Table of Contents

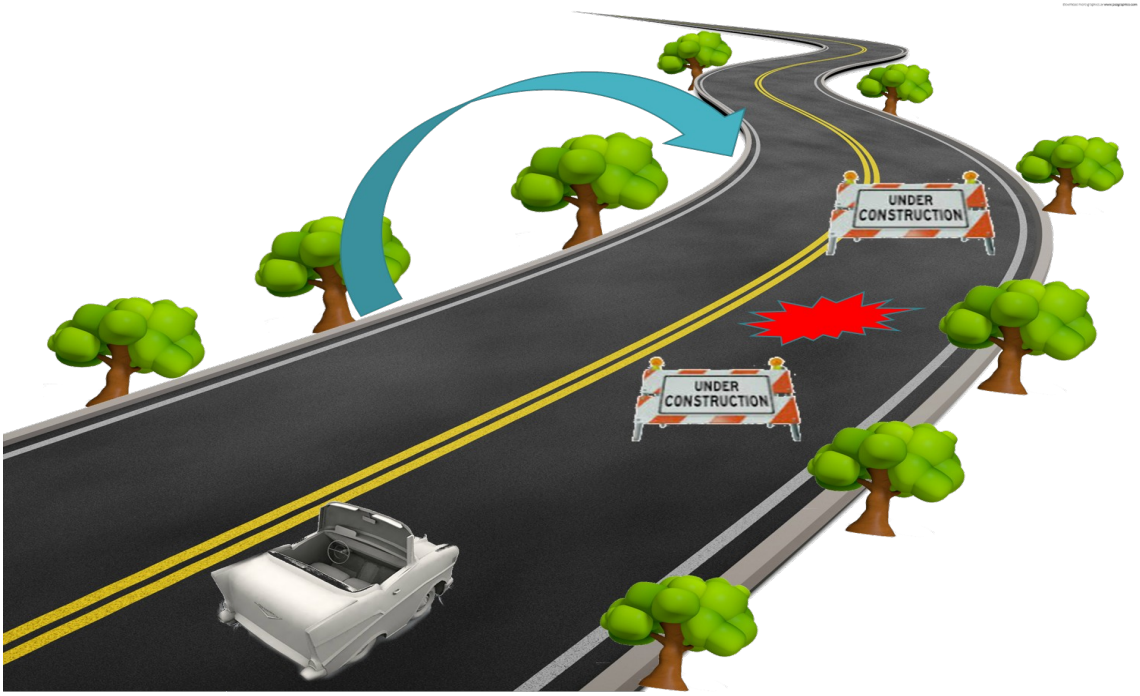
1. Pre-requisites

To Understand Exception Handling One should have programming experience in writing code using Object Oriented Concepts. The basic understanding between types of errors the code returns during the compilation and execution phases.

The exceptions occurs whenever there is wrong placement of code, exceptions are specific to the type of mistake one does.

2. Purpose with the Problem scenario

Rahul with his family is planning to go on a trip to Goa. Rahul started from Hyderabad to Goa on road on a Sedan. During his journey on a two lane road, there was a small bridge which was under construction. This is a kind of problematic situation for Rahul to continue his journey to reach the final destination.



In order to continue the journey and reach the final destination, the problematic road has to be avoided and take an alternative path and should continue the journey.

Comparing the journey with points, every point is a line of code which has to be cleared, if any part of the code has some risky or problematic syntaxes, the problem terminates. In order to

avoid this risky situations, EXCEPTION HANDLING will allow to avoid the problematic code and continues the flow of execution.

