**Exceptions**

* **What is an Exception**
  + The term *exception* is short hand for the phrase *exceptional event*.
  + An *exception is an event*, which occurs during the execution of a program that disrupts the normal flow of the program’s instruction.
  + When an error occurs within a method, the method *creates an object* and hands it off to the *runtime system*.
  + The object called an *exception object,* contain the information about the error, including its type and the state of the program when the error occurred.
  + Creating an exception object and handing it to the runtime system is called *throwing an exception*.
  + After a method throws an exception, the runtime system attempts to find something to *handle* it.
  + The set of possible *somethings to handle* the exception is the ordered list of methods that had been called to get to the method where the error occurred.
  + The list of method is known as the *call stack*.

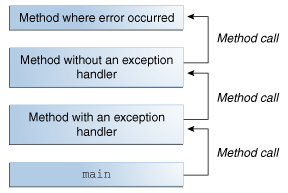


Fig : The call stack

* + The runtime system searches the call stack for a method that contains a block of code that *can handle the exception*.
  + The block of code is called an *exception handler*.
  + The search begins with the method in which the *error occurred* and proceeds through the call stack in the *reverse order* in which the methods were called.
  + When an appropriate handler is found, the runtime system passes the exception to the *handler*.
  + An exception handler is considered appropriate if the *type of the exception object* thrown matches the type that can be handled by the handler.
  + The exception handler chosen is said to *catch the exception*
  + If the runtime system without finding an appropriate exception handler, the *runtime system terminates*.

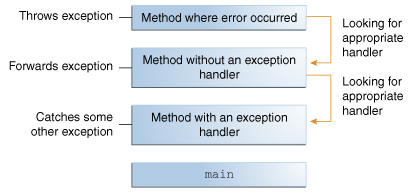


Fig: Searching the call stack for the exception handler

* **The Catch or Specify Requirement**
  + Valid java programming language code must honor the *catch or specify requirement*.
  + This means that the code that might *throw certain exceptions* must be enclosed by either of the following.
    - A *try statement* that catches the exception. The try must provide a handler for the exception.
    - A method that specifies that *it can throw the exception*. The method must provide a throws clause that lists the exception.
  + Code that fails to honor the catch or specify requirement *will not compile*.
  + **The Three Kinds of Exception**
    - **Checked Exception**
      * These are *exceptional conditions* that a well written application should *anticipate* and *recover* from.
      * For example, suppose an application prompts a user for an *input file name*.
      * Then opens the file by *passing the name* to the constructor for java.io.FileReader
      * Normally the user provides the name of an *existing, readable file*.
      * So the construction of the *file reader objects succeeds* and the execution of the application proceeds normally.
      * But sometimes the user supplies the *name of a non-existent file*, and the constructor throws java.io.FileNotFoundException.
      * A well written application *will catch this exception* and notify the user of the mistake, possibly prompting for a corrected file name. Checked exceptions are *subject to* the Catch or specify requirement.
    - **Error**
      * These are exceptional conditions that are *external to the application*, and that the application *cannot anticipate or recover from*.
      * For example, suppose that an application successfully opens a file for input but it’s unable to read the file because of *hardware or system malfunction*.
      * The unsuccessful read will throw java.io.IOError
      * Errors are *not subject* to catch or specify requirement.
      * Errors are those exceptions indicated by *Error and its subclasses*.
  + **Runtime Exception**
    - * These are exceptional conditions that are *internal to the application*, and that the application usually *cannot* anticipate or recover from.
      * These usually indicate programming bugs such as *logic error* or *improper use* of an API.
      * For example consider the application described previously that passes the file name to the *constructor* for *FileReader*.
      * If logic error causes a *null to be passed* to the constructor, the constructor will throw the NullPointerException.
      * Runtime exceptions are *not* subject to Catch or specify requirement.
      * Runtime exceptions are those indicated by RuntimeException and its *sub classes.*
  + Errors and runtime exceptions are collectively known as *unchecked exception*.
* **Catching And Handling Exceptions**
  + This section describes how to use the *three exception handler components*- the try, catch and finally blocks – to write an exception handler.
  + Then the *try with resource statement*, introduced in Java SE7.
  + The try with resource statement is particularly suited to situations that use *closeable resources*, such as streams.

