handling: try, catch, finally, throws and throw (take note that there is a difference between throw and throws).

Java’s exception handling consists of three operations:

1. Declaring exceptions
2. Throwing an exception
3. Catching an exception

### Declaring Exceptions

A Java method must declare in its signature the types of checked exception it may "throw" from its body, via the keyword "throws".

For example, suppose that methodD() is defined as follows:

public void methodD() throws XxxException, YyyException {

// method body throw XxxException and YyyException }

The method's signature indicates that running methodD() may encounter two checked exceptions: XxxException and YyyException. In other words, some of the abnormal conditions insidemethodD() may trigger XxxException or YyyException.

Exceptions belonging to Error, RuntimeException and their *subclasses* need not be declared. These exceptions are called *unchecked exceptions* because they are not checked by the compiler.

### Throwing an Exception

When a Java operation encounters an abnormal situation, the method containing the erroneous statement shall create an appropriate Exception object and throw it to the Java runtime via the statement "throw XxxException". For example, public void methodD() throws XxxException, YyyException { // method's signature

// method's body

...

...

// XxxException occurs if ( ... ) throw new XxxException(...); // construct an XxxException object and throw

to JVM

...

// YyyException occurs if ( ... )

throw new YyyException(...); // construct an YyyException object and throw

to JVM ...

}

Note that the keyword to declare exception in the method's signature is "throws" and the keyword to throw an exception object within the method's body is "throw".

### Catching an Exception

When a method throws an exception, the JVM searches backward through the call stack for a matching exception handler. Each exception handler can handle one particular class of exception. An exception handler handles a specific class can also handle its subclasses. If no exception handler is found in the call stack, the program terminates.

For example, suppose methodD() declares that it may throw XxxException and YyyException in its signature, as follows:

public void methodD() throws XxxException, YyyException { ...... }

To use methodD() in your program (says in methodC()), you can either:

1. Wrap the call of methodD() inside a try-catch (or try-catch-finally) as follows. Each catch-block can contain an exception handler for one type of exception.

public void methodC() { // no exception declared ...... try { ......

// uses methodD() which declares XxxException & YyyException methodD();

......

} catch (XxxException ex) {

// Exception handler for XxxException

......

} catch (YyyException ex} {

// Exception handler for YyyException

......

} finally { // optional

// These codes always run, used for cleaning up ......

} ...... }

2. Suppose that methodC() who calls methodD() does not wish to handle the exceptions (via a try-catch), it can declare these exceptions to be thrown up the call stack in its signature as follows:

public void methodC() throws XxxException, YyyException { // for next higher-level method to handle

...

// uses methodD() which declares "throws XxxException, YyyException" methodD(); // no need for try-catch ... }

In this case, if a XxxException or YyyException is thrown by methodD(), JVM will *terminate* methodD() as well as methodC() and pass the exception object up the call stack to the caller of methodC().

## Point 1: Exceptions must be declared

As an example, suppose that you want to use a java.util.Scanner to perform formatted input from a disk file. The signature of the Scanner's constructor with a File argument is given as follows:

public Scanner(File source) throws FileNotFoundException;

The method's signature informs the programmers that an exceptional condition "file not found" may arise. By declaring the exceptions in the method's signature, programmers are made to aware of the exceptional conditions in using the method.

## Point 2: Exceptions must be handled

If a method declares an exception in its signature, you cannot use this method without handling the exception - you can't compile the program.

**Example 1:** The program did not handle the exception declared, resutled in compilation error.

import java.util.Scanner; import java.io.File; public class ScannerFromFile {

public static void main(String[] args) { Scanner in = new Scanner(new File("test.in")); // do something ...

}

}

**ERROR**

ScannerFromFile.java:5: unreported exception java.io.FileNotFoundException; must be caught or declared to be thrown

**Scanner in = new Scanner(new File("test.in"));**

To use a method that declares an exception in its signature, you MUST either:

1. provide exception handling codes in a "try-catch" or "try-catch-finally" construct, or
2. if don’t want handling the exception in the current method, but declare the exception to be thrown up the call stack for the next higher-level method to handle.