3. Introduction

Definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

4. Detailed Explanation

4.1. Content

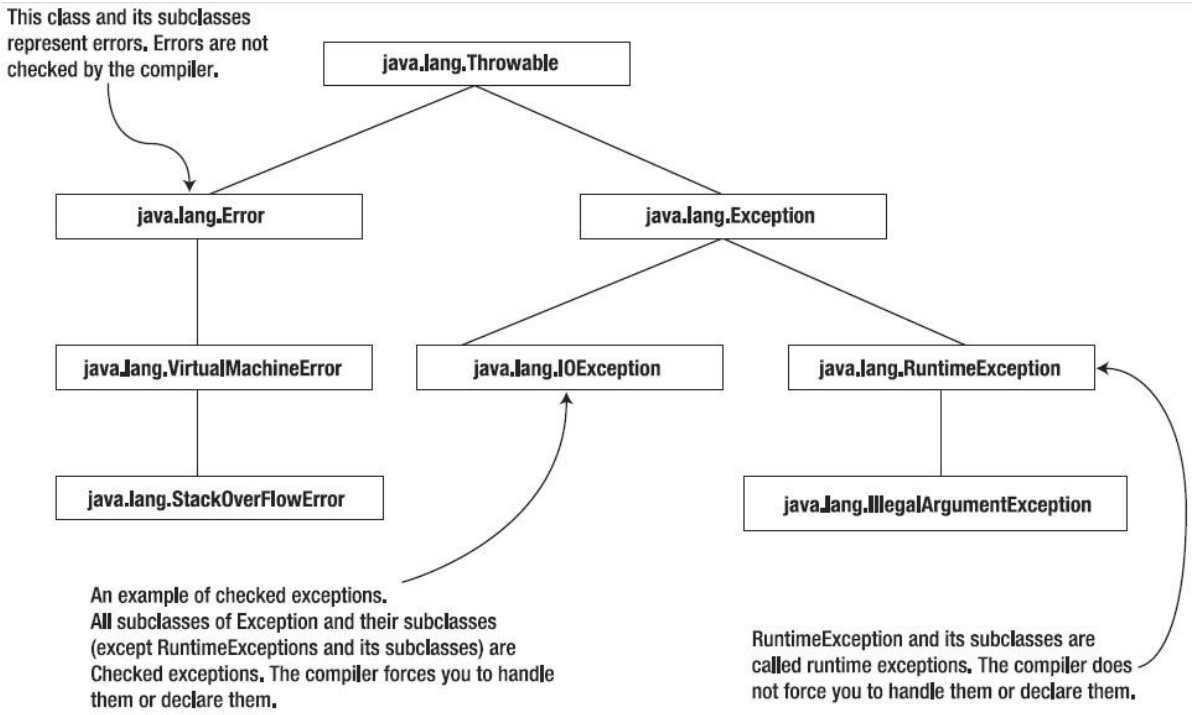
Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

➢ **Checked exceptions:**A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

➢ **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

➢ **Errors:**These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.



4.2. Un-Checked exceptions

The exceptions that are not checked at compile time are called unchecked exceptions, classes that extends RuntimeException comes under unchecked exceptions.

Examples of some unchecked exceptions are listed below.

ArithmeticException:

Mathematical operations that are not permitted in normal conditions,dividing a number from '0'.

```
package com.tcs.ilp.core;
public class ExceptionTest{
        public static void main(String[] args){
```

```
            int i =10/0;
            }
        }
```

ArrayIndexOutOfBoundsException:

Trying to access an index that does not exists or inserting values to wrong indexes results to ArrayIndexOutOfBound Exception at runtime.

```
package com.tcs.ilp.core
public class ExceptionTest{
        public static void main(String[] args){
        int arr[]={'0','1','2'};
        System.out.println(arr[4]);
        }
}
```

NullPointerException:

Trying to access a null object or performing operations on an object having a null value.

```
package com.tcs.ilp.core;
import java.util.ArrayList;
public class ExceptionTest{
        public static void main(String[] args){
        String string=null;
        System.out.println(string.length());
        }
}
```

Some common Unchecked Exceptions in Java

- ClassCastException
- IllegalStateException
- IndexOutOfBoundsException
- NegativeArraySizeException

4.3. Checked Exceptions

Exceptions that are checked at compile-time are called checked exceptions, in Exception hierarchy all classes that extends Exception class except UncheckedException comes under checked exception category.