// Note: This class won't compile by design!

import java.io.\*;

import java.util.List;

import java.util.ArrayList;

public class ListOfNumbers {

private List<Integer> list;

private static final int SIZE = 10;

public ListOfNumbers () {

list = new ArrayList<Integer>(SIZE);

for (int i = 0; i < SIZE; i++) {

list.add(new Integer(i));

}

}

public void writeList() {

PrintWriter out = new PrintWriter(**new FileWriter("OutFile.txt")**);

for (int i = 0; i < SIZE; i++) {

out.println("Value at: " + i + " = " + **list.get(i)**);

}

out.close();

}

}

* + The first line in boldface is a *call to a constructor*.
  + The constructor *initializes* an output stream on a file.
  + If the file cannot be opened, the constructor throws an *IOException*.
  + The second boldface line is a call to the *ArrayList class’s get method*.
  + Which throws an *IndexOutofBoundsException* if the value of its argument is too small or too large.
  + **The try Block**
    - The first step in *constructing an exception handler* is to enclose the code that might throw an exception within a *try block.*
    - In general, a try block looks like the following

try {

code

}

catch and finally blocks . . .

* + - To construct an exception handler for the *writeList method* from the *ListOfNumbers class*, enclose the exception-throwing statements of the writeList method within a try block.
    - There is more than one way to do this, you can put *each line of the code* that might throw an exception within its *own try block* and provide separate exception handler for each.
    - Or you can put *all the writeList code* within a single try block and associate multiple handlers with it.

private List<Integer> list;

private static final int SIZE = 10;

PrintWriter out = null;

try {

System.out.println("Entered try statement");

out = new PrintWriter(new FileWriter("OutFile.txt"));

for (int i = 0; i < SIZE; i++) {

out.println("Value at: " + i + " = " + list.get(i));

}

}

catch and finally statements . . .

* + - If any exception occurs within the try block, that exception is handled by an *exception handler* associated with it.
    - To associate an exception handler with a try block, you must put a *catch block* after it.
  + **The catch Blocks**
    - You associate exception handlers with a try block by providing one or more *catch blocks* directly after the try block.

try {

} catch (*ExceptionType name*) {

} catch (*ExceptionType name*) {

}

* + - Each catch block is an *exception handler* and handles the *type of exception* indicated by its argument.
    - The argument type, *ExceptionType*, declares the type of exception that the handler can handle and must be the name of a class that inherits from the *Throwable class*.
    - The handler can refer to the *exception with name*.
    - The catch block contains code that is executed if and when the exception handler is *invoked*.
    - *The runtime system* invokes the exception handler when the handler is the first one in the *call stack* whose ExceptionType matches the type of the exception thrown.
    - The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

try {

} catch (FileNotFoundException e) {

System.err.println("FileNotFoundException: " + e.getMessage());

throw new SampleException(e);

} catch (IOException e) {

System.err.println("Caught IOException: " + e.getMessage());

}

* + - Both handlers print an error message. The second handler does nothing else. By *catching any IOException* that's not caught by the first handler, it allows the program *to continue executing*.
    - The first handler, in addition to printing a message, *throws a user-defined exception.*
    - Exception handlers *can do more* than just print error messages or halt the program.
    - They can do *error recovery*, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions.
  + **Catching More Than One Type of Exception with One Exception Handler**
    - In Java SE7 and later, a *single catch block* can handle *more than one type* of exception.
    - In the catch clause, specify the types of exception that block can handle, and separate each exception type with *a vertical bar(|).*

catch (IOException|SQLException ex) {

logger.log(ex);

throw ex;

}

* + - If a catch block handles more than one exception type, then the *catch* *parameter* is implicitly *final*.
    - In this example the catch parameter ex is final and therefore you *cannot assign any values* to it within catch block.
  + **The finally Block**
    - The finally block *always executes* when the try block exits.
    - This ensures that the finally block is executed even if an *unexpected exception* occurs.
    - But finally is useful for more than just exception handling — it allows the programmer to avoid having *cleanup code* accidentally bypassed by a return, continue, or break.
    - Putting cleanup code in a finally block is always a *good practice*, even when no exceptions are anticipated.
    - If the *JVM exits* while the try or catch code is being executed, then the finally block may not execute.
    - Likewise, if the *thread executing* the try or catchcode is *interrupted* or *killed*, the finally block may not execute even though the application as a whole continues.
    - The following *finally block* for the *writeList method* cleans up and then closes the PrintWriter.

finally {

if (out != null) {

System.out.println("Closing PrintWriter");

out.close();

} else {

System.out.println("PrintWriter not open");

