Chapter 7: Exceptions, Assertions, and Logging

-Notify the user of an error-Save all work-Allow user to exit the program

# 7.1 Dealing with Errors

-**Exception handling** transfers control from where the error occurred to an **error handle**.

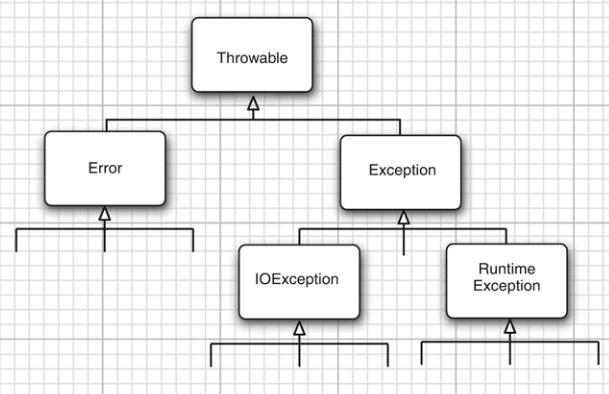
-Type of **problems**: user input errors, device errors, physical limitations, code errors

-The traditional reaction to error in method is to return a special error code: return -1, null

=>The method can throw an **object** that **encapsulates** the **error information**. The exception-handling mechanism search for an **exception handler***.*

## 7.1.1 The Classification of Exceptions

-An exception object is always an instance of class derived from **Throwable**-> **Error** and **Exception**. **Exception** includes ones that derive from **RuntimeException** and those that do not.



-**RuntimeException** happens because of a **programming error**. Such as: bad cast, out-of-bounds array access, null pointer access. **“If it is a RuntimeException, it was your fault”**.

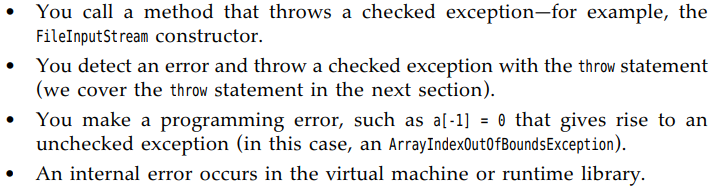
- Exception that derives from class Error or RuntimeException an **unchecked**exception. All other exceptions are **checked** exceptions. The compiler checks that you provide exception handlers for all checked exceptions.

## 7.1.2 Declaring Checked Exception

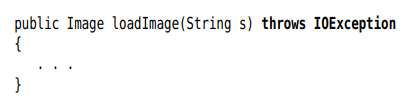
-A **method** will not only tell the compiler what values it can return, it is *also going to* ***tell the compiler what can go wrong****.* Example: Code that attempts to read from a file knows that the file might not exit or empty. It can throw IOException.

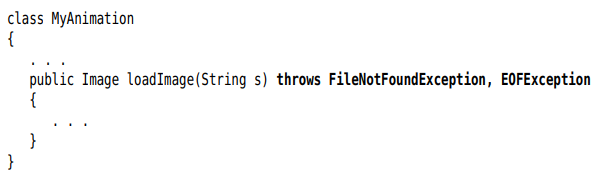


-An exception is thrown in any of 4 situations:

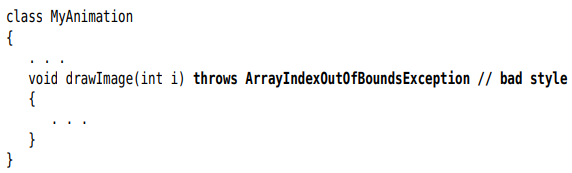


-The method may throw an exception with an **exception specification**in method header:





-Do not need to advertise **internal Java errors** – exceptions inheriting from Error and RuntimeException. These errors are under your control.



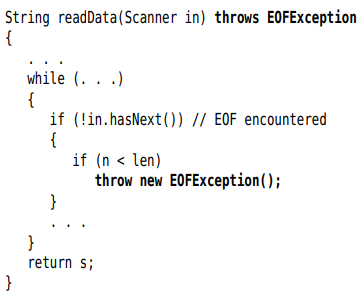
-**Summary**: method must declare all checked exceptions. Unchecked exceptions are beyond your control

-If you override a method from superclass, the checked exceptions that the subclass method declares **cannot** be more **general** than those of superclass method.