In certain situations Java Compiler forced the programmer to write a Exception handler at compile time, the Exceptions thrown in such situation are called Checked Exception, see the example below

```
try{
 String input = reader.readLine();
 System.out.println("You typed : "+input); // Exception prone area
 }catch(IOException e) {
  e.printStackTrace();
 }
```

While writing a code to read or write something from files or even from or to console, an checked Exception i.e. IOException is thrown, these exceptions are checked at compile time and we are forced to write a handler at compile time. That's why we called these exceptions Checked Exceptions. We will know about try-catch and other Exception Handling stuff in next blog.

Some common CheckedExceptions in Java

- FileNotFoundException
- ParseException

- SQLException
- IOException

4.4. Exception Handling in Java

In java exception handling mechanism is based on following five keywords,

lets see these keywords one by one in detail:

- try
- catch
- finally
- throws
- throw

Whenever an exception is occurred in a Java program, JVM assumes that there is no sense of keeping the program running until the exception is handled or being thrown to some other piece of code to handle it. In certain situations (In case of Checked Exceptions) Java compiler forces the programmer to write a handling code to deal with exceptions. For example let's assume we are having a code that reads from an external file, JVM expects some handling code for the situation if the required file is not available at the run time. This is how Java maintains its integrity against code failure by allowing the programmer to write handling code for failure prone areas of code. In java most of the handling code is being implemented using try-catch combination with a finally block added to them.

a) How to use try-catch block in Java

Most of the time we will use try-catch block to handle exceptions of our code, and most of the exception handling stuff can be maintained using this, lets take an complete example of reading from the console. This code will read user input from the console and print the value back.

```
package com.tcs.ilp.core;

import java.io.BufferedReader;

import java.io.IOException;
```

```
import java.io.InputStreamReader;
public class ExceptionHandling{
        public static void main(String args[]){
                BufferedReader reader = new BufferedReader(new
                InputStreamReader(System.in));
                System.out.println("Enter your name : ");
                try {
                   String input = reader.readLine();
                   System.out.println("You typed : "+input);// Exception prone area
                } catch (IOException e) {
                      e.printStackTrace();
                  }
        }
}
```

Output

```
Enter your name :
Ramana Karnati
You typed : Ramana Karnati
```

Now lets discuss the above example in details and come to know more about try and catch in details.

b) try in java

A 'try' is the most commonly used keyword of Java Exception handling paradigm, a try block is one that is used to wrap the exception prone code inside it. In above example we are reading user's input from the console, we have put the reading code in try catch because we don't know if the user will type something or not, that is what handling code will do for us it will define an strategy to handle with unwanted or wrong input situation.

```
try {
   String input = reader.readLine(); // Exception prone area
   System.out.println("You typed : "+input);
```

```
}
```

Now in case the input is not correct or not entered, the control will goes to catch block instantly. Here the thing to note down is that if an Exception occurs at any code line of try block, no further code in try block will be executed after that line. Every try block must have at least one catch or finally block with it, we will come to know about finally shortly.

c) catch in Java

This is the part of try-catch where we can write code to deal with Exception conditions or some message or a stack trace of exception object to detect the exception reasons.

```
catch(IOException e){
        e.printStackTrace();
}
```

We can have more than one catch block per try, each catch block is an exception handler that handles the type of exception that is being declared as its argument. The argument must be a class name inherited from Throwable class.

```
try {
        String input=reader.readLine();//Exception pronearea
        System.out.println("You typed:"+input);
   }catch(IOException e){
e.printStackTrace();
}
catch(Exception e){
e.printStackTrace();
}
```

The advantage behind having more than one catch is that, sometimes we can not predict which exception is going to be thrown at run time, so we can provide a list of all expected exceptions in preceding catch block.

