Chapter 6 – Exception Handling

Contents

[1 Runtime Errors & Exceptions 2](#_Toc161481666)

[2 The try/catch Block 4](#_Toc161481667)

[3 The *Exception* Class Hierarchy 6](#_Toc161481668)

[4 The *catch* Block 7](#_Toc161481669)

[4.1 Exercises 7](#_Toc161481670)

[5 Anticipating & Handling Errors 8](#_Toc161481671)

[6 Throwing Exceptions 9](#_Toc161481672)

[6.1 Exercises 10](#_Toc161481673)

[7 The *finally* Clause 12](#_Toc161481674)

[8 Exception Types 13](#_Toc161481675)

[9 A Real Example 14](#_Toc161481676)

[Appendix 1 Resources 15](#_Toc161481677)

[Appendix 2 Catching Multiple Exceptions 15](#_Toc161481678)

[Appendix 2.1 Exercises 16](#_Toc161481679)

[Appendix 3 Rethrowing Exceptions 16](#_Toc161481680)

[Appendix 3.1 Exercises 18](#_Toc161481681)

[Appendix 4 Chained Exceptions 20](#_Toc161481682)

[Appendix 4.1 Exercises 21](#_Toc161481683)

[Appendix 5 Summary of Error Handling Approaches 21](#_Toc161481684)

[Appendix 6 Custom Exceptions 22](#_Toc161481685)

[Appendix 6.1 Exercises 23](#_Toc161481686)

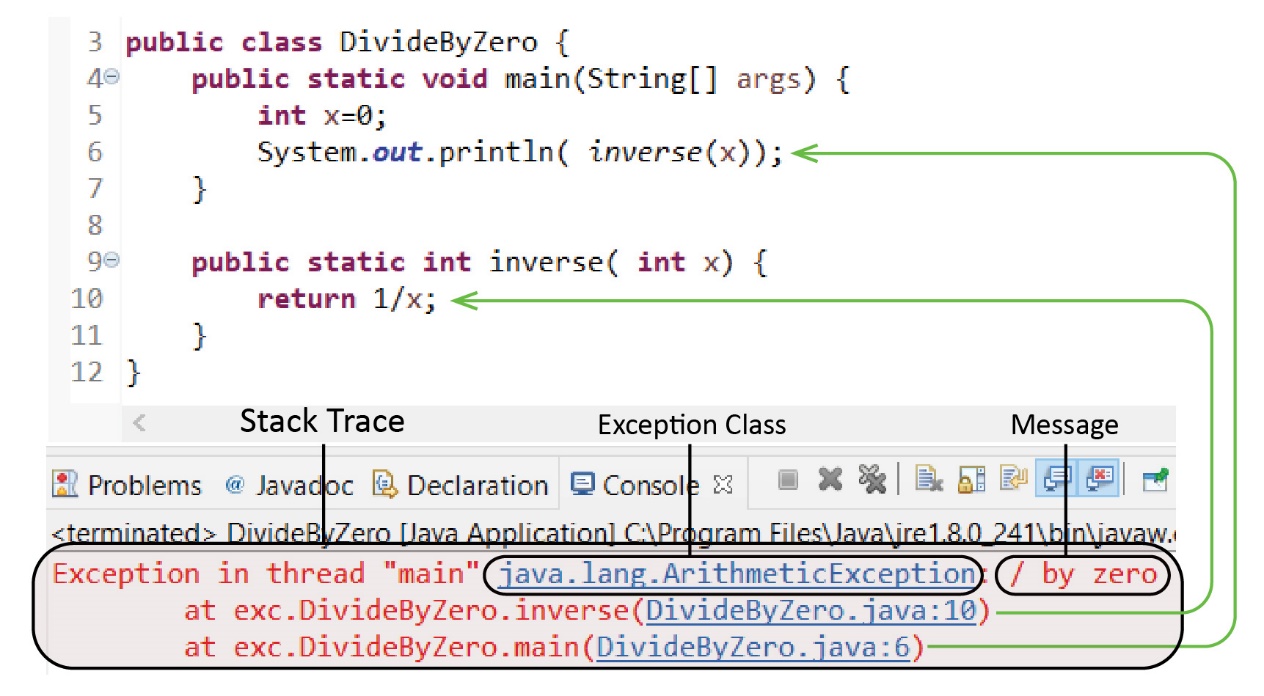
[Appendix 7 Checked Exceptions in the API 24](#_Toc161481687)

# Runtime Errors & Exceptions

The code in this section is in the *example\_exceptions\_in\_api* package.

A *runtime error*occurs when code performs an illegal operation (*e.g* divide by zero, access an array with an invalid index, *etc.*). When the JVM detects that a line of code is performing an illegal operation, it creates an *Exception* object and *throws* it. If the exception is not *caught* (we learn what this means shortly) the program ends.

As an example, in the code below, we see in line 6, that the *inverse* method is called, passing the value 0. Then, on line 10, there is an attempt to divide by zero, which causes the JVM to throw an *ArithmeticException*,which is not caught, and so the program ends displaying a *stack trace* in the console.