**Example 2:** Catch the exception via a "**try-catch**" (or "**try-catch-finally**") construct.

import java.util.Scanner; import java.io.File;

import java.io.FileNotFoundException; public class ScannerFromFileWithCatch { public static void main(String[] args) { try {

Scanner in = new Scanner(new File("test.in"));

// do something if no exception ...

// you main logic here in the try-block

}

catch (FileNotFoundException ex)

{

// error handling separated from the main logic

ex.printStackTrace(); // print the stack trace

}

} }

If the file cannot be found, the exception is caught in the catch-block. In this example, the error handler simply prints the *stack trace*, which provides useful information for debugging. In some situations, you may need to perform some clean-up operations, or open another file instead. Take note that the main logic in the try-block is separated from the error handling codes in the catch-block.

**Example 3:** You decided not to handle the exception in the current method, but throw the exception up the call stack for the next higher-level method to handle.

import java.util.Scanner; import java.io.File;

import java.io.FileNotFoundException; public class ScannerFromFileWithThrow {

public static void main(String[] args) throws FileNotFoundException { // to be handled by next higher-level method

Scanner in = new Scanner(new File("test.in"));

// this method may throw FileNotFoundException // main logic here ...

} }

In this example, you decided not to handle the FileNotFoundException thrown by the Scanner(File) method

(with try-catch). Instead, the caller of Scanner(File) - the main() method - declares in its signature "throws

FileNotFoundException", which means that this exce ption will be thrown up the call stack, for the next higher-level method to handle. In this case, the next higher-level method of main() is the JVM, which simply terminates the program and prints the stack trace.

## Point 3: Main logic is separated from the exception handling codes

As shown in Example 2, the main logic is contained in the try-block, while the exception handling codes are kept in the catch-block(s) separated from the main logic. This greatly improves the readability of the program.

For example, a Java program for file processing could be as follows:

try {

// Main logic here open file; process file; ......

} catch (FileNotFoundException ex) { // Exception handlers below

// Exception handler for "file not found"

} catch (IOException ex) {

// Exception handler for "IO errors"

} finally {

close file; // always try to close the file }

## Throws Keyword

### What if I really don't care about the exceptions

Certainly not advisable other than writing toy programs. But to bypass the compilation error messages triggered by methods declaring unchecked exceptions, you could declare "throws Exception" in your main() (and other methods), as follows:

public static void main(String[] args) **throws Exception** { // throws all subclass of Exception to JRE

Scanner in = new Scanner(new File("test.in")); // declares "throws FileNotFoundException" ......

// other exceptions

}

## Method Call Stack

A typical application involves many levels of method calls, which is managed by a so-called *method call stack*. A *stack* is a last-in-first-out queue. In the following example, main() method invokes methodA(); methodA() calls methodB(); methodB() calls methodC(). A screen shot of a cell phone

Description automatically generated

public class MethodCallStackDemo { public static void main(String[] args) { System.out.println("Enter main()"); methodA();

System.out.println("Exit main()");

}

public static void methodA() {

System.out.println("Enter methodA()"); methodB();

System.out.println("Exit methodA()");

}

public static void methodB() {

System.out.println("Enter methodB()"); methodC();

System.out.println("Exit methodB()");

}

public static void methodC() {

System.out.println("Enter methodC()");

System.out.println("Exit methodC()");

}

}

Enter main()

Enter methodA()

Enter methodB()

Enter methodC()

Exit methodC()

Exit methodB()

Exit methodA()

Exit main()

As seen from the output, the sequence of events is:

1. JVM invoke the main().
2. main() pushed onto call stack, before invoking methodA().
3. methodA() pushed onto call stack, before invoking methodB().
4. methodB() pushed onto call stack, before invoking methodC().
5. methodC() completes.
6. methodB() popped out from call stack and completes.
7. methodA() popped out from the call stack and completes.
8. main() popped out from the call stack and completes. Program exits.

