EXCEPTIONS THAT YOU CATCH IN A MULTI-CATCH BLOCK CAN’T SHARE AN INHERITANCE RELATIONSHIP

What happens if you add another line of code in the previous example, which involves reading from FileInputStream, which might throw an IOException? Let’s add IOException to the list of exceptions being caught in the multi-catch block:

class MultiCatch {

void myMethod(Connection con, String fileName) {

try {

File file = new File(fileName);// **May throw FileNotFoundException**

FileInputStream fin = new FileInputStream(file);

fin.read();//**May throw IOException**

Statement stmt = con.createStatement();//**May throw SQLException**

}

// **Fails to compile**

catch (**IOException| FileNotFoundException** | SQLException e) {

System.out.println(e.toString());

}

}

}

This code fails compilation with the following error message:

Alternatives in a multi-catch statement cannot be related by sub classing.

Looks like the code fails to compile because the IOException is caught before the FileNotFoundException. In regular catch blocks, if you catch a superclass exception before a derived class exception, the code won’t compile. So let’s swap the order of IOException and FileNotFoundException in the preceding code:

try {

//Exception code here

}

//**Swapping exception types doesn’t make a difference; code fails to compile.**

catch (**FileNotFoundException | IOException|** SQLException e) {

System.out.println(e.toString());

}

The correct multi-catch block for code that may throw an IOException, FileNotFoundException, and SQLException is as follows:

try {

//Exception code here

}

// **Catch IOException (superclass of FileNotFoundException) and SQLException or any of their subclasses.**

catch (**IOException | SQLException e**) {

System.out.println(e.toString());

}

}

Combining multi-catch and single-catch blocks

try {

//Exception code here.

}

//**Catch FileNotFoundException or any of its subclasses.**

catch (FileNotFoundException e) {}

//**Catch IOException and SQLException or any of their subclasses (except for FileNotFoundException or its subclasses).**

catch (IOException | SQLException e) {}

EXAM TIP Watch out for a combination of multi-catch and single-catch exception handlers on the exam. They can get quite tricky.

In multi-catch block we should use single variable.

catch (SQLException | IOException e ) {...}//**Compiles**

catch (SQLException el | IOException e2 ) {...}//**Fails to compile**

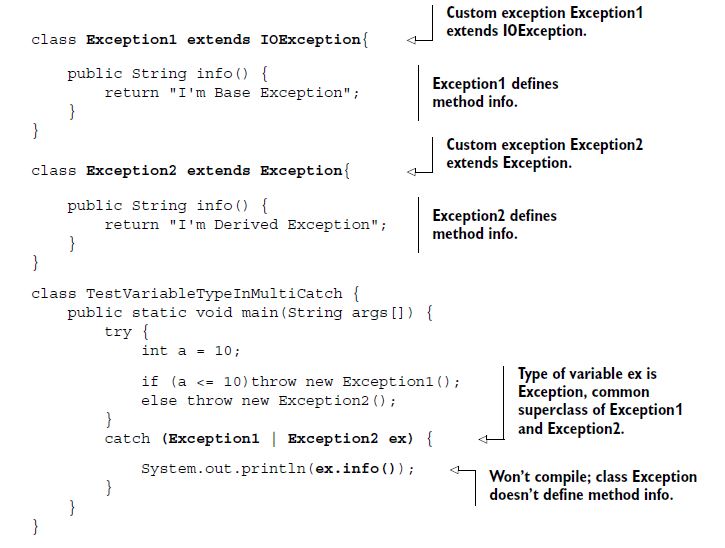
In a multi-catch block, variable e is implicitly final.