d) Multiple try-catch in Java

In situation of having more than one catch block, JVM will trace an appropriate catch block

from start and will execute accordingly. As soon as JVM finds exact catch, it will skip all other catch blocks.
Here one thing to note down is that we must have declared narrow exceptions first and broad one last, for example Exception class contains all exceptions in it, if we would write a catch with Exception class first JVM will use that catch and will not reach to exact IOException block.
Here one important thing to note down is that, at a time only one exception is thrown and only one catch is executed.

e) 'finally' keyword in java

Sometimes we have situations when a certain piece of code must be executed, no matters if try block is executed successfully or not, this is what a 'finally' block does for us.
A 'finally' block is written followed by a try block or a catch block, code written inside a finally block is always executed.
See the example below:

Using 'finally' with a try only

As we have discussed earlier, a try catch mush have atleast one catch or finally block with it. Here is the syntax of using a try followed by finally block,

```
try{
        //java code lines
}
finally{
        //java code lines that would always be executed, no matters what happened inside try block
}
```

Using 'finally' with try-catch combination

A finally block can be used with a combination of try-catch blocks,

see the example below.

```
try{
        //java code lines
 }
 catch(Exception e){
```

```
}
finally{
   //java code lines that would always be executed, no matters what happened inside try block
}
```

f) 'throws' keyword in Java

In java if a code written within a method throws an exception, there can be two cases. Either the code is being enclosed by try block and exception handling is implemented in the same method, or the method can throws the exception to the calling method simply.

This is what 'throws' does in exception handling, it throws the exception to immediate calling method in the hierarchy. The throws keyword appears at the end of a method's signature.
See the example below:

```
public void callingMethod() {
try{
calledMethod();//here the exception is handled
}catch(IOException e){
// TODOAuto-generated catch block
e.printStackTrace();
}
}
public void calledMethod()throws IOException{
BufferedReader bfr=new BufferedReader(new InputStreamReader(System.in));
bfr.readLine();//because of throws keyword the exception occurred here will be handled by calling method
}
```

The calling method itself can use a throws keyword and forward the exception to handle by its own calling method.. so on . This can be forwarder up in the hierarchy, if all calling methods are using throws and no more calling method is available to handle the exception that the exceptions are ultimately be handled by main method. And if the main method is also using an

throws itself the exception will be handled by JVM itself.

g) 'throw' keyword in Java

In java it's possible to throw an exception programmatically , throws keyword is used to throw an exception explicitly. Using throws we handle certain situations in our program. Here is an example that will validate if the age enter by the user is more than 18 or not, see the example below:

```
public void saveUserAge(in age){
        if(age<18){
        throw new ArithmeticException();;
}
else{
System.out.println("Correct age is entered");
}
}
```

5. Relating real world scenario using technical Concepts

The previous sections described how to construct the try, catch, and finally code blocks for the writeList method in the ListOfNumbers class. Now, let's walk through the code and investigate what can happen.

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. This could be an IOException caused by the new FileWriter statement or an ArrayIndexOutOfBoundsException caused by a wrong index value in the for loop.
- 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

The statement that creates a FileWriter can fail for a number of reasons. For example, the constructor for the FileWriter throws an IOException if the program cannot create or write to the file indicated.

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

The boldface code in the following listing shows the statements that get executed during this scenario:

```
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");
            }
        }
    }
```

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

The boldface code in the following sample shows the statements that get executed during this scenario.

```
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");
      }
```

```
    }
}
```

Common scenarios of Exception Handling where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

i)int a=50/0;//ArithmeticException

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

i).String s=null;

ii).System.out.println(s.length());//NullPointerException

String s=null;

System.out.println(s.length());//NullPointerException

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

i).String s="abc";

ii).int i=Integer.parseInt(s);//NumberFormatException

String s="abc";

int i=Integer.parseInt(s);//NumberFormatException

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

i).int a[]=new int[5];

ii).a[10]=50; //ArrayIndexOutOfBoundsException

5.1. Advantages of Exceptions

Now that you know what exceptions are and how to use them, it's time to learn the advantages of using exceptions in your programs.

