**Exception Handling and Testing in Java**

**-Warning(warning shown, does run):** Creating variables but not using them.

**-Error:** Syntax error, using variable without declaration.

**-Compile time errors:**

**-Syntax error(error shown, doesnt run):** Wrong syntax. They are also called compiler errors since compiler show them during compliation phase. Also called compiler error, compile-time error, compliation error.

Using variable without declaration.

Forgetting a “;”

When one brace of a block is mising.

**-Semantic error (error shown, doesnt run):** Things that are not meaningful to complier.

b + c = a;

**-Run time(execution time) error/ Logic error:**

**-Design error/Logic error (error not shown, does run):** Produces wrong output. Cant be detected by complier.

Using “(a + b) / 2” instead of “a + b / 2”

Not using both braces of a block might cause problems like “Dangling-else Problem”.

**-Fatal error/Fatal logic error (error not shown, does run and crash):** No error while compilation. Error happens while the program is running. Infinite loops, dividing by zero, forgetting ampersand while using function scanf, trying to open a file that wasnt created, lack of memory, input isnt in wanted format or input data file wasnt found at specified location, hardware errors,

i = 0;

array[--i] = a;

Warnings, errors in C: <http://www.c4learn.com/c-programming/c-pragma-warn-macro-directive/>

**-Exception Handling**

All errors and exceptions inherit from the Throwable class. Only classes that extend Throwable (package java.lang) directly or indirectly can be used with exception handling.

No exception handling:

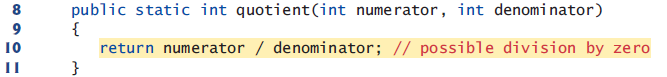
Exceptions are thrown (i.e., the exception occurs) by a method when it detecs a problem and is unable to handle it such as dividing by zero or reading a string when expecting int. JVM throws an object of the exception that occured.

Several lines of information are displayed in response to invalid input(divide by zero, enter string instead of int). This information is known as a stack trace, which includes the name of the exception (java.lang.ArithmeticException) in a descriptive message that indicates the problem that occurred and the method-call stack (i.e., the call chain) at the time it occurred. The stack trace includes the path of execution that led to the exception method by method. This helps you debug the program.

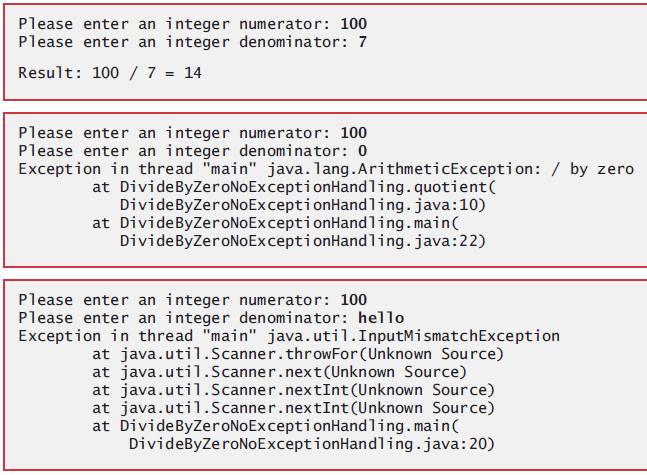
Each line of the stack trace contains the class name and method (e.g., DivideByZeroNoExceptionHandling.main) followed by the filename and line number (e.g., DivideByZeroNoExceptionHadling.java:22).

The top row of the call chain indicates the throw point—the initial point at which the exception occured.

Sometimes a program may continue even though an exception has occured and a stack trace has been printed, In such cases, the application may produce unexpected results. For example, a graphical user interface (GUI) application will often continue executing.



2



With exception handling:

Try block encloses the code that might throw an exception and the code that should not execute if an exception occurs (i.e., if an exception occurs, the remaining code in the try block will be skipped.)

Catch block (also called catch clause or exception handler) catches (i.e., receives) and handles an exception. Each catch block specifies in parantheses an exception parameter that identifies the exception type the handler can process.

In cases such as “InputMismatchException” where input of user wasn’t succesfully read, you should discard the bad input using “input.nextLine();”.

At least one catch block or a finally block must immediately follow the try block. When an exception occurs in a try block, the catch block that executes first is the one whose type matches the type of the exception that occured (i.e., the type in the catch block matches the thrown exception type exactly or is a direct or indirect superclass of it.)