Suppose that we modify methodC() to carry out a "divide-by-0" operation, which triggers a ArithmeticException:

public static void methodC() {

System.out.println("Enter methodC()");

System.out.println(1 / 0); // divide-by-0 triggers an ArithmeticException

System.out.println("Exit methodC()"); }

The exception message clearly shows the *method call stack trace* with the relevant statement line numbers:

Enter main()

Enter methodA()

Enter methodB()

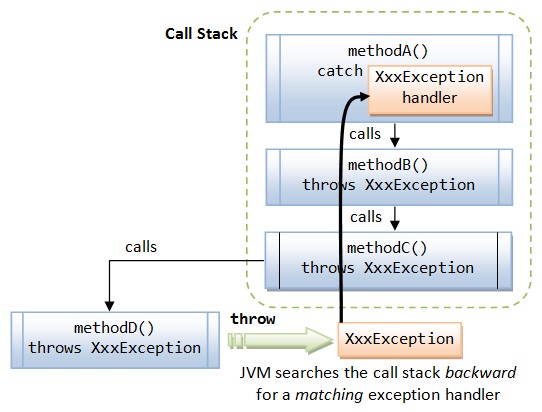
Enter methodC()

Exception in thread "main" java.lang.ArithmeticException: / by zero at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22) at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16) at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10) at MethodCallStackDemo.main(MethodCallStackDemo.java:4)

MethodC() triggers an ArithmeticException. As it does not handle this exception, it popped off from the call stack immediately. MethodB() also does not handle this exception and popped off the call stack.

So does methodA() and main() method. The main() method passes back to JVM, which abruptly terminates the program and print the call stack trace, as shown.

## Exception & Call Stack



When an exception occurs inside a Java method, the method creates an Exception object and passes the Exception object to the JVM (in Java term, the method "throw" an Exception).

The Exceptionobject contains the type of the exception, and the state of the program when the exception occurs. The JVM is responsible for finding an exception handler to process the Exception object. It searches backward through the call stack until it finds a matching exception handler for that particular class of Exception object (in Java term, it is called "catch" the Exception). If the JVM cannot find a matching exception handler in all the methods in the call stack, it terminates the program.

This process is illustrated as follows. Suppose that methodD() encounters an abnormal condition and throws a XxxException to the JVM. The JVM searches backward through the call stack for a matching exception handler. It finds methodA() having a XxxException handler and passes the exception object to the handler. Notice

that methodC() and methodB() are required to declare "throws XxxException" in their method signatures in order to compile the program.

## try-with-resources structure

So far you have been familiar with the try-catch-finally structure. Now I’m about to tell you the advanced version of exception handling in Java - it is the **try-with-resources** structure which was added to the Java language from Java SE 7.

Let’s look at a typical try-catch-finally example I showed you previously:

FileWriter writer = null; try {

writer = new FileWriter("Name.txt"); writer.write("Hello "); writer.close(); } catch (IOException e) {

e.printStackTrace(); } finally {

if (writer != null) { try {

writer.close();

} catch (IOException ce) { ce.printStackTrace();

}

}

}

The finally block is usually used to close a resource such as a file, a network connection, a database connection and the like. This pattern is repeated again and again so Java 7 makes our lives easier by enhancing the exception handling with the introduction of try-with-resources structure.

The above code can be re-written using the try-with-resources construct as follows:

try (FileWriter writer = new FileWriter("Name.txt")) { writer.write("Hello "); writer.close(); } catch (IOException e) {

e.printStackTrace(); }

With this new structure, we don’t have to explicitly close the resource used by the finally block. Instead, the Java compiler will figure it out and automatically adds code to close the resource for us.