-When a method in class declares that it throws an exception that is an instance of a class, it may throw an exception of that class or any of its subclasses.

## 7.1.3 How to Throw an Exception

-Example: reading a file



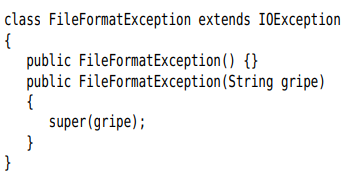
+EOFException has a 2nd constructor that takes string argument:

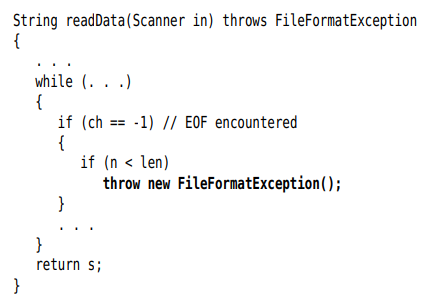


-**Throwing an exception** is easy: find an appropriate exception class/Make an object of that class/throw it.

## 7.1.4 Creating Exception Classes

-Derive your own exception class from Exception or a child class of Exception. Give both a default constructor and constructor that contains detailed message.





-**Note**: Throwable: Throwable(), Throwable(String message), getMessage()

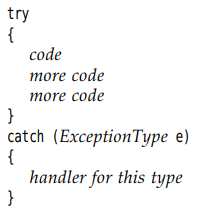
# 7.2 Catching Exception

## 7.2.1 Catching an Exception

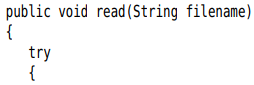
-If an **exception** is **not caught**, the program will **terminate** and print a **message** to the console..

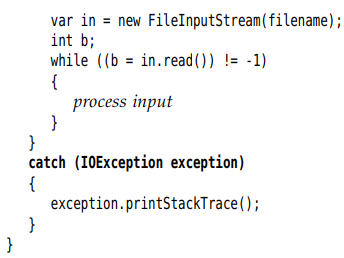
-GUI programs catch exceptions, print stack trace and go back to UI processing loop.

-To **catch an exception**, Set up **try/catch** block

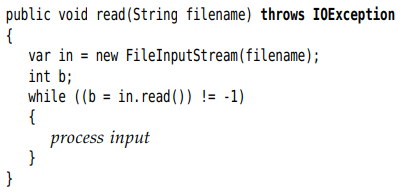


+If any code inside **try block** throw an exception of class specified in **catch clause**: the program **skips** the **remainder code** in try block and executes **handler code** in **catch clause**.



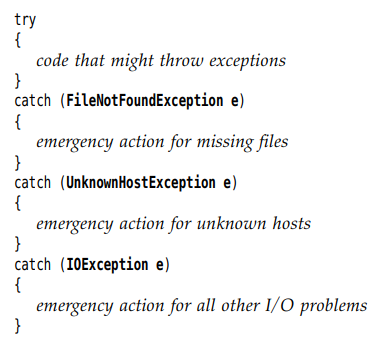


+Often, if an error occurs, let the **caller** of read method **worry** about it:



-If you call a method that throws a checked exception, you must handle it or pass it on. Look at the API documentation to see what methods throw which exceptions. Then decide whether you should handle them or add them to throws list.

## 7.2.2 Catching Multiple Exceptions

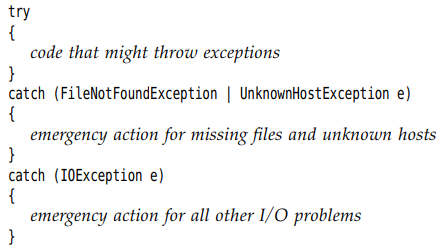


-Exception object may contain information about the nature of exception:





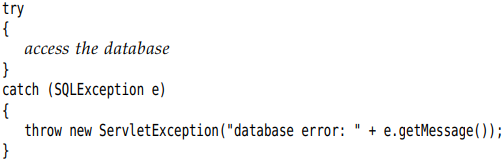
-You can catch multiple exception types in the **same catch clause**:



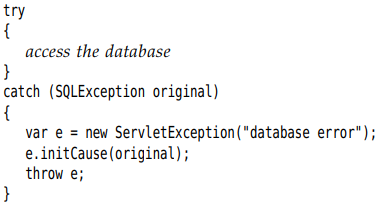
+This feature only needed when catching exception types are **not subclasses** of one another. The exception variable is implicitly **final**

## 7.2.3 Rethrowing and Chaining Exceptions

-You can **throw an exception** in **catch clause** to **change** the **exception type**.