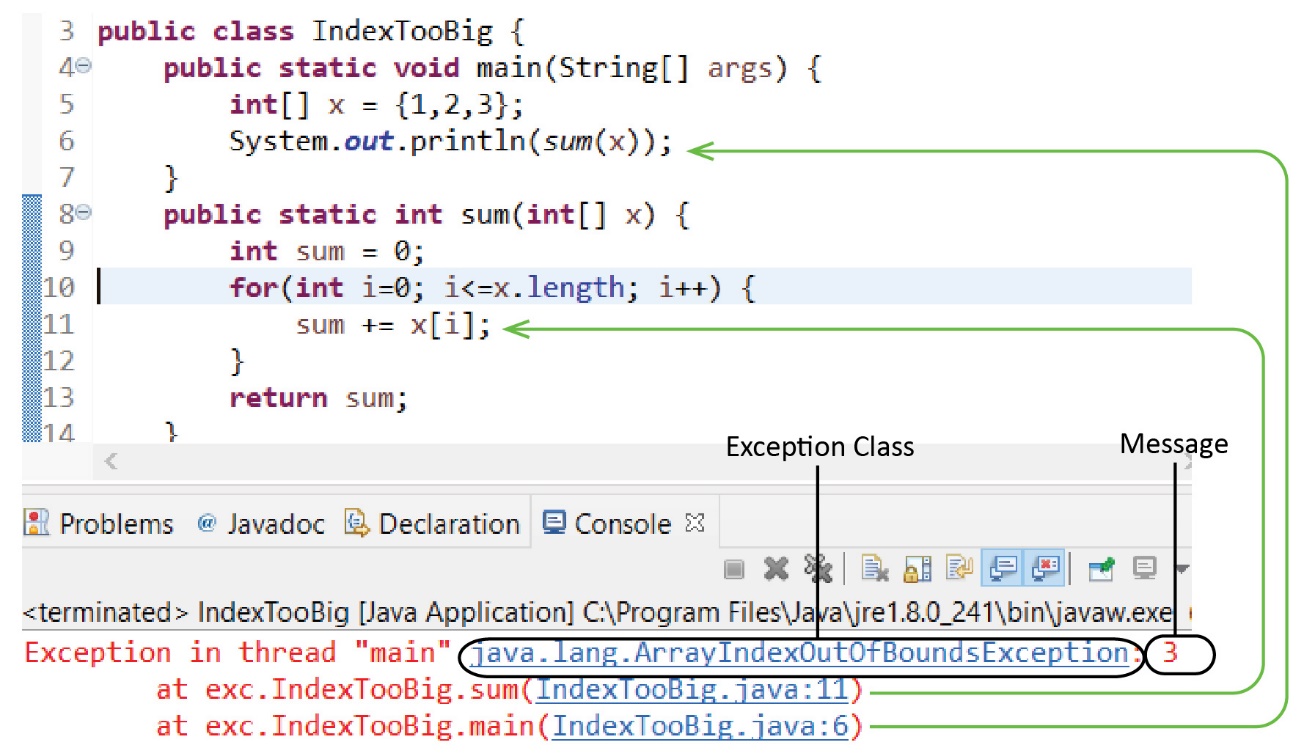


A *stack trace* provides useful information:

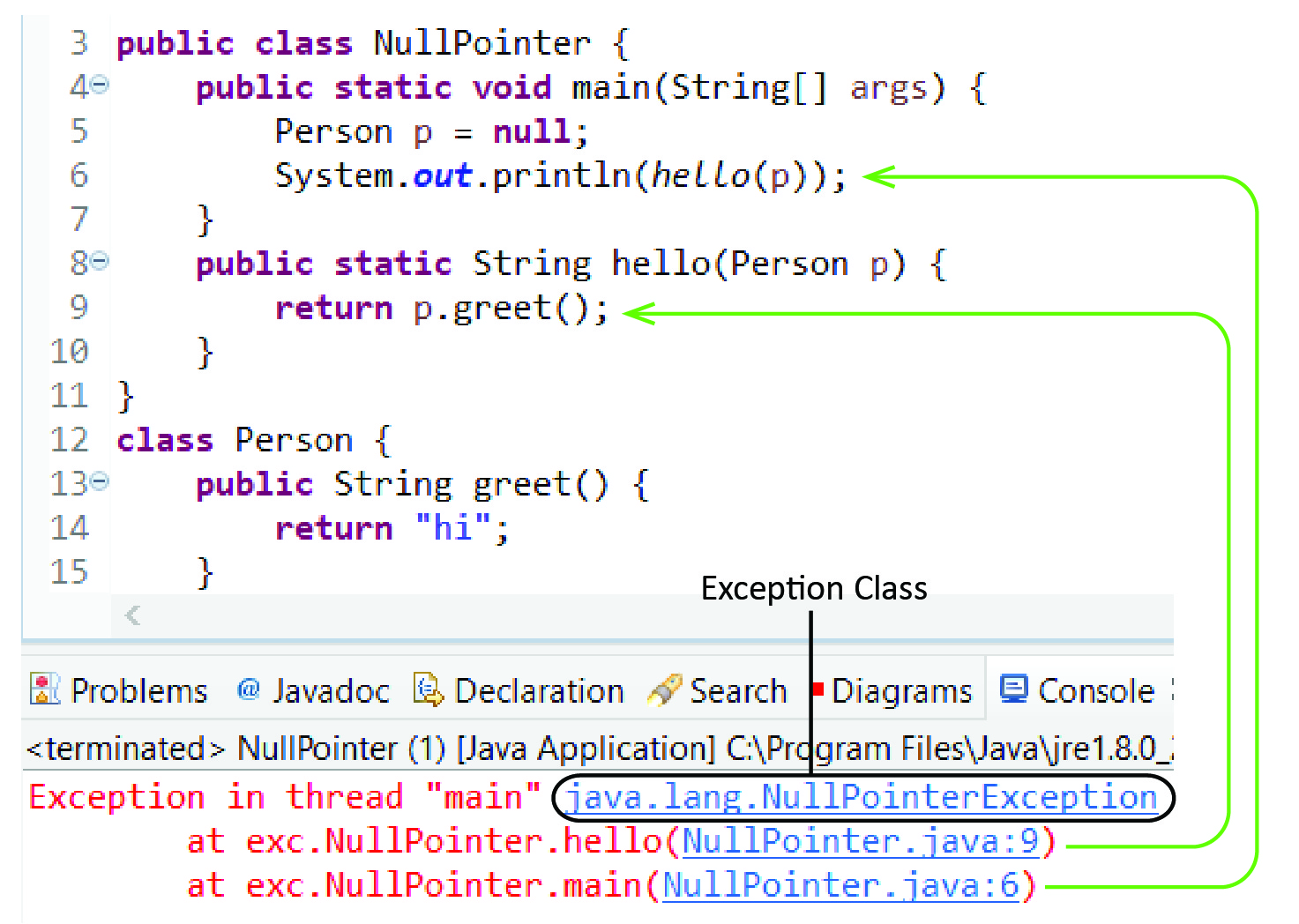
* The first entry in a stack trace shows the name of the *exception class* for the exception that was thrown, followed by the *message*, a brief (often cryptic) description
* The second entry shows the line where the run-time error occurred. In the example above, line 10.
* All subsequent entries show the exact sequence of method calls that lead to the illegal operation. In the example above, there is only one method, *inverse*, which is called on line 6.

As a programmer, you must learn to interpret and use a stack trace to resolve errors. Eclipse helps with this by providing links to the sequence of method calls that lead to the error. You typically work backwards. First, you inspect any code before the line of code that caused the error. Then, you move to the method that called the previous one and inspect the code before the call, *etc*.

In the example below, an *ArrayIndexOutOfBoundsException* is thrown. As you can see in the loop (line 11), the termination condition is incorrect (it should be i<x.length). The *message* is “3”, which means the illegal index was 3.



In the example below, a *NullPointerException* is thrown and there is no *message*. Of course, the problem is that on line 9, the *Person* object, *p* is null.



In this chapter, we discuss: how and when to *catch, handle, create* and *throw* exceptions. The appendices for this chapter consider: catching multiple exceptions, rethrowing exceptions, chaining exceptions, and creating and using custom exceptions.