Advantage 1: Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {
   open the file;
   determine its size;
   allocate that much memory;
   read the file into memory;
   close the file;
}
```

At first glance, this function seems simple enough, but it ignores all the following potential errors.

•What happens if the file can't be opened?

•What happens if the length of the file can't be determined?

•What happens if enough memory can't be allocated?

•What happens if the read fails?

•What happens if the file can't be closed?

To handle such cases, the readFile function must have more code to do error detection, reporting, and handling. Here is an example of what the function might look like.

```
errorCodeType readFile {
   initialize errorCode = 0;

   open the file;
   if (theFileIsOpen) {
      determine the length of the file;
      if (gotTheFileLength) {
         allocate that much memory;
         if (gotEnoughMemory) {
            read the file into memory;
            if (readFailed) {
               errorCode = -1;
            }
         } else {
            errorCode = -2;
         }
      } else {
         errorCode = -3;
      }
      close the file;
      if (theFileDidntClose && errorCode == 0) {
         errorCode = -4;
      } else {
         errorCode = errorCode and -4;
      }
   } else {
```

```
        errorCode = -5;
    }
    return errorCode;
}
```

There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. Worse yet, the logical flow of the code has also been lost, thus making it difficult to tell whether the code is doing the right thing: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing when you modify the method three months after writing it. Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the readFile function used exceptions instead of traditional error-management techniques, it would look more like the following.

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
```

```
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the readFile method is the fourth method in a series of nested method calls made by the main program: method1 calls method2, which calls method3, which finally calls readFile.

```
method1 {
    call method2;
}

method2 {
    call method3;
}

method3 {
    call readFile;
}
```

Suppose also that method1 is the only method interested in the errors that might occur within readFile. Traditional error-notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1—the only method that is interested in them.

```
method1 {
    errorCodeType error;
```

```
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

Recall that the Java runtime environment searches backward through the call stack to find any methods that are interested in handling a particular exception. A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors.

```
method1 {
```

```
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

However, as the pseudocode shows, ducking an exception requires some effort on the part of the middleman methods. Any checked exceptions that can be thrown within a method must be specified in its throws clause.

Advantage 3: Grouping and Differentiating Error Types

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy. An example of a group of related exception classes in the Java platform are those defined in java.io — IOException and its descendants. IOException is the most general and represents any type of error that can occur when performing I/O. Its descendants represent more specific errors. For example, FileNotFoundException means that a file could not be located on disk.

A method can write specific handlers that can handle a very specific exception. The FileNotFoundException class has no descendants, so the following handler can handle only one type of exception.

```
catch (FileNotFoundException e) {
```

```
    ...
}
```

A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement. For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an IOException argument.

```
catch (IOException e) {
    ...
}
```

This handler will be able to catch all I/O exceptions, including FileNotFoundException, EOFException, and so on. You can find details about what occurred by querying the argument passed to the exception handler. For example, use the following to print the stack trace.

```
catch (IOException e) {
    // Output goes to System.err.
    e.printStackTrace();
    // Send trace to stdout.
    e.printStackTrace(System.out);
}
```

You could even set up an exception handler that handles any Exception with the handler here.