-Better idea is to set the **original exception** as the **“cause”** of **new exception**:

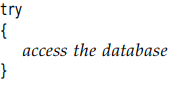


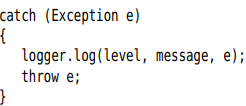
+When the exception is caught, the **original exception** can be **retrieved**:



+This wrapping technique is recommended. It allows you to throw high-level exceptions in subsystems without losing the details of original failure.

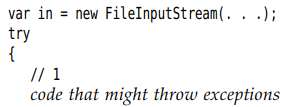
-**Log an exception** and **rethrow** it without any change

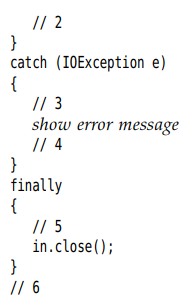




## 7.2.4 The finally Clause

-To **continue executing** the **method** when an **exception** is **caught**:



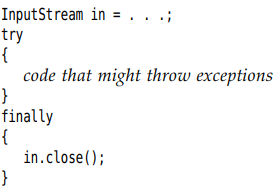


+The code throws no exceptions: 1,2,5,6

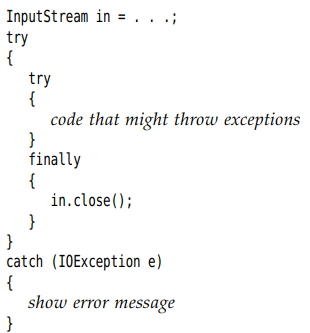
+The code throw an exception: if the catch does not throw an exception: 1,3,4,5,6; if the catch clause throws an exception: 1,3,5

+The code throw an exception that is not caught: 1,5

-You can use **finally** clause **without** a **catch** clause:



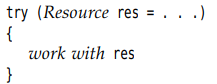
+Method in **finally clause** is executed **whether or not** an **exception** is encountered. If an exception is encountered, it is **rethrown** and must be **caught** in **another catch** clause



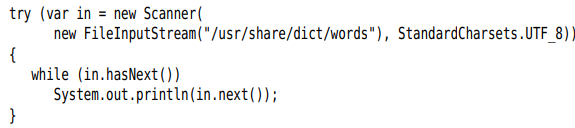
## 7.2.5 The try-with-Resources Statement

-As of Java 7, there is a useful shortcut to the code pattern: try {} finally {} provided the resource belongs to a class that implements **AutoCloseable** interface having a single method: 

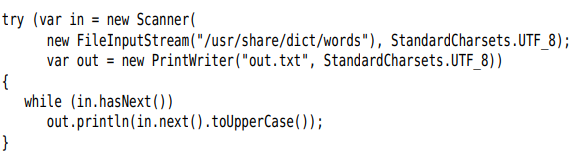
-The simplest **try-with-resources**



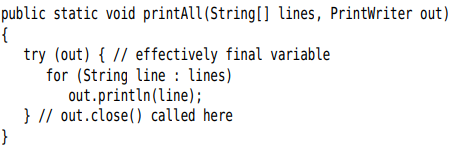
+When try block exits**, res.close()** called automatically:



+Specify **multiple resources**:



-Java 9: you can provide previously declared effectively **final variables** in **try header**



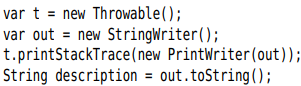
-Use the **try-with-resources** statement whenever you need to **close a resource**.

-**Note**: try-with-resource statement **can** have **catch and finally**, there are **executed** **after** **closing** the **resources**.

## 7.2.6 Analyzing Stack Trace Elements

-A **stack trace** is a listing of all pending method calls at a particular point in the execution of program. You have seen stack trace listings whenever Java program **terminates** with **uncaught exception**.