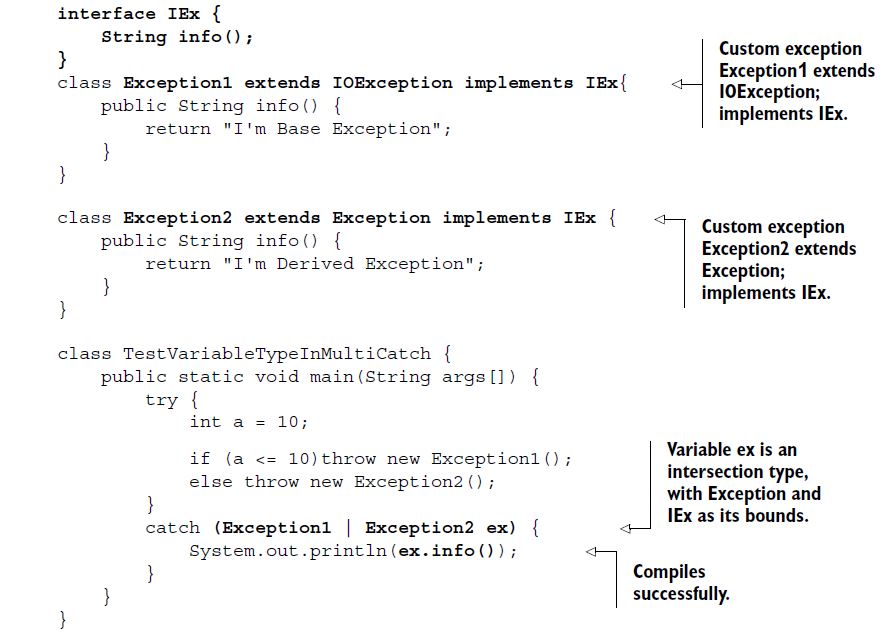
catch (IOException| SQLException e) {

**e = new FileNotFoundException();**//**Won’t compile**

}

Type of exception variable in a multi-catch block is Exception





You can call methods accessible to class Exception and interface IEx on the reference variable ex.

***Auto-closing resources with a try-with-resources***

try {

}

finally {

if (fis != null)

try {

fis.close();

}

catch (Exception e) {/\* \*/}

}

As you can see, closing a resource required a lot of boilerplate code and was error prone too. What if a developer didn’t close a resource in a finally block?

If you declare a resource by using a try-with-resources statement, it automatically *closes* the resource by calling its close method, *just* before the end of the try block. A resource must implement the java.lang.AutoCloseable interface or any of its sub interfaces to be eligible to be declared in a try-with-resources statement.

NOTE FileInputStream implements java.io.Closeable, which, starting with Java 7, extends java.lang.AutoCloseable. So FileInputStream is a valid resource to be used by the try statement.

**try (FileInputStream fin = new FileInputStream(file)){**

//.. some code

}

// **FileInputStream; can throw FileNotFoundException**

catch (FileNotFoundException e) {

System.out.println(e.toString());

}

But this code doesn’t compile and gives the following compilation error: AutoClose.java:7: error: unreported exception IOException; must be caught or declared to be thrown.

The try-with-resources statement calls method close() *just* before the completion of the try block. Method close() throws IOException we must handle it.

try (FileInputStream fin = new FileInputStream(file)){

//.. some code

}

catch (**IOException** e) {

System.out.println(e.toString());

}

EXAM TIP Watch out for questions on the exam that include or exclude a semicolon at the end of the resource defined by a try-with-resources statement. A try-with resources statement can declare multiple resources, which are separated by a semicolon. After the last resource declaration, a semicolon is optional.

***Suppressed exceptions***

In a try-with-resources statement, if both the code in the try block and close() throw an exception, the exception thrown by close() is *suppressed* by the exception thrown by the try block.