Well, the secret lies in the **AutoCloseable** interface that defines only a single method:

public void close();

So when a resource used by the try block implements this interface, the compiler knows that it’s safe to call the **close()** method on the resource object.

That means the try-with-resources structure works only with AutoCloseable’s implementations. And fortunately, Java 7 refactors almost resource-like classes to implement this interface to support programmers.

Thanks to the try-with-resources construct that brings us the following benefits:

* We can write more compact code as eliminating the finally block. This saves time.
* We can write more safe and efficient code as if we forget to close a resource, the compiler does the work for us behind the scenes. Using a database connection within a **try-catch-finally** structure:

Connection conn = null;

try {

String dbURL = "jdbc:oracle:thin:tiger/scott@localhost:1521:DB"; conn = DriverManager.getConnection(dbURL);

// execute SQL statements } catch (SQLException ex) { ex.printStackTrace();

} finally { try {

if (conn != null && !conn.isClosed()) { conn.close();

}

} catch (SQLException ex) { ex.printStackTrace();

}

}

It is now more compact with try-with-resources version:

String dbURL = "jdbc:oracle:thin:tiger/scott@localhost:1521:DB";

try (Connection conn = DriverManager.getConnection(dbURL)) {

// execute SQL statements } catch (SQLException ex) { ex.printStackTrace();

}

**NOTE:**we can use initialize multiple resources in the try block and the compiler is smart enough to close them all. Here’s an example that copies one file to another using the try-catch-finally fashion:

public void copyFile(File sourceFile, File destFile)

throwsIOException {

FileChannel sourceChannel = null; FileChannel destChannel = null; try{

sourceChannel = newFileInputStream(sourceFile).getChannel(); destChannel = newFileOutputStream(destFile).getChannel(); sourceChannel.transferTo(0, sourceChannel.size(), destChannel);

} finally{

if(sourceChannel != null) { sourceChannel.close();

}

if(destChannel != null) { destChannel.close();

}

}

}

And now with the try-with-resources fashion:

public void copyFile(File sourceFile, File destFile) throwsIOException {

try(

FileChannel sourceChannel = newFileInputStream(sourceFile).getChannel(); FileChannel destChannel = newFileOutputStream(destFile).getChannel();

) {

sourceChannel.transferTo(0, sourceChannel.size(), destChannel); }

}

Exception Chaining?

Basically, exception chaining is the process of re-throwing multiple exceptions across different abstraction layers of a program. The key principle here is that, these exceptions are chained together to maintain the stack trace from the exception at the lowest layer to the one at the highest layer. The following picture illustrates this concept visually:

A diagram of a diagram

Description automatically generated

As you can see, each abstraction layer defines its own exception classes. When code in a layer throws an exception, the higher layer re-throws it under a new type of exception which corresponds to the abstraction level of that layer. In turn, the next higher layer re-throws the exception under its own type of exception, and so on. This process continues until a layer handles the exception instead of re-throwing. During this chaining process, the higher exception always wraps the lower exception as its cause. Therefore, when an exception occurs, the programmer has a complete stack trace of the exceptions, which is very helpful for debugging.

* **Why is Exception Chaining?**

The main purpose of exception chaining is to preserve the original exception when it propagates across multiple logical layers in a program. This is very helpful for the debugging process when an exception is thrown, as the programmer can analyze the full stack trace of the exceptions.

In addition, exception chaining also helps promoting abstraction among logical layers in a program, as each layer defines its own exceptions which are specific for that layer. For example, the StudentBusinessclass throws StudentException would be more meaningful than SQLException, right?

You know, exception chaining is sometimes referred as *exception propagation*, as when a layer throws an exception, the exception propagates through higher layers until a layer handles it such as displaying a message/warning to the user.