-You can access the text description of a stack trace by **printStackTrace()** of **Throwable** class:



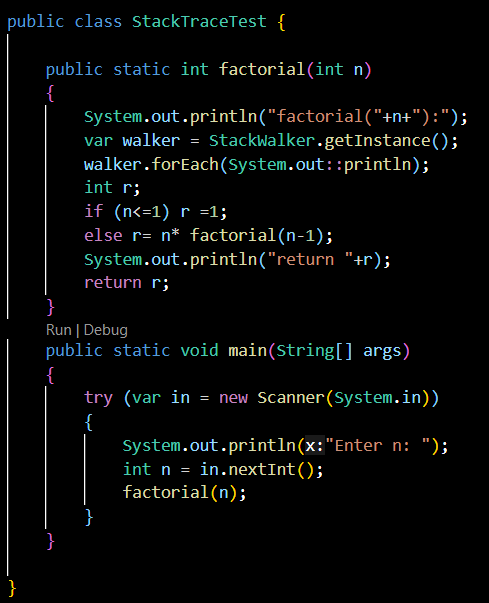
-A more flexible approach is **StackWalker** class that yield a stream of **StackWalker.StackFrame** instances, each describing one stack frame. You can iterate over the stack frames with:



-Process the Stream<StackWalker.StackFrame>



-Example: The stack trace of recursive factorial function:



-API documents: p393

# 7.3 Tips for Using Exceptions

-Exception handling is not supposed to replace a simple test.

-Do not micromanage exception

-Make good use of the exception hierarchy.

-Do not squelch exception

-When you detect an error, “tough love” works better than indulgence.

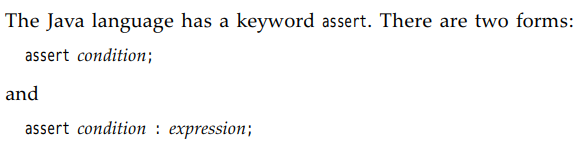
-Propagating exceptions is not a sign of shame

# 7.4 Using Assertions

-Assertions are idiom of defensive programming.

7.4.1 The Assertion Concept

-The assertion mechanism allows you to put in checks during testing and automatically removed them in production code.



+Both evaluate the condition and throw **AssertionError** if fit is false. The 2nd statement, the expression is passed in AssertionError object and turned into message string

-Example:

+assert x is non-negative: assert x >= 0;

+Display x later: assert x >= 0 : x;

7.4.2 Assertion Enabling and Disabling

-By default, assertions are disabled.

-Enable them by running program with **–enableassertions** or **–ea**



-You don’t need to recompile the program to enable or disable assertion. Enable or disable assertion is a function of **class loader**.

-You can turns on assertions in specification classes or in entire packages:



-Disable assertions: 

7.4.3 Using Assertions for Parameter Checking

- 3 mechanisms to deal with system failures: throwing an exception, logging, using assertions.

-When to use assertions: failures are intended to be fatal, unrecoverable errors, assertions checks are turned on only during development and testing. Assertions should only be used to locate internal program errors during testing.

-*Precondition*: Look at the method contract to decide to make assertions.

7.4.4 Using Assertions for Documenting Assumptions

-Use comments to document the assumptions -> assertion instead.

# 7.5 Logging

-Java programmer is familiar with inserting calls to System.out.println() into troublesome code to gain insight into program. The logging API overcomes this problem.

## 7.5.1 Basic Logging

-Simple logging: use global logger and call its info()



-Suppress all logging:



## 7.5.2 Advanced Logging

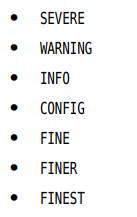
-Define your own loggers:



+A logger that is not referenced by any variable can be garbage-collected. -> use static

-Logger names are hierarchical. Logger parents and children share certain properties.

-There are 7 logging levels:



+By default: top 3 levels are logged.

-You can set different level:



-**Level.ALL, Level.OFF**: turn on and off all logging.

-There are logging methods for all levels:



+Alternatively, you can use **log()** and supply the level:

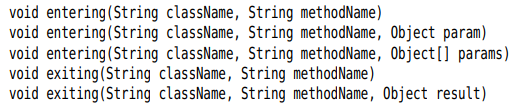


+Tip: the default logging configuration logs all records with level of INFO or higher. Use other levels for **debugging messages** that are useful for diagnostics but meaningless to user.