The resources initialized by the try-with-resources statement are automatically closed, just before the end of execution of the try block. This happens regardless of whether any exceptions are thrown or not thrown.

class RiverRaft implements AutoCloseable {

public RiverRaft() throws Exception {

System.out.println("Start raft");

}

public void crossRapid() throws Exception {

System.out.println("Cross rapid");

throw new Exception("Cross Rapid exception");

}

public void close() throws Exception {

System.out.println("Reach river bank");

}

}

The class SuppressedExceptions initializes an instance of RiverRaft by using a try-with-resources statement:

public class SuppressedExceptions {

public static void main(String[] args) throws Exception {

try ( RiverRaft raft = new RiverRaft(); ) {

raft.crossRapid();

}

catch (Exception e) {

System.out.println("Exception caught:" + e);

}

}

}

**Output:**

Start raft

Cross rapid

Reach river bank

Exception caught:java.lang.Exception: Cross Rapid exception

Now what happens if close() in RiverRaft also throws an exception? Which exception will be propagated to the exception handler? Will it be the exception from close() or from crossRapid()?

public class SuppressedExceptions {

public static void main(String[] args) throws Exception {

try ( RiverRaft raft = new RiverRaft(); ) {

raft.crossRapid();

}

catch (Exception e) {

System.out.println("Exception caught:" + e);

}

}

}

class RiverRaft implements AutoCloseable {

public RiverRaft() throws Exception {

System.out.println("Start raft");

}

public void crossRapid() throws Exception {

System.out.println("Cross rapid");

throw new Exception("Cross Rapid exception");

}

public void close() throws Exception {

System.out.println("Reach river bank");

**throw new Exception("Close exception");**

}

}

**Output:**

Start raft

Cross rapid

Reach river bank

Exception caught:java.lang.Exception: Cross Rapid exception

Because the output of the previous code snippets looks identical, what do you think happened to the exception thrown by method close()? This exception was *suppressed* by the exception thrown by crossRapid().

You can retrieve the suppressed exceptions by calling getSuppressed() on the exception that has suppressed the other exceptions. The getSuppressed() method returns an array containing all of the exceptions that were suppressed in order to deliver the exception thrown by crossRapid().

Throwable[] exs = e.getSuppressed();

if (exs.length>0)

System.out.println(exs[0]);//**Retrieves and prints first suppressed exception.**

The getSuppressed() never returns null. If there aren’t any suppressed expressions, the length of the returned array is 0.

EXAM TIP The variables defined in a try-with-resources statement are implicitly final.

The variables used to refer to resources are implicitly final variables. You must *declare* and *initialize* resources in the try-with-resources statement. You can’t define un-initialized resources:

try (FileInputStream fin; FileOutputStream fout;){//**Won’t compile;**

}

It’s acceptable to the Java compiler to initialize the resources in a try-with-resources statement to null, only as long as they aren’t being reassigned a value in the try block.

try (FileInputStream fin = null; FileOutputStream fout = null;){// **Compiles**

}

try (FileInputStream fin = null; FileOutputStream fout = null;){

fin = new FileInputStream(inFile);// **Won’t compile**

}

The resource declared by try-with-resources scope is limited to the try block.

try (FileInputStream fin = new FileInputStream(inFile);){

}

finally {

fin.read();//**Won’t compile out of scope.**

}

Exceptions that are thrown by method close() defined in the AutoCloseable and Closeable interfaces.

package java.lang;

public interface AutoCloseable {

void close() **throws Exception**;

}

package java.io;

public interface Closeable extends AutoCloseable {

public void close() **throws IOException**;

}

Method close() in the Closeable interface overrides method close() from the AutoCloseable base interface.

THE RESOURCES DECLARED WITH TRY-WITH-RESOURCES ARE CLOSED IN THE REVERSE ORDER OF THEIR DECLARATION

class MyResource implements AutoCloseable{

String name;

MyResource(String name) {

this.name = name;

System.out.println("Created:"+name);

}

public void close() {

System.out.println("Closed:"+name);

}

}

class TestAutoCloseOrder {

public static void main(String args[]) {

try (MyResource res1 = new MyResource("1");

MyResource res2 = new MyResource("2")){

System.out.println("within try-with-resources");

}

finally {

System.out.println("finally");

}

}

}

**Output:**

Created:1

Created:2

within try-with-resources

Closed:2

Closed:1

finally