(streams part 3) We use the System.err (standart error stream) object to output error messages. By default, System.err’s print methods, like those of System.out, display data to the command prompt. Use System.err for debugging but not for logging.

System.out and System.err are streams—sequences of bytes. While System.out

(known as the standard output stream) displays a program’s output, System.err (known

as the standard error stream) displays a program’s errors. Output from these streams can

be redirected (i.e., sent to somewhere other than the command prompt, such as to a file).

Using two different streams enables you to easily separate error messages from other

output.

<https://stackoverflow.com/questions/1049795/whats-wrong-with-using-system-err-in-java>

It’s relatively common for a try block to be followed by several catch blocks to handle various types of exceptions. If the bodies of several catch blocks are identical, you can use the multi-catch feature. (tag: multi catch)

catch (*Type1* | *Type2* | *Type3* e)

Java uses a “multithreaded” model of program execution—each thread is a concurrent activity. One program can have many threads. If a program has only one thread, an uncaught exception will cause the program to terminate. If a program has multiple threads, an uncaught exception will terminate only the thread in which the exception occurred. In such programs, however, certain threads may rely on others, and if one thread terminates due to an uncaught exception, there may be adverse effects on the rest of the program.

If an exception occurs in a try block the try block terminates immediately and program control transfers to the first of the following catch blocks in which the exception parameter’s type matches the thrown exception’s type.

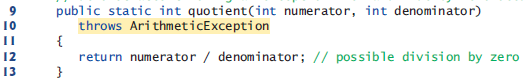
After the exception is handled, program control does not return to the throw point, because the try block has expired (and its local variables have been lost). Rather, control resumes after the last catch block. This is known as the termination model of exception handling. Some languages use the resumption model of exception handling, in which, after an exception is handled, control resumes just after the throw point.

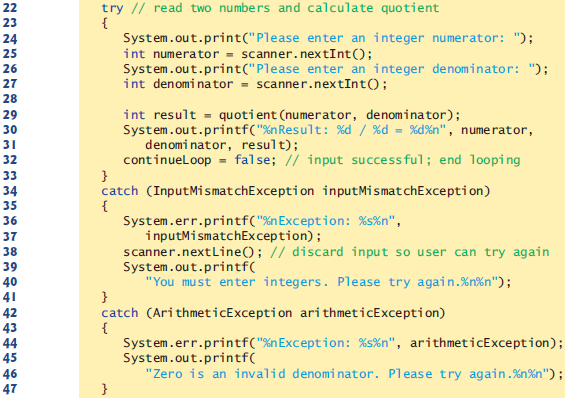
If no exceptions are thrown in the try block, the catch blocks are skipped and control continues with the first statement after the catch blocks.

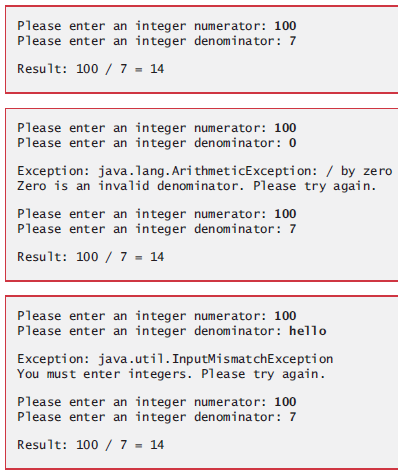
Throws clause specifies the exceptions the method might throw if problems occur. This clause, which must appear after the method’s parameter list and before the body, contains a comma-separated list of the exception types. Such exceptions may be thrown by statements in the method’s body or by methods called from there.

Read the online API documentation for a method before using it in a program. The documentation specifies the exceptions thrown by the method (if any) and indicates reasons why such exceptions may occur. Next, read the online API documentation for the specified exception classes. The documentation for an exception class typically contains potential reasons that such exceptions occur. Finally, provide for handling those exceptions in your program.

When a method throws an exception, method terminates and does not return a value, and method’s local variables go out of scope (and are destroyed). If method contained local variables that were references to objects and there were no other references to those objects, the objects would be marked for garbage collection.