# The try/catch Block

The Java language provides a *try/catch block* to allow a program to continue running when an exception is thrown. As shown below, we use a *try* block to surround a block of code that could potentially cause a runtime error. Immediately following is a *catch* block. If a runtime error occurs in the *try* block, control is immediately transferred to the *catch* block. The purpose of the *catch* block is to correct or mitigate the problem (more on this later). When the *catch* block is complete, the code immediately after the *catch* block executes and the program continues with normal execution. The general syntax is:

**try** {

// code that may fail (cause a run-time error)

}

**catch**(Exception e) {

// Code to execute when an exception is thrown

|  |
| --- |
| **Exception Thrown** |
| Shows try/catch block where statement throws an exception.  Output: 10 |

}

When a program has a runtime error, the JVM throws an exception object. Then, the JVM follows this general algorithm:

if exception is caught

handle exception – execute code to recover

resume normal program execution

else

program ends

When we use *try/catch,* the two possible paths of execution:

1. **Exception Thrown** – see the example on the right and note the following (Code in *example\_first\_examples,* in *ExceptionInMain* class):

* Step 1 – An attempt is made to access the 12th element in an array that only holds 3 elements and an exception is thrown.
* Step 2 – Control is transferred to the *catch* block where *z* is assigned (arbitrarily) the value 10.
* Step 3 – The *catch* block ends and program execution resumes, printing *z.*

Additionally, note:

* The *catch* block looks like a method, *i.e.* it defines a parameter, in the example above: *Exception e.*
* A reference to the exception that was thrown, *e* is available in the *catch* block; however, we do not use it here. Later, we will show how we can use it.

Summary: When a run-time error occurs on a line of code in the *try* block, an *Exception* is thrown and control is immediately transferred to the *catch* block. The code in the *catch* block is executed and when complete, the code immediately after the *catch* block is executed (assuming no run-time error occurs in the *catch* block). Thus, any lines of code after the line that caused the run-time error in the *try* block are not executed.

|  |
| --- |
| **No Exception Thrown** |
| Shows try/catch block where statement does not throw an exception.  Output: 4 4 |

1. **No Exception Thrown** – see the example on the right and note the following:

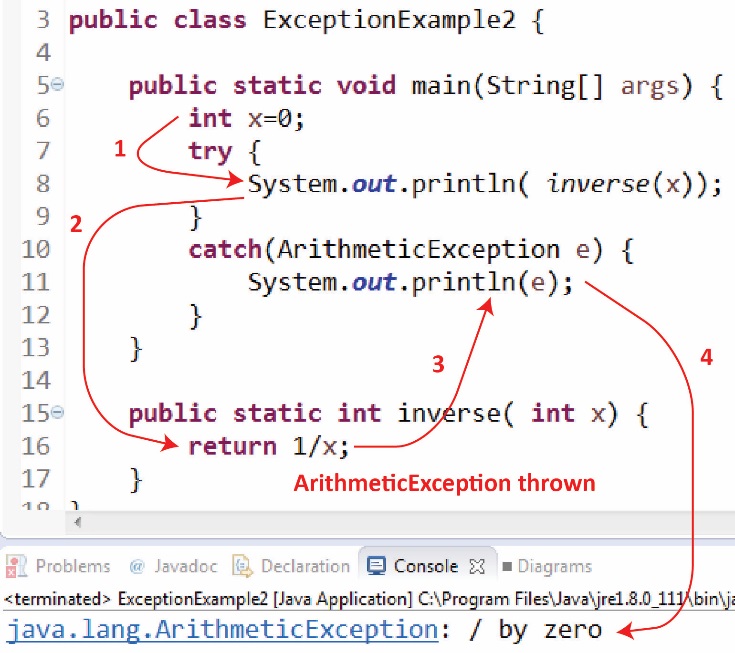
* Step 1 – An attempt is made to access the 3rd element in an array that holds 3 elements, which is successful.
* Step 2 – The code continues, printing *z*.
* Step 3 – The *try* block ends and program execution resumes after the *catch* block, printing the value of *z* again.

Summary: At the conclusion of the *try* block (*i.e.* no runtime error), control is transferred to the first line immediately after the *catch* block. Thus, the *catch* block is not executed

Consider the pseudo-code below and assume that *statement\_3* is subject to run-time failure (or calls a method that is subject to run-time failure). Which statements execute successfully when:

1. Statement 3 fails?
2. Statement 3 succeeds?

|  |  |
| --- | --- |
| **Code** | **Statements that execute successfully when:** |
| statement\_1;  try {  statement\_2;  statement\_3;  statement\_4;  }  catch( Exception e ) {  statement\_5;  }  statement\_6; | 1. *statement 3* fails: 1, 2, 5, 6 2. *statement 3* succeeds: 1, 2, 3, 4, 6 |

****The examples we have considered so far, that use *try/catch* illustrated an exception being thrown and caught in the same method (*main*). When a runtime error occurs in a method without a *try/catch* block, then the exception is thrown to the calling method. If there is a *try/catch* block there, then the JVM attempts to catch it there. If not, it is again thrown to its calling method. In general, an exception is thrown up the call hierarchy until it is caught. If it finally gets to *main* and is not caught, the program terminates.

Consider the example on the right and note the following(Code in *example\_first\_examples,* in *ExceptionInMethod* class):

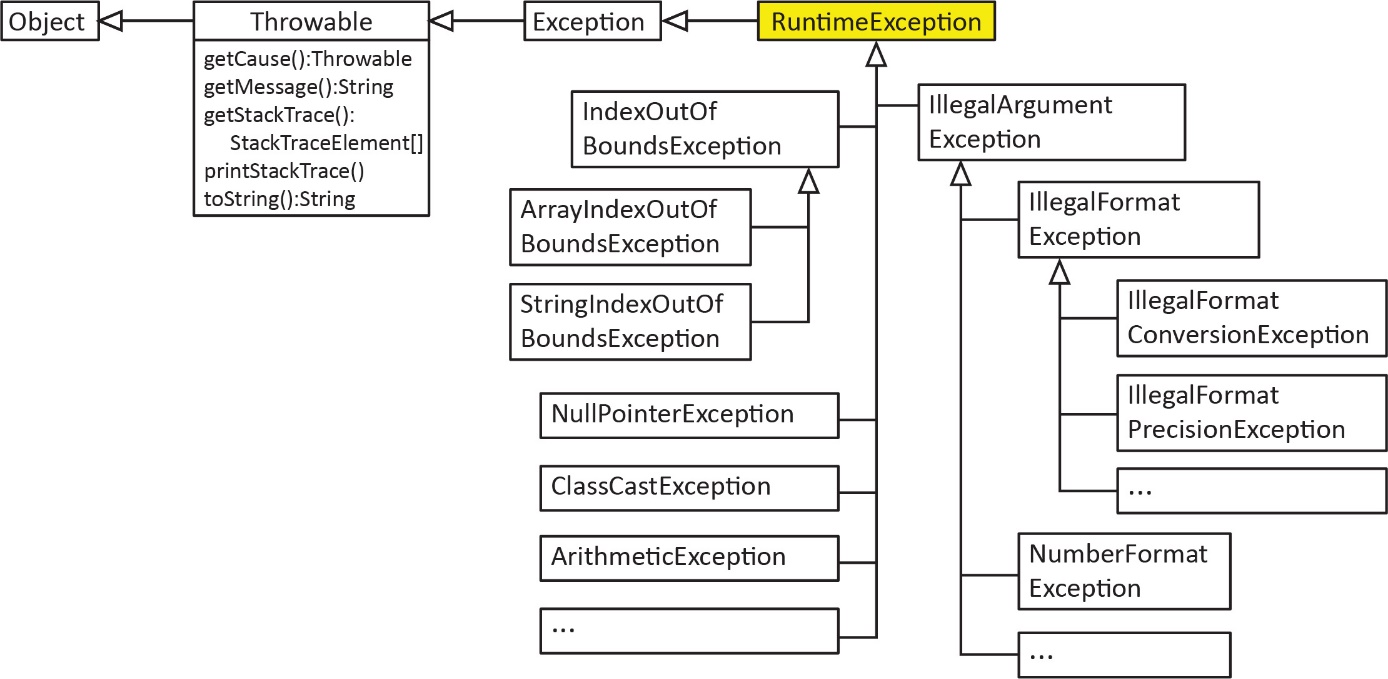
* Step 1 – *inverse* is called passing the value 0.
* Step 2 – An attempt to divide by zero is made, which generates an *ArithmeticException*.
* Step 3 – Since there is no *try/catch* block, the exception is thrown to the calling methed, *main*, where it is caught.
* Step 4 – The exception is printed.