-The default log record shows the name of class and method contain the logging call. You can use **logp()** to give the precise location of calling class and method

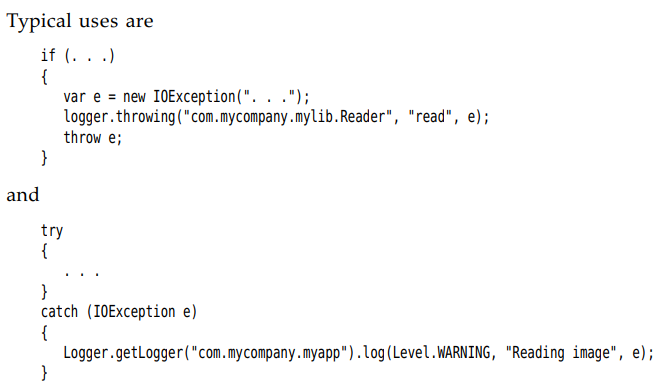


-Tracing execution flow methods:



-A common use for logging is to log unexpected exceptions. 2 methods include a description of exception in log record





## 7.5.3 Changing the Log Manager Configuration

-You can change properties of logging system by editing configuration file in jdk-17.0.8/conf/logging.properties.

-Set another configuration file:



-Change the default logging level: 

-Specify the logging levels for your own loggers by adding lines:loggername +.level=FINE

-Handlers send the messages to the console and they have levels. Set the level of handler:



-The log manager is initialized during VM startup, before main executes.

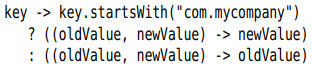
-You can update the logging configuration by calling:



+Merge the old and new configuration:



+update the keys start with “”:



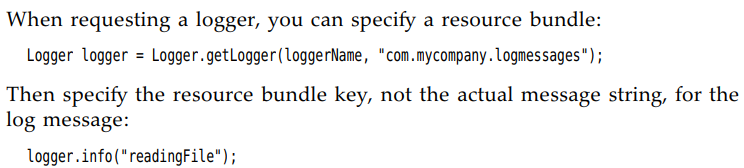
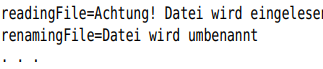
-It’s possible to change logging levels in a running program by using **jconsole** program.

-Note: The logging properties file is processed by java.util.logging.LogManager class.

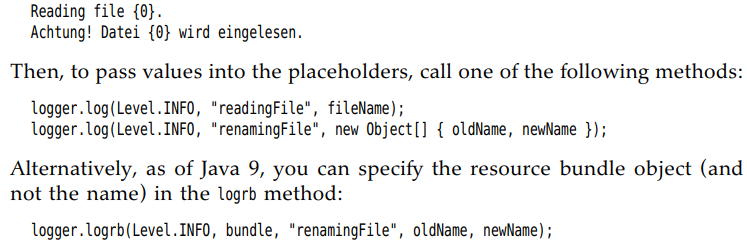
## 7.5.4 Localization

-You want to localize logging messages so that they are readable for international users.

-Localized apps contain locale-specific information in **resource bundles** which consist of a set of mappings for various locales. A program may contain multiple resource bundles (like com.mycompany.logmessages), supply a file for locale to add mapping to (com/mycompany/logmessages\_en.properties).



-You often need to include arguments into localized messages:

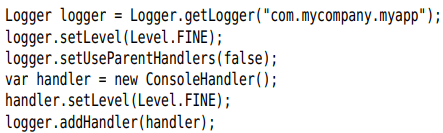


## 7.5.5 Handlers

-By default, loggers send records to **ConsoleHandler** that prints them to **System.err** stream. The logger sends the record to the ultimate ancestor (name “”) has a ConsoleHandler. For a record to be logged, its logging level must be above the threshold of both logger and handler.