-When to use exception handling:

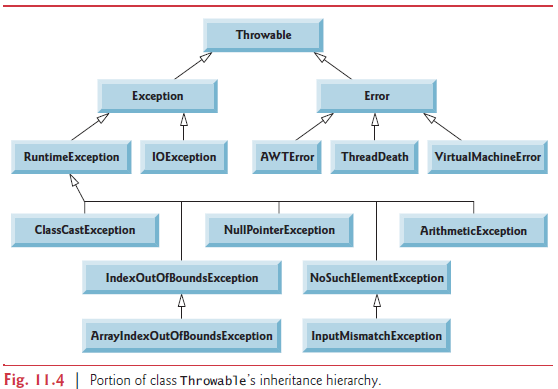
Exception handling is designed to process synchronous errors, which occur when a statement executes. Exception handling is not designed to process problems associated with asynchronous events (e.g., disk I/O completions, network message arrivals mouse clicks and keystrokes), which occur in parallel with, and independent of, the program’s flow of control.

Exception handling provides a single, uniform technique for documenting, detecting and

recovering from errors. This helps programmers working on large projects understand each other’s error-processing code.

-Java exception hierarchy:

Class Exception and its subclasses represent exceptional situations that can occur in a Java program and that can be caught by the application. Class Error and its subclasses represent abnormal situations that happen in the JVM. Most Errors happen infrequently and should not be caught by applications—it’s usually not possible for applications to recover from Errors.



<https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html>

<https://www.tutorialspoint.com/java/java_builtin_exceptions.htm>

All exception types that are direct or indirect subclasses of RuntimeException (package

java.lang) are unchecked exceptions. These are typically caused by defects in your program’s code. Classes that inherit directly or indirectly from class Error (Fig. 11.4) are unchecked, because Errors are such serious problems that your program should not even attempt to deal with them.

Unlike checked exceptions, the Java compiler does not examine the code to determine

whether an unchecked exception is caught or declared. Unchecked exceptions typically

can be prevented by proper coding. For example, the unchecked ArithmeticException

thrown by method quotient (lines 9–13) in Fig. 11.3 can be avoided if the method ensures

that the denominator is not zero before performing the division.

For unchecked exceptions, you dont have to declare throws statements for methods and you dont have to catch the exceptions. The application will compile, but it will run with unexpected results. Still, it is recommended that you do handle unchecked exceptions.

All classes that inherit from class Exception but not directly or indirectly from class RuntimeException are considered to be checked exceptions. Such exceptions are typically caused by conditions that are not under the control of the program—for example, in file processing, the program can’t open a file if it does not exist.

The compiler checks each method call and method declaration to determine whether the

method throws a checked exception. If so, the compiler verifies that the checked exception

is caught or is declared in a throws clause—this is known as the catch-or-declare requirement.

For unchecked exceptions, you do have to declare throws statements for methods and you do have to catch the exceptions. The application wont compile otherwise.

If your method calls other methods that throw exceptions, those exceptions must

be caught or declared. If an exception can be handled meaningfully in a method, the

method should catch the exception rather than declare it.

-Finally block (finally clause):

The finally block will execute whether or not an exception is thrown in the corresponding

try block. The finally block also will execute if a try block exits by using a return,

break or continue statement or simply by reaching its closing right brace. The one case

in which the finally block will not execute is if the application exits early from a try block

by calling method System.exit.

If an exception that occurs in a try block cannot be caught by one of that try block’s

catch handlers, the program skips the rest of the try block and control proceeds to the

finally block. Then the program passes the exception to the next outer try block—normally

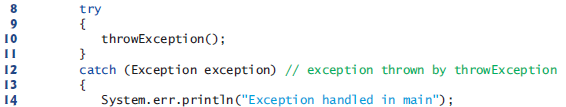
in the calling method—where an associated catch block might catch it. This process

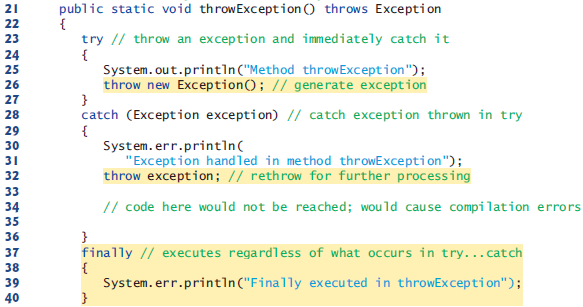
can occur through many levels of try blocks. Also, the exception could go uncaught.

If a catch block throws an exception, the finally block still executes. Then the

exception is passed to the next outer try block—again, normally in the calling method.

Because a finally block always executes, it typically contains resource-release code. Resources such as objects, files, database connections and network connections.