The purpose of the *catch* block is to provide code to keep the code running, inform the user, send an error report, correct the error, *etc*. An explanation of such strategies is beyond the scope of the course. Some resources are provided at the end of this document. However, frequently, as we are learning about exception handling, we will simply print the exception in the *catch* block. This is illustrated in line 11 in the example above. When an exception is printed, the *exception class* name is displayed followed by *message*. In the example above:

java.lang.ArithmeticException: / by zero

# The *Exception* Class Hierarchy

Java defines an *Exception* hierarchy, a part of which is shown in the class diagram below. This hierarchy includes the *RuntimeException* subclass, which represents failures that result from programmer error. Subclasses of *RuntimeException* correspond to specific types of failures that can occur in the program.



The members of the *Throwable[[1]](#footnote-1)* class are inherited by all *RuntimeExceptions*. The methods are:

|  |  |
| --- | --- |
| **Method** | **Description** |
| *getCause():Throwable* | (Omit) Returns the cause of the exception in the situation where an exception is chained to another exception. See Appendix 7 for more information |
| *getMessage():String* | Returns the message associated with the exception, *e.g.* “/ zero”. |
| *getStackTrace():StackTraceElement[]* | (Omit) Contains the information in *printStackTrace* broken down into pieces that can be accessed by code. |
| *printStackTrace()* | Prints the stack trace to the console. |
| *toString():String* | Returns the name of the exception and the message. |

For example, from the most recent example above (Code in *example\_first\_examples,* in *ExceptionInMethod2* class):

e.getCause(): null

e.getMessage(): / by zero

e.toString(): java.lang.ArithmeticException: / by zero

e.getStackTrace(): first\_example.ExceptionExample2.inverse(ExceptionExample2.java:30)

: first\_example.ExceptionExample2.main(ExceptionExample2.java:9)

e.printStackTrace(): java.lang.ArithmeticException: / by zero

at first\_example.ExceptionExample2.inverse(ExceptionExample2.java:30)

at first\_example.ExceptionExample2.main(ExceptionExample2.java:9)

# The *catch* Block

The parameter for the *catch* block can be *Throwable* or any subclass. A *catch* block will catch a thrown exception whose class is the same as the class of the parameter in the *catch* block, or any subclass.

For example, the *catch* block below will catch any type of *Exception* (for example: *ArithmeticException, NullPointerException, etc.*)

**try** {

// code that may fail

}

**catch**(Exception e) {

// Code to execute when an exception is thrown

}

However, the *catch* block below will only catch an *ArithmeticException* (or any subclass of *ArithmeticException*)*.* If a *NullPointerException* is thrown, it will not be caughtbecause a *NullPointerException* is not a subclass of *ArithmeticException.*

**try** {

// code that may fail

}

**catch**(ArithmeticException e) {

// Code to execute when an exception is thrown

}

If we know our code might exhibit a specific type of failure, then we should try to catch the corresponding exception. Often, however, we are lazy and simply catch *RuntimeException,* because that will catch any programming error. The appendix has a section on [catching multiple exceptions](#Catching_Multiple_Exceptions).

## Exercises

1. Consider the *try/catch*  block below. Name the classes of exceptions that will be caught with this block. Hint: reference the *Exception* hierarchy above.

**try** {

// code that may fail

}

**catch**(IndexOutOfBoundsException e) {

|  |
| --- |
| **import** java.util.Random;  **public** **class** Foo {  **public** Foo() {}    **public** **int** eval() {  **int**[] vals = {4,22,9,49,18};  Random random = **new** Random();  **int** index = random.nextInt(10);  **return** vals[index];  }  **public** **static** **void** main(String[] args) {  }  } |

// Code to execute when an exception is thrown

}

1. (Solution in *example\_foo* package) Consider the class shown on the right. The *eval* method defines an array with 5 elements, and then generates a random[[2]](#footnote-2) integer between 0 and 9. Finally, the method attempts to return the array element at the index, which of course will fail if the index is 5 or greater.

Add code in *main* to create a *Foo* instance and call the *eval* method. They code should print the value returned if no exception is thrown. Otherwise, if an exception is thrown, it should be printed to the console.

# Anticipating & Handling Errors

Frequently, we use the terms: error, bug, defect, failure, system crash/bomb synonymously. However, it is useful to define the terms more precisely.

* A software developer makes an ***error*** (careless error, inappropriate decision) which causes a ***defect***in a software product.
* A ***defect***(bug) is a flaw in any aspect of the system that contributes, or may potentially contribute to the occurrence of one or more ***failures***.
* A ***failure*** (system crash/bomb, runtime error) is an unacceptable behaviour exhibited by a system.

All software systems have defects (bugs) and for robust software systems we must try to *handle* run-time errors so that the program does not stop unexpectedly. In general, there are two techniques:

1. Anticipate and trap sources of error *before they result in a run-time error.*
2. Detecting and handling run-time errors *after they occur (try/catch)*

If we can anticipate an error we should usually try to write code to prevent it. For example, we have seen when we have a getter for an item in an *ArrayList* we check to make sure the index is valid before attempting to return the object, as shown below:

**public** **class** Company {

**private** ArrayList<Employee> emps = **new** ArrayList<>();

**public** Employee getEmployee(**int** i) {

**if**(i<0 || i>=emps.size()) {

**return** **null**;

}

**return** emps.get(i);

}

...