}

}

* + - The finally block is a key tool for *preventing resource leaks*.
    - When closing a file or otherwise recovering resources, place the code in a finallyblock to ensure that *resource is always recovered*.
    - If you are using Java SE 7 or later, consider using the *try-with-resources statement* in these situations, which *automatically releases* system resources when no longer needed.
  + **The try-with-resources Statement**
    - The try-with-resources statement is a *try statement that declares one or more resources.*
    - A resource*is an object* that must be closed after the program is finished with it.
    - The try-with-resources statement ensures that *each resource is closed* at the *end* of the statement.
    - Any object that implements java.lang.*AutoCloseable*, which includes all objects which implement java.io.*Closeable*, can be used as a resource.

static String readFirstLineFromFile(String path) throws IOException {

**try (BufferedReader br =**

**new BufferedReader(new FileReader(path)))** {

return br.readLine();

}

}

* + - In this example, the *resource* declared in the try-with-resources statement is a *BufferedReader*.
    - The *declaration statement* appears within parentheses immediately after the try keyword.
    - The class BufferedReader, in Java SE 7 and later, *implements* the interface java.lang.AutoCloseable.
    - Because the *BufferedReader instance* is declared in a try-with-resource statement.
    - It will be *closed* regardless of whether the try statement completes normally or abruptly.
    - Prior to Java SE 7, you can use a *finally block* to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly.
    - The *exception* thrown from the try-with-resources block is *suppressed*. In Java SE 7 and later, you can retrieve suppressed exceptions.
    - You may *declare one or more resources* in a try-with-resources statement.
    - The following *example* retrieves the names of the files packaged in the zip file zipFileName and creates a text file that contains the names of these files.

public static void writeToFileZipFileContents(String zipFileName,

String outputFileName)

throws java.io.IOException {

java.nio.charset.Charset charset =

java.nio.charset.StandardCharsets.US\_ASCII;

java.nio.file.Path outputFilePath =

java.nio.file.Paths.get(outputFileName);

// Open zip file and create output file with

// try-with-resources statement

**try (**

**java.util.zip.ZipFile zf =**

**new java.util.zip.ZipFile(zipFileName);**

**java.io.BufferedWriter writer =**

**java.nio.file.Files.newBufferedWriter(outputFilePath, charset)**

**)** {

// Enumerate each entry

for (java.util.Enumeration entries =

zf.entries(); entries.hasMoreElements();) {

// Get the entry name and write it to the output file

String newLine = System.getProperty("line.separator");

String zipEntryName =

((java.util.zip.ZipEntry)entries.nextElement()).getName() +

newLine;

writer.write(zipEntryName, 0, zipEntryName.length());

}

}

}

* + - In this example, the try-with-resources statement contains *two declarations* that are separated by a semicolon: ZipFile and BufferedWriter.
    - When the block of code that directly follows it *terminates*, either normally or because of an exception.
    - The *close methods* of the BufferedWriter and ZipFile objects are *automatically* called in this order.
    - Note that the *close methods* of resources are called in the opposite*order* of their creation.
    - A try-with-resources statement can have *catch* and *finally blocks* just like an ordinary try statement.
    - In a try-with-resources statement, any catch or finallyblock is run after the *resources* *declared* have been *closed*.
  + **Suppressed Exceptions**
    - An *exception can be thrown* from the block of code associated with the try-with-resources statement.
    - In the example writeToFileZipFileContents, an exception can be thrown from the *try block*.
    - And up to *two exceptions* can be thrown from the try-with-resources statement when it tries to close the ZipFile and BufferedWriter objects.
    - If an exception is thrown from the try block and one or more exceptions are thrown from the try-with-resources statement.
    - Then *those exceptions* thrown from the try-with-resources statement are *suppressed*.
    - And the exception thrown by the block is the one that is thrown by the writeToFileZipFileContents method.
    - You can *retrieve* these *suppressed exceptions* by calling the *Throwable.getSuppressed* method from the exception thrown by the try block.
  + **Classes That Implements the AutoCloseable or Closeable Interface**
    - The Closeable interface *extends* the AutoCloseable interface.
    - The *close method* of Closeable interface throws exceptions of type *IOException*.
    - While the *close method* of the AutoCloseable interface throws exception of type *Exception*.
    - Consequently subclasses of AutoCloseable interface *can override* this behaviour of the close method to throw specialized exception such as IOException or no exception at all.