* **How to Chain Exceptions Together?**

Let’s consider the following code example:

public void setBirthday(String birthDate) throws InvalidBirthdayException {

DateFormat formatter = new SimpleDateFormat(); try {

Date birthday = formatter.parse(birthDate);

} catch (ParseException ex) {

throw new InvalidBirthdayException("Date of birth is invalid", ex);

}

}

As you can see in the setBirthday() method, the ParseException is re-thrown under a new exception called InvalidBirthdayException. The ParseException is chained via the constructor of InvalidBirthdayException class:

throw new InvalidBirthdayException("Date of birth is invalid", ex);

This custom exception is implemented as following:

public class InvalidBirthdayException extends Exception {

public InvalidBirthdayException(String message, Throwable cause) { super(message, cause);

}

}

You can notice that, this constructor invokes its super’s constructor:

super(message, cause);

The supertypes of all exceptions Throwable and Exception implement this constructor, so any custom exceptions can call it. The origin exception (the cause) is passed to the being-created exception via its constructor.

Remember that the Exception class provides the following constructors that help chaining an exception:

* Exception(Throwable cause)
* Exception(String message, Throwable cause)

Besides chaining an exception via constructor, you can also chain an exception through the following Throwable’s method:

public Throwable initCause(Throwable cause)

That’s how exceptions are chained together.

Let’s see another example which is illustrated by the following picture:

A diagram of a student chain

Description automatically generated

And following is source code of each class.

**DAOException.java:**

public class DAOException extends Exception { public DAOException(String message, Throwable cause) { super(message, cause);

}

}

**StudentException.java:**

public class StudentException extends Exception { public StudentException(String message) { super(message);

}

public StudentException(String message, Throwable cause) { super(message, cause);

}

}

**DatabaseUtils.java:**

import java.sql.\*; public class DatabaseUtils {

public static void executeQuery(String sql) throws SQLException { throw new SQLException("Syntax Error");

}

}

**StudentDAO.java:**

import java.sql.\*; public class StudentDAO {

public void list() throws DAOException {

try {

DatabaseUtils.executeQuery("SELECT");

} catch (SQLException ex) {

throw new DAOException("Error querying students from database", ex);

}

}

}

**StudentManager.java:**

public class StudentManager { private StudentDAO dao;

public StudentManager(StudentDAO dao) { this.dao = dao;

}

public void findStudents(String keyword) throws StudentException { try {

dao.list();

} catch (DAOException ex) {

throw new StudentException("Error finding students", ex);

}

}

}

**StudentProgram.java:**

public class StudentProgram {

public static void main(String[] args) { StudentDAO dao = new StudentDAO();

StudentManager manager = new StudentManager(dao); try {

manager.findStudents("Tom"); } catch (StudentException ex) { ex.printStackTrace();

}

}

}

Run the StudentProgram and you should see the following output:

StudentException: Error finding students

at StudentManager.findStudents(StudentManager.java:13) at StudentProgram.main(StudentProgram.java:9)

Caused by: DAOException: Error querying students from database at StudentDAO.list(StudentDAO.java:11)

at StudentManager.findStudents(StudentManager.java:11)

... 1 more

Caused by: java.sql.SQLException: Syntax Error at DatabaseUtils.executeQuery(DatabaseUtils.java:5) at StudentDAO.list(StudentDAO.java:8)

... 2 more

You see? The printed exception stack trace reveals an exception propagates from the DatabaseUtils layer up to the StudentProgram layer in which the exception is handled by printing this trace.

## try-catch-finally

The syntax of try-catch-finally is: **try {**

// main logic, uses methods that may throw Exceptions ......

**} catch (***Exception1 ex***) {**

// error handler for Exception1 ......

**} catch (***Exception2 ex***) {**

// error handler for Exception1 ......

**} finally {** // finally is optional

// clean up codes, always executed regardless of exceptions ......

**}**

If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and finally-block will be executed after the try-block. If one of the statements in the try-block throws an exception, the Java runtime ignores the rest of the statements in the try-block, and begins searching for a matching exception handler. It matches the exception type with each of the catch-blocks sequentially.