}

It is hard to anticipate every single situation that could cause a runtime error. Hopefully it is clear, that even if we had an abundance of time, it would be nearly impossible to anticipate every possibly source of error. Thus, run-time errors are going to occur, so, we use *try/catch* wherever appropriate.

# Throwing Exceptions

The example in this this section is in the *example\_circle\_throws\_exception* package.

We sometimes find ourselves in a situation where we can anticipate and detect an error, but there is no code we can write to fix the situation. For example, suppose we have a *Circle* class that accepts a radius in the constructor. If the *radius* is required to be greater than 0, then obviously we can detect when it is not, but what should we do in that case? Arbitrarily set the *radius* to 1? But then how would the code that created the *Circle* know that this had happened?

**public** **class** Circle {

**private** **double** radius;

**public** Circle(**double** radius) {

**if**(radius < 0.0) {

// What should we do?

}

**this**.radius = radius;

}

...

}

A solution to the problem above is to have our code create an exception object and *throw* it. The *Exception* class and all its subclasses define a constructor that takes string argument, which is a description of what caused the exception (the *message*). Java defines the keyword *throw,* which allows the programmer to throw an exception. This is useful when we detect an error and we are not prepared to deal with it in that location in the code. Thus, we can create and throw Java exceptions with custom descriptions. For example:

**throw** **new** RuntimeException ("Description of error");

In the *Circle* example below, we throw an *IllegalArgumentException,* as shown below on the left, when the radius is 0 or less. On the right, in another class, we write code to create a *Circle*. If we do not surround that code in *try/catch*, then the program will terminate if the supplied radius is negative. To be safe, we surround that code with *try/catch* so that the program will continue running in the case of a negative radius.

|  |  |
| --- | --- |
| **Create and throw an Exception** | **Use Class Properly – Graceful Termination** |
| **public** **class** Circle {  **private** **double** radius;  **public** Circle(**double** radius) {  **if**(radius <= 0.0) {  **throw** **new** IllegalArgumentException  ("Radius must be greater than zero.");  }  **this**.radius = radius;  }  ...  } | Circle c;  **try**{  c = **new** Circle(-5.5);  }  **catch**(IllegalArgumentException e) {  System.***out***.println(e);  } |
| **Abnormal Termination** |
| c = **new** Circle(-5.5); |

Notes:

* *IllegalArgumentException* is a subclass of *RuntimeException* and this would be the natural, descriptive, choice of exception to throw for this example. However, we could have used *RuntimeException* or even *Exception* instead.
* In a real application, with a GUI where the user entered a radius, we would probably check to make sure it was valid before even calling the constructor, where we could inform the user and let them try again. Thus, we might not even use *try/catch* above. However, the throws in the constructor is there for an additional check.

## Exercises

1. (Solution in *exercise\_account* package) Consider the class below.The *mergeAccount* method accepts another *Account* object and adds its *balance* to this balance.

**public** **class** Account {

**private** **double** balance;

**private** String name;

**public** Account(**double** balance, String name) {

**this**.balance = balance;

**this**.name = name;

}

**public** **void** mergeAccount(Account a) {

**this**.balance += a.balance;

}

}

1. Modify the *mergeAccount* method so that it throws an *IlleagalArgumentException* if the account *name*s are not the same. Otherwise, if they are the same, the method should merge the accounts as shown.
2. Add a *main* that creates a few accounts and tries to merge them handling any exception that is thrown by printing the exception.
3. (Solution in *exercise\_engine* package, *EngineTest* class) Consider the class shown below.

**class** Engine {

**public** **void** init() {

**if**( Math.*random*() < 0.5 )

**throw** **new** RuntimeException();

**else**

System.***out***.println("init()");

}

**public** **void** run() {

System.***out***.println("run()");

}

**public** **void** close() {

System.***out***.println("close()");

}

}

Write a static method to implement an algorithm that creates an *Engine,* and then calls, *init, run,* and then *close* if no exception is thrown in *init.* Thus, if no exception is thrown, the method should display:

init()

run()

close()

However, if *init* does throw an exception then *run* should not be called, but *close* should be. The desired output if an exception is thrown is:

java.lang.RuntimeException

close()

1. (Solution in *exercise\_engine* package, *EngineTest\_MultipleChoice* class) Below are five attempts at writing the static method requested in the previous problem. For each attempt answer these questions:
2. Does the code compile? (Hint: only one does not)
3. If the code compiles,
4. What output does it produce when no exception is thrown?
5. What output does it produce when an exception is thrown?

|  |  |  |
| --- | --- | --- |
| Attempt 1 | Attempt 2 | Attempt 3 |
| Engine e = **new** Engine();    **try** {  e.init();  e.run();  }  **catch**( RuntimeException ex ) {  System.***out***.println(ex);  e.close();  }  e.close(); | Engine e = **new** Engine();    **try** {  e.init();  e.run();  e.close();  }  **catch**( RuntimeException ex ) {  System.***out***.println(ex);  } | **try** {  Engine e = **new** Engine();  e.init();  e.run();  }  **catch**( RuntimeException ex ) {  System.***out***.println(ex);  }  e.close(); |
| Attempt 4 | Attempt 5 |  |
| Engine e = **new** Engine();  **try** {  e.init();  e.run();  }  **catch**( RuntimeException ex ) {  System.***out***.println(ex);  }  e.close(); | Engine e = **new** Engine();  **try** {  e.init();  e.run();  }  **catch**( RuntimeException ex ) {  System.***out***.println(ex);  e.close();  } |  |

# The *finally* Clause

Java also allows us to add a *finally* block to a *try/catch*. The *finally* block of code is *always* executed no matter whether an exception was thrown or not (and even if an exception is thrown in the *catch* block).

**try** {

// Statements

}

**catch**( Exception e ) {

// Statements

}

**finally** {

// Statements

}

For example, suppose A, B, C, D, E are each a single statement of code and that A or C might throw an exception.