* + **Putting It All Together**
    - When *all the components are put together*, the writeList method looks like the following.

public void writeList() {

PrintWriter out = null;

try {

System.out.println("Entering" + " try statement");

out = new PrintWriter(new FileWriter("OutFile.txt"));

for (int i = 0; i < SIZE; i++)

out.println("Value at: " + i + " = " + vector.elementAt(i));

} catch (ArrayIndexOutOfBoundsException e) {

System.err.println("Caught ArrayIndexOutOfBoundsException: "

+ e.getMessage());

} catch (IOException e) {

System.err.println("Caught IOException: " + e.getMessage());

} finally {

if (out != null) {

System.out.println("Closing PrintWriter");

out.close();

}

else {

System.out.println("PrintWriter not open");

}

}

}

* + - As mentioned previously, this method's try block has *three different exit* possibilities; here are two of them.
      * Code in the try statement fails and throws an exception.
      * Everything succeeds and the try statement exits normally.
    - Let's look at *what happens* in the writeList method *during these two exit* possibilities.
    - **Scenario 1: An Exception Occurs**
      * When FileWriter throws an IOException, the *runtime system immediately stops executing the try block*; method calls being executed are not completed.
      * The *runtime system* then starts searching at the top of the method *call stack* for an appropriate exception handler.
      * In this example, when the IOException occurs, the FileWriter constructor is at the top of the call stack.
      * However, the FileWriter constructor doesn't have an appropriate exception handler, so the runtime system checks the next method — the writeList method — in the method call stack.
      * The writeList method has *two exception handlers*: one for IOException and one for ArrayIndexOutOfBoundsException.
      * The runtime system checks writeList's handlers in the order in which they appear after the try statement.
      * The argument to the first exception handler is ArrayIndexOutOfBoundsException.
      * This *does not match* the type of exception thrown, so the runtime system checks the next exception handler — IOException.
      * This matches the *type of exception* that was thrown, so the runtime system ends its search for an appropriate exception handler.
      * Now that the runtime has *found an appropriate handler*, the code in that catch block is executed.
      * After the exception handler executes, the runtime system passes *control* to *the finally block*.
      * Code in the finally block executes regardless of the exception caught above it. In this scenario, the FileWriter was never opened and doesn't need to be closed.
      * After the finally block finishes executing, the *program continues* with the first statement after the finally block.
      * Here's the *complete output* from the ListOfNumbers program that appears when an IOException is thrown.
        + Entering try statement
        + Caught IOException: OutFile.txt
        + PrintWriter not open
    - **Scenario 2: The try Block Exits Normally**
      * In this scenario, all the statements within the scope of the try block *execute successfully* and throw no exceptions.
      * Execution falls off the end of the *try block*, and the runtime system passes control to the *finally block*.
      * Because everything was successful, the PrintWriter is open when control reaches the finally block, which closes the PrintWriter.
      * Again, after the finally block finishes executing, the program continues with the *first statement* after the finally block.
      * Here is the *output* from the ListOfNumbers program when no exceptions are thrown.
        + Entering try statement
        + Closing PrintWriter
* **Specifying the Exceptions Thrown by a Method**
  + The previous section showed how to write an *exception handler* for the *writeList* method in the ListOfNumbers class.
  + Sometimes, it's appropriate for *code* to *catch exceptions* that can occur within it.
  + In other cases, however, it's better to let a method further up the *call stack* handle the exception.
  + If the writeList method *doesn't catch* the checked exceptions that can occur within it, the writeList method must specify that *it can throw* these exceptions.
  + To specify that writeList can throw two exceptions, add a *throws clause* to the method declaration for the writeList method.
  + The throws clause comprises the *throws keyword* followed by a comma-separated list of all the exceptions thrown by that method.
  + The clause goes *after the method name* and argument list and before the brace that defines the scope of the method; here's an example.

*public void writeList() throws IOException, ArrayIndexOutOfBoundsException { }*

* + Remember that ArrayIndexOutOfBoundsException is an *unchecked* exception.
  + Including it in the throws clause is *not mandatory*.
  + You could just write the following.

*public void writeList() throws IOException {}*

* **How to Throw Exceptions**
  + Before you can catch an exception, some code somewhere must *throw* one.
  + Any code *can throw an exception*: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment.
  + Regardless of what throws the exception, it's always thrown with the *throw statement.*
  + **The throw Statement**
    - All methods use the *throw statement* to throw an exception.
    - The throw statement requires a single argument: *a throwable object*.
    - Throwable objects are *instances* of any subclass of the *Throwable class*. Here's an example of a throw statement.