If a catch-block catches that exception class or catches a *superclass* of that exception, the statement in that catch-block will be executed. The statements in the finally-block are then executed after that catch-block. The program continues into the next statement after the try-catch-finally, unless it is pre-maturely terminated or branch-out.

If none of the catch-block matches, the exception will be passed up the call stack. The current method executes the finally clause (if any) and popped off the call stack. The caller follows the same procedures to handle the exception.

The finally block is almost certain to be executed, regardless of whether or not exception occurs (unless JVM encountered a severe error or a System.exit() is called in the catch block).

Example 1 import java.util.Scanner; import java.io.File;

import java.io.FileNotFoundException; public class TryCatchFinally {

public static void main(String[] args) {

try { // main logic

System.out.println("Start of the main logic"); System.out.println("Try opening a file ...");

Scanner in = new Scanner(new File("test.in"));

System.out.println("File Found, processing the file ...");

System.out.println("End of the main logic");

} catch (FileNotFoundException ex) { // error handling separated from the main logic

System.out.println("File Not Found caught ...");

} finally { // always run regardless of exception status

System.out.println("finally-block runs regardless of the state of exception");

}

// after the try-catch-finally

System.out.println("After try-catch-finally, life goes on...");

}

}

This is the output when the FileNotFoundException triggered: Start of the main logic Try opening a file ... File Not Found caught ... finally-block runs regardless of the state of exception After try-catch-finally, life goes on...

This is the output when no exception triggered:

Start of the main logic Try opening a file ...

File Found, processing the file ... End of the main logic

finally-block runs regardless of the state of exception After try-catch-finally, life goes on..

### try-catch-finally

1. A try-block must be accompanied by at least one catch-block or a finally-block.
2. You can have multiple catch-blocks. Each catch-block catches only one type of exception.
3. A catch block requires one argument, which is a throwable object (i.e., a subclass of java.lang.Throwable), as follows:

catch (AThrowableSubClass aThrowableObject) {

// exception handling codes

}

You can use the following methods to retrieve the type of the exception and the state of the program from the Throwable object:

**printStackTrace ():** Prints this Throwable and its call stack trace to the standard error stream System.err. The first line of the outputs contains the result of toString(), and the remaining lines are the stack trace. This is the most common handler, if there is nothing better that you can do. For example,

try {

Scanner in = new Scanner(new File("test.in"));

// process the file here

......

} catch (FileNotFoundException ex) {

**ex.printStackTrace();**

}

You can also use printStackTrace(PrintStream s) or printStackTrace(PrintWriter s).

* **getMessage():** Returns the message specified if the object is constructed using constructor Throwable(String

message).

* **toString():** Returns a short description of this Throwable object, consists of the name of the class, a colon ':', and a message from getMessage().

* A catch block catching a specific exception class can also catch its *subclasses*. Hence, catch(Exception ex) {...} catches all kinds of exceptions. However, this is not a good practice as the exception handler that is too general may unintentionally catches some subclasses' exceptions it does not intend to.

* The order of catch-blocks is important. A subclass must be caught (and placed in front) before its superclass. Otherwise, you receive a compilation error "exception XxxException has already been caught".

* The finally-block is meant for cleanup code such as closing the file, database connection regardless of whether the try block succeeds. The finally block is always executed (unless the catch-block pre-maturely terminated the current method).

|  |
| --- |
| Finally Block |
| public class ExceptionDemo  {  public static void main(String[] args)  {  String str = "onetwothree";  int no;    try  {  no = Integer.parseInt(str);  System.out.println("\n\t no = " + no);  }  catch(NumberFormatException e)  {  System.out.println("\n\t Error :" +  e.getMessage());  } |

Program N0.

|  |  |  |
| --- | --- | --- |
|  |  | finally  {  System.out.println("\n\t End of Program"); } |
| } | } |  |