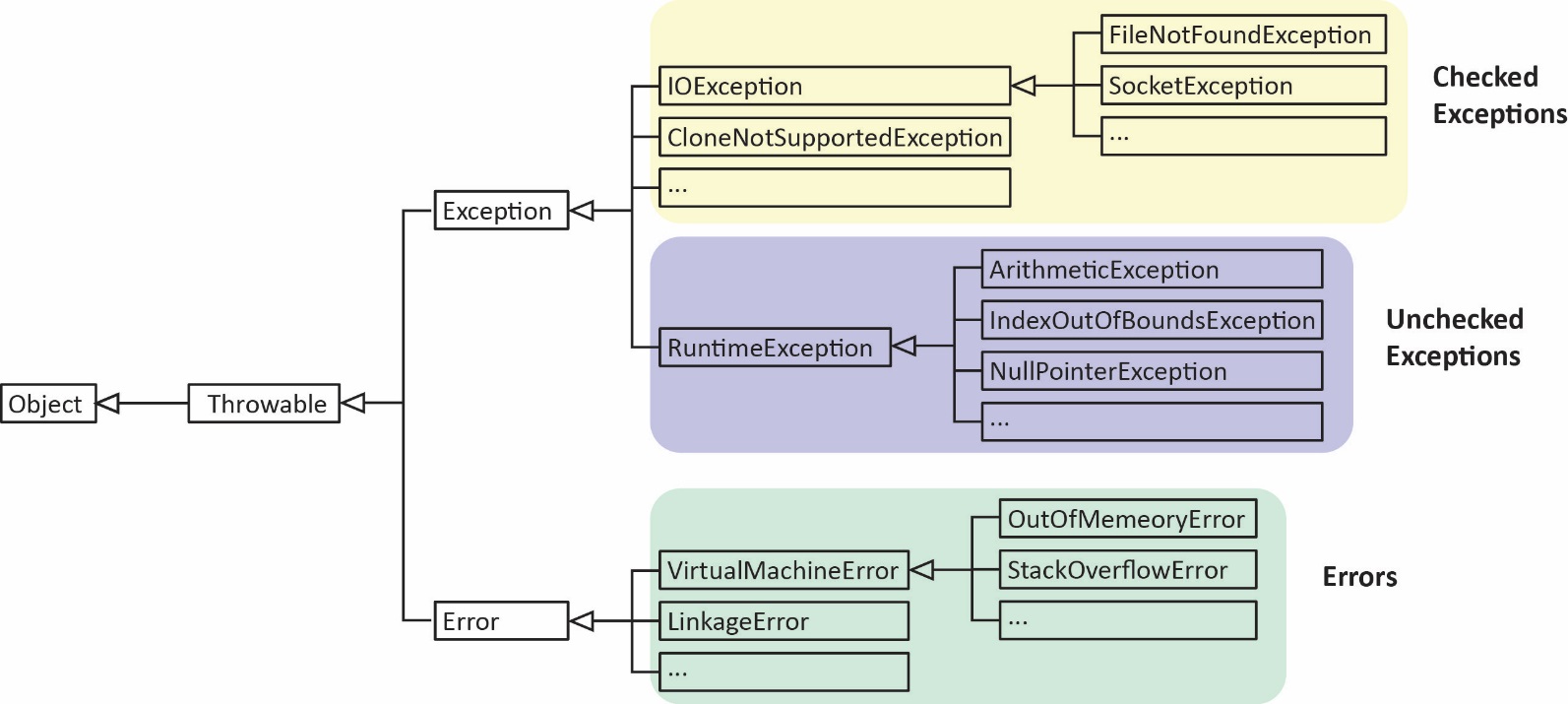
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Code** | | try {  A  B  }  catch(Exception e) {  C  }  finally {  D  }  E | | |  |  | | --- | --- | | **Scenario** | **Statements Executed Successfully** | | A doesn’t throw exception | A, B, D, E | | A throws exception | C, D, E | | A & C throw exceptions | D | |

As another example, consider the code below (Solution in *example\_try\_catch\_finally* package, *TryCatchFinally* class). What is the output?

|  |  |
| --- | --- |
| **No Exception Thrown** | **Exception Thrown** |
| Shows path of executions when no exception is thrown and try/catch/finally is used. | Shows path of executions when an exception is thrown and try/catch/finally is used. |
| Output: 1 2 4 5 | Output: 1 3 4 5 |

# Exception Types

The Exception hierarchy is a bit larger than shown earlier:



Note the following:

* The *Exception* class describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.
* *RuntimeException* and its subclasses is caused by programming errors, such as bad casting, trying to access an array element that doesn’t exist, numeric errors, *etc* as we have discussed before. These are known as *unchecked* *exceptions*. This means that these types of errors can and will occur, but you do not have to catch them, *e.g try/catch* is not required.
* The *Error* class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.
* **Checked Exceptions** – Any exceptions other than *RuntmeExceptions* are known as *checked exceptions*. A checked exception that might occur in a method must be caught (with *try/catch*) or the method must *declare the exception* using the *throws* keyword. For example, when you attempt to read from a text file it is possible that a *FileNotFoundException* (subclass of *IOException*, a checked exception) will be thrown. Thus, one of the two approaches below must be used. We will see more of this in the next chapter.

|  |  |
| --- | --- |
| ***try/catch*** | ***throws*** |
| **public** **static** **void** main(String[] args) {  **try** {  // Read file  }  **catch**( FileNotFoundException e ) {  System.***out***.println(e);  }  } | **public** **static** **void** main(String[] args)  **throws** FileNotFoundException {  // Read File  } |

Good reads: [Are checked exceptions good or bad?](https://www.infoworld.com/article/3142626/are-checked-exceptions-good-or-bad.html) and [Unchecked Exceptions — The Controversy](https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html)

# A Real Example

Below, are two methods from a real system[[3]](#footnote-3). The first method, *openConnection,* is called to open a connection to a server and establish input and output communication (socket) channels. It utilizes *try/catch* and a *try/catch* within the *catch* block. It also calls a method, *closeAll,* which utilizes *try/catch*.

**public** **final** **void** openConnection() **throws** IOException {

// Do not do anything if the connection is already open

**if**(isConnected()) **return**;

//Create the sockets and the data streams

**try** {

clientSocket= **new** Socket(host, port);

output = **new** ObjectOutputStream(clientSocket.getOutputStream());

input = **new** ObjectInputStream(clientSocket.getInputStream());

}

**catch** (IOException ex) {

// All three of the above must be closed when there is a failure to create any of them

**try** {

closeAll();

}

**catch** (Exception exc) { }

**throw** ex; // Rethrow the exception.

}

clientReader = **new** Thread(**this**); //Create the data reader thread

readyToStop = **false**;

clientReader.start(); //Start the thread

}

**private** **void** closeAll() **throws** IOException {

**try** {

//Close the socket

**if** (clientSocket != **null**) clientSocket.close();

//Close the output stream

**if** (output != **null**) output.close();

//Close the input stream

**if** (input != **null**) input.close();

}

**catch** (Exception exc) {

**throw** exc;

}

**finally** {

// Set the streams and the sockets to NULL no matter what. Doing so allows,

// but does not require, any finalizers of these objects to reclaim system

// resources if and when they are garbage collected.

output = **null**;

input = **null**;

clientSocket = **null**;

}

}

Appendix

1. Resources

Designing robust error handling for a system is beyond the scope of this course. Here, we mostly focus on the techniques themselves. Some references on designing robust error handling:

<http://codebuild.blogspot.com/2012/01/15-best-practices-about-exception.html>

<https://stackoverflow.com/questions/4589750/exception-handling-pattern>

<https://stackoverflow.com/questions/425281/java-style-properly-handling-exceptions?rq=1>

Some references on throwing and handling exceptions:

<https://docs.oracle.com/javase/tutorial/essential/exceptions/handling.html>

<https://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html>

<http://mindprod.com/jgloss/exception.html>

1. Catching Multiple Exceptions

The example in this section is in the *example\_multiple\_catch* package.