You can throw exceptions yourself by using the throw statement.

Throw exceptions from constructors to indicate that the constructor parameters are not valid—this prevents an object from being created in an invalid state.

You can explicitly throw an exception after catching it to send it to upper scope to be handled. This is called rethrowing the exception.

-Stack unwinding and ontaining information from an exception object:

When an exception is thrown but not caught in a particular scope, the method-call stack is “unwound,” and an attempt is made to catch the exception in the next outer try block.

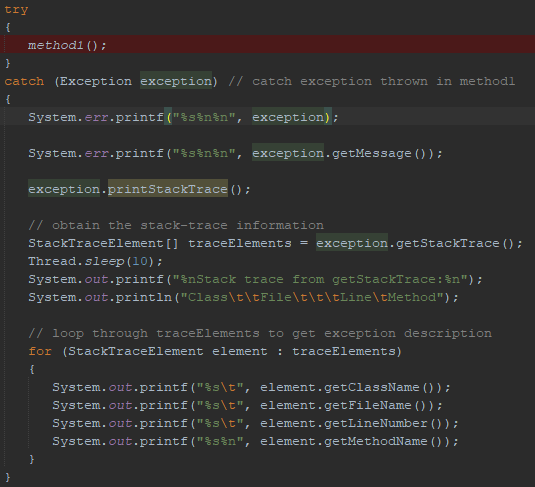
This process is called stack unwinding. Unwinding the method-call stack means that the

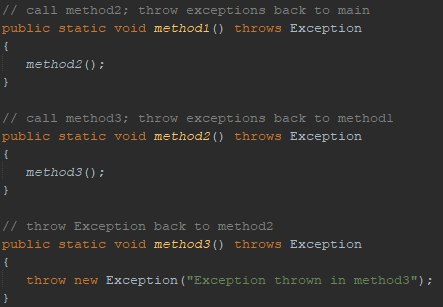
method in which the exception was not caught terminates, all local variables in that method go out of scope and control returns to the statement that originally invoked that method. If a try block encloses that statement, an attempt is made to catch the exception. If a try block does not enclose that statement or if the exception is not caught, stack unwinding occurs again.

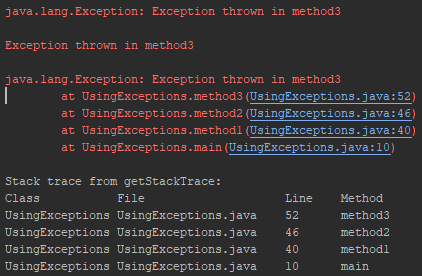
When toString is invoked on any Throwable object, its resulting String includes class name and the descriptive string that was supplied to the constructor, or simply the class name if no string was supplied.

Class Throwable’s getMessage method returns the descriptive string stored in an exception object.

Class Throwable has a printStackTrace method that outputs to the standard error stream the stack trace. Often this is helpful in testing and debugging. Class Throwable also provides a getStackTrace method that retrieves the stack-trace information that might be printed by printStackTrace.

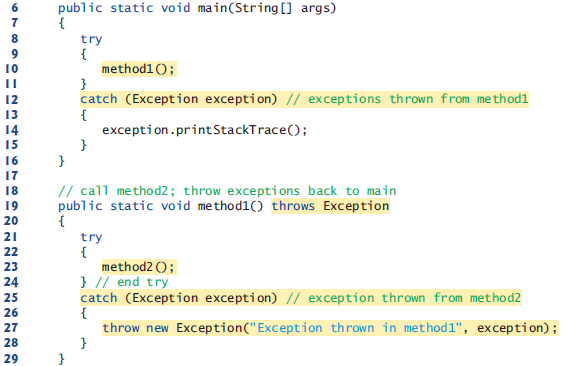


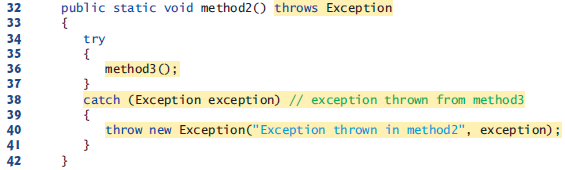


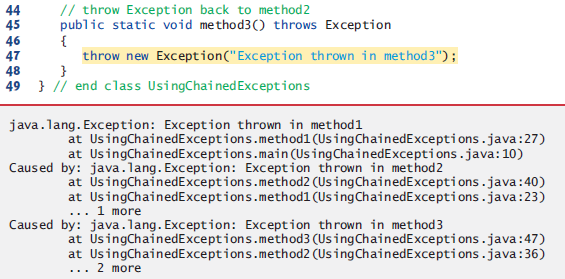


-Chained exceptions:

When we did exception rethrowing, we did partial processing and threw the same exception object. Sometimes a method responds to an exception by throwing a different exception type that’s specific to the current application. If a catch block throws a new exception, the original exception’s information and stack trace are lost. Earlier Java versions provided no mechanism to wrap the original exception information with the new exception’s information to provide a complete stack trace showing where the original problem occurred. This made debugging such problems particularly difficult. Chained exceptions enable an exception object to maintain the complete stack-trace information from the original exception.







-Declaring new exception types:

A typical new exception class contains only four constructors:

1) one that takes no arguments and passes a default error message String to the superclass

constructor

2) one that receives a customized error message as a String and passes it to the superclass

constructor

3) one that receives a customized error message as a String and a Throwable (for

chaining exceptions) and passes both to the superclass constructor

4) one that receives a Throwable (for chaining exceptions) and passes it to the superclass

constructor.

When defining your own exception type, study the existing exception classes in the Java API and try to extend a related exception class. If the existing classes are not appropriate superclasses for your new exception class, decide whether your new class should be a checked or an unchecked exception class.

By convention, all exception-class names should end with the word Exception.

-Preconditions and postconditions:

Programmers spend significant amounts of time maintaining and debugging code. To facilitate these tasks and to improve the overall design, you can specify the expected states before and after a method’s execution. These states are called preconditions and postconditions, respectively.

A precondition must be true when a method is invoked. Preconditions describe constraints on method parameters and any other expectations the method has about the current state of a program just before it begins executing. If the preconditions are not met, then the method’s behavior is undefined

A postcondition is true after the method successfully returns. Postconditions describe constraints on the return value and any other side effects the method may have.

Typically, a method’s preconditions and postconditions are described as part of its

specification. When designing your own methods, you should state the preconditions and

postconditions in a comment before the method declaration.

-Assertions:

When implementing and debugging a class, it’s sometimes useful to state conditions that

should be true at a particular point in a method. These conditions, called assertions, help

ensure a program’s validity by catching potential bugs and identifying possible logic errors

during development. Preconditions and postconditions are two types of assertions.

Java includes two versions of the assert statement for validating assertions programatically.

The assert statement evaluates a boolean expression and, if false, throws an

AssertionError (a subclass of Error). The first form of the assert statement is

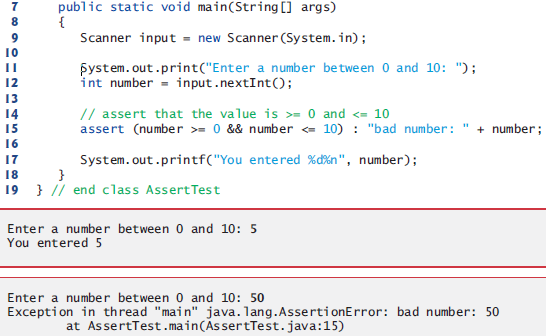
assert *expression*;

which throws an AssertionError if expression is false. The second form is

assert *expression1* : *expression2*;

which evaluates expression1 and throws an AssertionError with expression2 as the error

message if expression1 is false.



You must explicitly enable assertions when executing a program, because they reduce performance and are unnecessary for the program’s user. To do so, use the java command’s -ea command-line option, as in

Java -ea AssertTest

Use assertions only during development, not in production code. This can be debugging, identifying logic errors and testing. When we are explicity testing, we will use Junit. Dont catch AssertionErrors. Allow the program to terminate, see the error message, then locate and fix the source of the problem.

-Try-with-resources: automatic resource deallocation:

An alternative notation to releasing resources in finally block is the try-with-resources statement (introduced in Java SE 7). Each resource must be an object of a class that implements the Auto-Closeable interface, and thus provides a close method.

try (*ClassName* theObject = new *ClassName*()) {

// use theObject here

}

catch (Exception e) {

// catch exceptions that occur while using the resource

}

The try-with-resources statement implicitly calls the theObject’s close method at the end of the try block. You can allocate multiple resources in the parentheses following try by separating them with a semicolon (;).

In testing we want to have test code seperate from actual code. We can achieve that using Junit.