*throw someThrowableObject;*

* + - Let's look at the throw statement in context. The following pop method is taken from a class that implements a common stack object.
    - The method *removes* the top *element* from the *stack* and returns the object.

public Object pop() {

Object obj;

if (size == 0) {

**throw new EmptyStackException();**

}

obj = objectAt(size - 1);

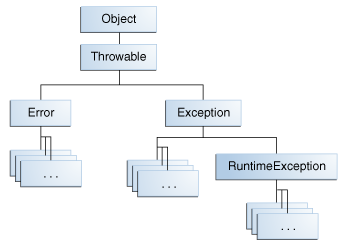
setObjectAt(size - 1, null);

size--;

return obj;

}

* + - The pop method checks to see whether *any elements* are on the stack.
    - If the stack is empty (its size is equal to 0), pop instantiates a new *EmptyStackException* object (a member of java.util) and throws it.
    - You need to remember is that you can *throw* only *objects* that inherit from the *java.lang.Throwable* class.
  + **Throwable Class and Its Subclasses**
    - The objects that inherit from the Throwable class include *direct descendants* (objects that inherit directly from the Throwable class) and *indirect descendants* (objects that inherit from children or grandchildren of the Throwable class).
    - The figure below illustrates the class *hierarchy* of the Throwable class and its most significant subclasses.
    - As you can see, Throwable has two direct descendants: *Error* and *Exception*.



* + **Error Class**
    - When a *dynamic linking* failure or other *hardware* failure in the Java virtual machine occurs, the virtual machine throws an Error.
    - Simple programs typically *do not catch or throw Errors*.
  + **Exception Class**
    - Most programs throw and catch objects that derive from the *Exception class*.
    - An Exception indicates that a problem occurred, but it is *not a serious system problem*.
    - Most programs you write will throw and catch Exceptions as opposed to *Errors*.
    - The Java platform defines the many *descendants* of the Exception class.
    - These descendants indicate *various types* of exceptions that can occur.
    - For example, *IllegalAccessException* signals that a particular *method could not* *be found*.
    - And *NegativeArraySizeException* indicates that a program attempted to create an *array with a negative size*.
    - One Exception subclass, *RuntimeException*, is reserved for exceptions that indicate *incorrect* use of an API.
    - An example of a runtime exception is *NullPointerException*, which occurs when a method tries to access a member of an object through a *null reference*.
* **Chained Exceptions**
  + An application often responds to an exception by *throwing another exception*.
  + In effect, the first exception *causes* the second exception.
  + It can be very helpful to know when one exception *causes another*.
  + *Chained Exceptions* help the programmer do this.
  + The following are the *methods* and *constructors* in *Throwable* that support chained exceptions.

Throwable getCause()

Throwable initCause(Throwable)

Throwable(String, Throwable)

Throwable(Throwable)

* + The *Throwable argument* to initCause and the Throwable constructors is the exception that caused the current exception.
  + *getCause* returns the exception that caused the current exception.
  + And *initCause* sets the current exception’s cause.
  + The following *example* shows how to use a chained exception.

try {

} catch (IOException e) {

throw new SampleException("Other IOException", e);

}

* + In this example when an *IOException* is caught, a new *sampleException* is created with the original cause attached
  + And the chain of exceptions is thrown up to *next higher level* exception handler.
  + **Accessing Stack Trace Information**
    - A *stack trace* provides information on the *execution history* of the current thread and lists the name of the *classes and methods* that were called at the point when the exception occurred.
    - A stack trace is a useful *debugging tool* that you will normally take advantage of when an exception has been thrown.
    - The following code shows how to call the *getStackTrace* method on the *exception object*.

catch (Exception cause) {

StackTraceElement elements[] = cause.getStackTrace();

for (int i = 0, n = elements.length; i < n; i++) {

System.err.println(elements[i].getFileName()

+ ":" + elements[i].getLineNumber()

+ ">> "

+ elements[i].getMethodName() + "()");

}

}

* + **Logging API**
    - The next code snippet *logs* *where an exception occurred* from within the catch block.
    - However, rather than *manually parsing* the stack trace and sending the output to System.err().
    - It sends the output to a file using the *logging facility* in the java.util.logging package.

try {

Handler handler = new FileHandler("OutFile.log");

Logger.getLogger("").addHandler(handler);

} catch (IOException e) {

Logger logger = Logger.getLogger("*package.name*");

StackTraceElement elements[] = e.getStackTrace();

for (int i = 0, n = elements.length; i < n; i++) {

logger.log(Level.WARNING, elements[i].getMethodName());

}

}