**Catching Multiple Exceptions** – In general, you can catch multiple types of exceptions by supplying multiple *catch* blocks.

try {

statements; // Statements that may throw exceptions

}

catch (Exception1 e1) {

code to handle Exception1;

}

catch (Exception2 e2) {

code to handle Exception2;

}

...

catch (ExceptionN eN) {

code to handle ExceptionN;

}

Note the following:

* The compiler forces you to arrange the *catch* blocks from most specific to most general. In other words, subclasses must be caught before superclasses. In the example below *ArithmeticException* is a subclass of *Exception* so it must be listed as the first catch block.

|  |  |
| --- | --- |
| **Valid** | **Invalid (Doesn’t Compile)** |
| **try** {  ...  }  **catch** ( ArithmeticException e ) {  ...  }  **catch** ( Exception e ) {  ...  } | **try** {  ...  }  **catch** ( Exception e ) {  ...  }  **catch** ( ArithmeticException e ) {  ...  } |

* Only one catch block will activate, the first one that matches. In the example above on the left, an *ArithmeticException* would be caught in the first *catch* block even though it is also an *Exception*. An *IndexOutOfBounds* exception would be caught in the second *catch block.*
* If no *catch* block is found that matches the argument type, then the exception is passed (thrown) to the calling method and this process repeats, *i.e.* searches for a matching *catch* block. If no *catch* block is found through the entire chain of method calls, the program terminates and a message is printed in the console.
  1. Exercises

1. (Solution not available) Suppose you are trying to call a method, *foo()* which can throw any of these types of exceptions: *IlleagalArgumentException*, *IllegalFormatException, IllegalFormatCoversionException*. Write a snippet of code that tries to call *foo* and catches any of these exceptions. Hint: see Exception hierarchy presented earlier.
2. Rethrowing Exceptions

We can *rethrow* an exception that is caught in a *catch* block. We do this if we want the caller to handle the situation. For example:

**try** {

...

}

**catch**(RuntimeException e) {

**throw**(e);

}

In the examples below, suppose A, B, C, D, E are each a single statement of code and that A might throw an exception.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Code** | | try{  A  B  }  catch(Exception e) {  C  throw(e) // or return  }  finally{  D  }  E | | |  |  | | --- | --- | | **Scenario** | **Statements Executed Successfully** | | A doesn’t throw exception | A, B, D, E | | A throws exception | C, D, throw(e) | |

Consider the code shown (code not available) on the right below which calls a method to divide to number. The explanation of the numbered steps is shown on the left.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Step | Description | | 1 | *divide* is called on line 7 | | 2 | Attempt to divide by zero on line 16, throws an exception which is caught on line 19 | | 3 | Before the exception is thrown on line 20, the *finally* block is executed | | 4 | Then, the *catch* blockthrows the exception which is caught by *main* on line 10. | | E:\Data-Classes\CS 1302 - Programming 2-Fall 2016\notes\06_ch12_Exceptions\d5.jpg |

The two examples below (code in *example\_rethrow* package) are identical except that the one on the left does not have a *finally* block while the one on the right does. The arrows on both examples show what is output when *myMethod(4,0)* is executed. What is output when *myMethod(4,3)* is executed for each example? Answers are in a footnote[[4]](#footnote-4).

|  |  |
| --- | --- |
| *myMethod(4,0)* 🡺 1 3 6  *myMethod(4,3)* 🡺? | *myMethod(4,0)*: 1 3 4 6  *myMethod(4,3)*:🡺? |
| E:\Data-Classes\CS 1302 - Programming 2-Spring 17\notes\06_ch12_Exceptions\f3.jpg | E:\Data-Classes\CS 1302 - Programming 2-Spring 17\notes\06_ch12_Exceptions\f2.jpg |

* 1. Exercises

1. (Solution in foot note below) Consider the following:

* *A* and *B* are subclasses of *RuntimeException*. *C* is a subclass of *B.*
* *Goo()* is a method (not shown) that can throw an *A, B,* or *C* exception.
* Consider *main* and *foo()* shown below.:

|  |  |
| --- | --- |
| **public** **static** **void** main(String[] args) {  **try** {  *foo*();  System.***out***.println("5");  }  **catch**(C c) {  System.***out***.println("6");  }  **catch**(RuntimeException e) {  System.***out***.println("7");  }  } | **public** **static** **void** foo() {  **try** {  *goo*();  System.***out***.println("1");  }  **catch**(C c) {  System.***out***.println("2");  **throw**(c);  }  **catch**(B b) {  System.***out***.println("3");  }  **finally** {  System.***out***.println("4");  }  } |

What is the output when[[5]](#footnote-5)…

1. No exception occurs?
2. A *B* exception occurs?
3. A *C* exception occurs?
4. An *A* exception occurs?
5. (Solution in *exercise\_too* package) Consider the classes shown on the right. The *sum* method sums the values in the input array in positons 0 through *numVals-1.* However, the method can cause a run-time error if *numVals* is too big or small. You should modify this method to try the computation and if an exception is thrown, it is caught and rethrown to main. You should also modify main to catch this exception should it be thrown and print it.

**public** **class** Problem3 {

**public** **static** **void** main(String[] args) {

**int** vals[] = **new** **int**[5];

vals[0]=2; vals[1]=6; vals[2]=8;

Too too = **new** Too();

**int** sum = too.sum(vals, 7);

System.***out***.println(sum);

}

}

**class** Too {

**public** **int** sum(**int**[] vals, **int** numVals) {

**int** sum = 0;

**for**(**int** i=0; i<numVals; i++) {

sum += vals[i];

}

**return** sum;

}

}

1. Chained Exceptions

**Chaining Exceptions** *–* Sometimes, we want to *chain* exceptions together. Here is how this works:

1. Catch an exception
2. Create a **new** exception
3. Link the original (caught) exception to the new one
4. Throw the new one.

We call this *exception chaining*. The Exception class has another constructor that takes two arguments: (1) a string description and (2) a reference to another exception. It also provides the *getCause* method which returns a chained exception.