-The log manager configuration file sets the logging level of default console handler: 

-You can install your own handler:

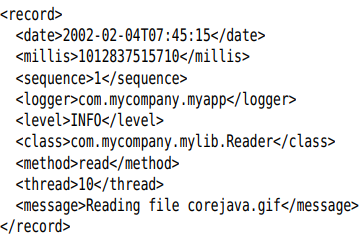


-To send log records elsewhere, add another handles: **FileHandler** and **SocketHandler.**

-Send records to a default **file handler** (name javan.log – n is number to make the file unique)



+By default, the records are formatted in XML



+You can modify the default behavior of file handle by setting parameters in log manager configuration: p412

+You should use another log file name: p413

+Log files are kept in a rotation sequence. When a file exceeds the size limit, the oldest log is deleted, the other files are renamed and new file is created

-You can also define your own handlers by extending **Handler** or **StreamHandler** class: p414

## 7.5.6 Filters

-By default, records are filtered according to their logging levels.

-To define a filter, implement the **Filter** interface and define the method 

-To install a filter into logger or handler:  method

## 7.5.7 Formatters

-ConsoleHandler and FileHandler classes emit the log records in text and XML formats.

-To define your own format, extend the **Formatter** class and override 





-Install the formatter into the handler: method

## 7.5.8 A Logging Recipe

-For simple app, choose a single logger with the same name as the main app package.

+Get the logger: 

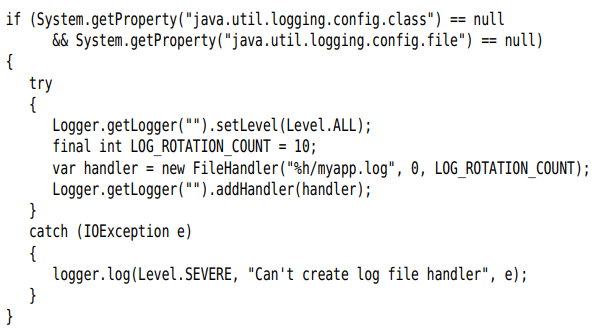
+Add static fields to classes:



- The default logging configuration logs all messages of level >= INFO to the console.

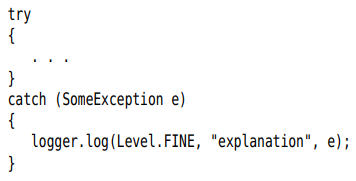
+It’s a good idea to install a more reasonable default in your app.

+This code ensures all messages are logged to an app-specific file. Place the code into main()



-INFO, WARNING, SEVERE show up the console, use these level for messages to the users of the program. The level FINE is for programmers messages: 

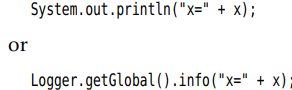
+Log unexpected exceptions:



-**Logging API**: p420

# 7.6 Debugging Tips

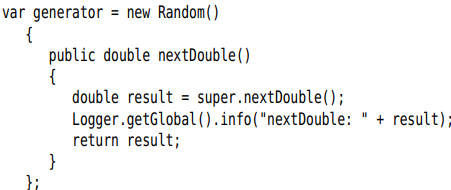
-You can print or log the value of any variable with code:



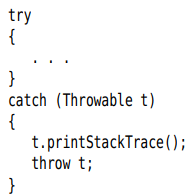
-Put a separate main() in each class for unit test

-You can check out JUnit from <http://junit.org>

-Logging proxy: ab object of subclass that intercepts method calls, logs them, and calls the superclass.



-Get a stack trace from any exception object with:

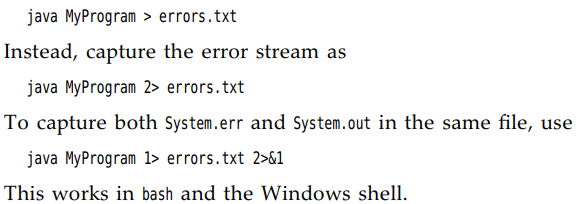


 anywhere to get a stack trace

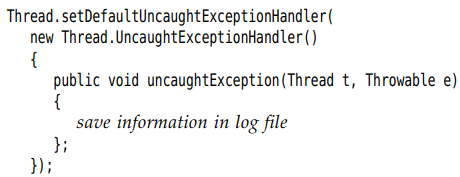
-Normally, stack trace is displayed on System.err. To log or display the stack trace:



-Trap program errors in a file



-Log uncaught exceptions to a file, change the handler for them with:



-Watch class loading: + **-verbose**

-**-Xlint** tells the compiler to spot common code problems



-JVM supports for **monitoring and management** of Java apps: **jconsole** displays statistics about the performance of a VM.

-Java Mission Control.