**Example** (code in *example\_chained* package) – Chained exceptions

public class ChainedTester1 {

public static void main(String[] args) {

try {

A a = new A();

a.m1();

}

catch( Exception e ) {

System.out.println( e + "\n" + e.getCause() + "\n" );

}

}

}

class A {

public void m1() throws Exception {

try {

B b = new B();

b.m2();

}

catch (Exception e) {

throw new Exception("A.m1() caused an exception", e);

}

}

}

class B {

public void m2() throws Exception {

throw new Exception("B.m2() caused an exception");

}

}

Output:

java.lang.Exception: A.m1() caused an exception

java.lang.Exception: B.m2() caused an exception

* 1. Exercises

1. (Solution in *exercise\_roo* package) Consider the classes shown on the right. The *sum* method sums the values in the input array in positons 0 through *numVals-1.* However, the method can cause a run-time error if *numVals* is too big or small. You should modify this method to try the computation and if an exception is thrown create a new exception (with a description) chaining the original exception to it, and rethrow it to main. You should also modify main to catch this exception should it be thrown and print it, and the chained exception.

**public** **class** Problem4 {

**public** **static** **void** main(String[] args) {

**int** vals[] = **new** **int**[5];

vals[0]=2; vals[1]=6; vals[2]=8;

Roo roo = **new** Roo();

**int** sum = roo.sum(vals, 7);

System.***out***.println(sum);

}

}

**class** Roo {

**public** **int** sum(**int**[] vals, **int** numVals) {

**int** sum = 0;

**for**(**int** i=0; i<numVals; i++) {

sum += vals[i];

}

**return** sum;

}

}

1. Summary of Error Handling Approaches

There are two approaches to error handling:

1. Anticipate and trap sources of error *before they result in a run-time error.* There are two ways to handle this:

|  |  |
| --- | --- |
| Fix the situation | Throw a new exception |
| if( detect error )  fix it, etc | if( detect error )  throw new Exception(“desc”); |

1. Handling run-time errors *after they occur.* There are four ways to handle this:

|  |  |
| --- | --- |
| Fix the situation | Throw a new exception |
| try {  something  }  catch(Exception e) {  fix it, etc  } | try {  something  }  catch(Exception e) {  throw new Exception(“desc”)  } |
| Rethrow caught exception | Throw a new exception, chaining caught exception |
| try {  something  }  catch(Exception e) {  throw e  } | try {  something  }  catch(Exception e) {  throw new Exception(“desc”,e)  } |

1. Custom Exceptions

The example in this section is in the *example\_custom\_exception\_mymathexception* package.

**Custom Exceptions** – In some situations, it is useful to write our own customized exception classes. These are classes just like any other class except that they extend *Exception* or its subclasses). Thus, we can provide properties and methods that contain information about the error that might be useful to code that catches the exception.

Example – Consider a method in a class that divides two numbers which obviously can throw an exception if the value of *y* is 0:

class A {  
 public int m1( int x, int y ) {  
 return x/y;  
 }  
}

We would like to detect this situation and if it occurs, throw a custom exception, MyMathException that contains a description of the error and the values of x and y.

class A {  
 public int m1( int x, int y ) throws MyMathException {  
 if( y==0 )   
 throw new MyMathException( "Can't divide by zero", x, y );  
 return x/y;  
 }  
}

Thus, we define a *MyMathException* class that extends *RuntimeException*. The constructor accepts a description of the error and two integers. We also provide getters so the code that calls *m1* above and catches an exception can obtain the values of *x* and *y*.

class MyMathException extends RuntimeException {

private int x;

private int y;  
   
public MyMathException( String msg, int x, int y ) {

super(msg);

this.x = x;

this.y = y;

}

public int getX() { return x; }

public int getY() { return y; }

}

Finally, we can call the method and catch the exception, displaying the values of *x* and *y.*

public class ExceptionTester9 {  
 public static void main(String[] args) throws Exception {  
 A a = new A();  
 try {  
 System.out.println(a.m1(4,0));  
 }  
 catch( MyMathException me ) {  
 System.out.println(me);   
 System.out.println("x=" + me.getX() + ", y=" + me.getY() );   
 }  
 catch( Exception e ) {  
 System.out.println(e);  
 }  
 }

Output:

MyMathException: Can't divide by zero  
x=4, y=0

* 1. Exercises

1. (Solution in *exercise\_blob\_custom\_exception* package) Consider the class shown below. We want to modify the *merge* method so that it only merges *Blobs* if their *power* is the same. If their power is not the same, then you should throw a custom *BlobMergeException.* This exception should contain a description of the error and the *power* value for each of the two *Blobs.*
2. Write the *BlobMergeException*
3. Modify the *merge* method to utilize the exception as described above.
4. Modify *main* to catch any *BlobMergeException* that may be thrown, displaying the description and the two *power* values.

**public** **class** Blob {

**private** **int** power;

**private** **int** health;

**public** Blob(**int** power, **int** health) {

**this**.power = power;

**this**.health = health;

}

**public** **int** getPower() {

**return** power;

}

**public** **int** getHealth() {

**return** health;

}

**public** **void** merge(Blob blob) {

**this**.health += blob.health;

}

**public** String toString(){

**return** String.*format*("power=%d, health=%d", power, health);

}

**public** **static** **void** main(String[] args) {

Blob b1 = **new** Blob(5,10);

Blob b2 = **new** Blob(4,8);

b1.merge(b2);

System.***out***.println(b1);

}

}

1. Checked Exceptions in the API

The Scanner constructor, *Scanner(File source)* can throw a checked exception as shown by the “throws” clause in the signature of method:

<https://docs.oracle.com/javase/9/docs/api/java/util/Scanner.html>

This means that you must either try/catch when creating a Scanner, or add throws to the method you wrote that uses Scanner.

The Scanners, nextDouble (and lots of other *next* methods) can throw a number of exceptions

<https://docs.oracle.com/javase/9/docs/api/java/util/Scanner.html>

Notice that there is NOT a throws in the signature, but below the description of method, there is a section labelled “Throws:”. These are unchecked exceptions. You can easily see this by clicking on one of the exceptions. For example, click on the second one, “NoSuchElementException”,

<https://docs.oracle.com/javase/9/docs/api/java/util/NoSuchElementException.html>

and you can see that it is a subclass of RuntimeException (meaning that it is unchecked)

1. <https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html> [↑](#footnote-ref-1)
2. <https://docs.oracle.com/javase/8/docs/api/java/util/Random.html> [↑](#footnote-ref-2)
3. These methods are a part of the Object-Oriented Client-Server Framework (OCSF) which accompanies the text, *Object-Oriented Software Engineering*, by Lethbridge & Laganiere. [↑](#footnote-ref-3)
4. Left example: 1 2 5, Right example: 1 2 4 5 [↑](#footnote-ref-4)
5. (a) 1 4 5 (b) 3 4 5 (c) 2 4 6 (d) 4 7 [↑](#footnote-ref-5)