```
// A (too) general exception handler
catch (Exception e) {
    ...
}
```

The Exception class is close to the top of the Throwable class hierarchy. Therefore, this handler will catch many other exceptions in addition to those that the handler is intended to catch. You may want to handle exceptions this way if all you want your program to do, for example, is print out an error message for the user and then exit.

In most situations, however, you want exception handlers to be as specific as possible. The

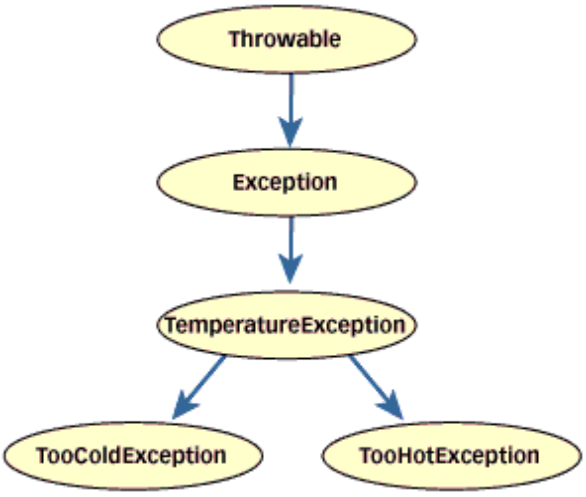
reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy. In effect, by not catching specific errors, the handler must accommodate any possibility. Exception handlers that are too general can make code more error-prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.

As noted, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

5.2. Real Time Usage

As an example, imagine you are writing a Java program that simulates a customer of a virtual cafe drinking a cup of coffee. Consider the exceptional conditions that might occur while the customer sips. The class hierarchy of exceptions shown in Figure 2 represents a few possibilities.

Figure 2. Exception hierarchy for coffee sipping



If the customer discovers, with dismay, that the coffee is cold, your program could throw a TooColdException. On the other hand, if the customer discovers that the coffee is overly hot, your program could throw a TooHotException. These conditions could be exceptions because they are (hopefully) not the normal situation in your café. (Exceptional conditions are not necessarily rare, just outside the normal flow of events.) The code for your new exception classes might look like this:

```
class TemperatureException extends Exception {
}

class TooColdException extends TemperatureException {
}

class TooHotException extends TemperatureException {
}
```

This family of classes, the TemperatureException family, declares three new types of exceptions for your program to throw. Note that each exception indicates by its class the kind of abnormal condition that would cause it to be thrown: TemperatureException indicates some kind of problem with temperature; TooColdException indicates something was too cold; and TooHotException indicates something was too hot. Note also that TemperatureException extends Exception -- not Throwable, Error, or any other class declared in java.lang.

Throwing exceptions

To throw an exception, you simply use the throw keyword with an object reference, as in:

```
throw new TooColdException();
```

The type of the reference must be Throwable or one of its subclasses.

The following code shows how a class that represents the customer, class VirtualPerson, might throw exceptions if the coffee didn't meet the customer's temperature preferences. Note that Java also has a throws keyword in addition to the throw keyword. Only throw can be used to throw an exception. The meaning of throws will be explained later in this article.

```
class VirtualPerson {
   private static final int tooCold = 65;
```

```
    private static final int tooHot = 85;
    public void drinkCoffee(CoffeeCup cup) throws
        TooColdException, TooHotException {
        int temperature = cup.getTemperature();
        if (temperature <= tooCold) {
            throw new TooColdException();
        }
        else if (temperature >= tooHot) {
            throw new TooHotException();
        }
        //...
    }
    //...
}

class CoffeeCup {
    // 75 degrees Celsius: the best temperature for coffee
    private int temperature = 75;
    public void setTemperature(int val) {
        temperature = val;
    }
    public int getTemperature() {
        return temperature;
    }
    //...
}
```

Catching exceptions

To catch an exception in Java, you write a try block with one or more catch clauses. Each catch clause specifies one exception type that it is prepared to handle. The try block places a fence

around a bit of code that is under the watchful eye of the associated catchers. If the bit of code delimited by the try block throws an exception, the associated catch clauses will be examined by the Java virtual machine. If the virtual machine finds a catch clause that is prepared to handle the thrown exception, the program continues execution starting with the first statement of that catch clause.

As an example, consider a program that requires one argument on the command line, a string that can be parsed into an integer. When you have a String and want an int, you can invoke the parseInt() method of the Integer class. If the string you pass represents an integer, parseInt() will return the value. If the string doesn't represent an integer, parseInt() throws NumberFormatException. Here is how you might parse an int from a command-line argument:

```
// In Source Packet in file except/ex1/Example1.java
class Example1 {
   public static void main(String[] args) {
      int temperature = 0;
      if (args.length > 0) {
         try {
            temperature = Integer.parseInt(args[0]);
         }
         catch(NumberFormatException e) {
            System.out.println(
               "Must enter integer as first argument.");
            return;
         }
      }
      else {
         System.out.println(
            "Must enter temperature as first argument.");
         return;
      }
      // Create a new coffee cup and set the temperature of
      // its coffee.
```

```
        CoffeeCup cup = new CoffeeCup();
        cup.setTemperature(temperature);
        // Create and serve a virtual customer.
        VirtualPerson cust = new VirtualPerson();
        VirtualCafe.serveCustomer(cust, cup);
    }
}
```

Here, the invocation of parseInt() sits inside a try block. Attached to the try block is a catch clause that catches NumberFormatException:

```
catch(NumberFormatException e) {
    System.out.println(
        "Must enter integer as first argument.");
    return;
}
```

The lowercase character e is a reference to the thrown (and caught) NumberFormatException object. This reference could have been used inside the catch clause, although in this case it isn't. (Examples of catch clauses that use the reference are shown later in this article.)

If the user types Harumph as the first argument to the Example1 program, parseInt() will throw a NumberFormatException exception and the catch clause will catch it. The program will print:

```
Must enter integer as first argument.
```

Although the above example had only one catch clause, you can have many catch clauses associated with a single try block. Here's an example:

```
// In Source Packet in file except/ex1/VirtualCafe.java
class VirtualCafe {
    public static void serveCustomer(VirtualPerson cust,
        CoffeeCup cup) {
        try {
            cust.drinkCoffee(cup);
            System.out.println("Coffee is just right.");
        }
```

```
        catch (TooColdException e) {
            System.out.println("Coffee is too cold.");
            // Deal with an irate customer...
        }
        catch (TooHotException e) {
            System.out.println("Coffee is too hot.");
            // Deal with an irate customer...
        }
    }
}
```

If any code inside a try block throws an exception, its catch clauses are examined in their order of appearance in the source file. For example, if the try block in the above example throws an exception, the catch clause for TooColdException will be examined first, then the catch clause for TooHotException. During this examination process, the first catch clause encountered that handles the thrown object's class gets to "catch" the exception. The ordering of catch-clause examination matters because it is possible that multiple catch clauses of a try block could handle the same exception.

catch clauses indicate the type of abnormal condition they handle by the type of exception reference they declare. In the example above, the catch clauses declare exception type TooColdException and TooHotException. Had a single catch clause declared a TemperatureException, a thrown TooColdException or TooHotException still would have been caught, because TemperatureException is the superclass of both these classes. In the object-oriented way of thinking, a TooColdException is a TemperatureException, therefore, a catch clause for TemperatureException also will catch a thrown TooColdException. An example of this is shown below:

```
// In Source Packet in file except/ex2/VirtualCafe.java
class VirtualCafe {
    public static void serveCustomer(VirtualPerson cust,
        CoffeeCup cup) {
        try {
            cust.drinkCoffee(cup);
            System.out.println("Coffee is just right.");
```

```
        }
        catch (TemperatureException e) {
            // This catches TooColdException, TooHotException,
            // as well as TemperatureException.
            System.out.println("Coffee is too cold or too hot.");
            // Deal with an irate customer...
        }
    }
}
```

Multiple catch clauses could handle the same exception because you may, for example, declare two catch clauses, one for TooColdException and another for TemperatureException. In this case, however, you must place the catch clause for TooColdException above the one for TemperatureException, or the source file won't compile. If a catch clause for TemperatureException could be declared before a catch clause for TooColdException, the first catch clause would catch all TooColdExceptions, leaving nothing for the second catch clause to do. The second catch clause would never be reached. The general rule is: subclass catch clauses must precede superclass catch clauses. Here's an example of both orders, only one of which compiles:

5.3. Check yourself

1) Explain the exception hierarchy in java

2) What is Runtime Exception or unchecked exception?

3) What is checked exception?

4) What is difference between Error and Exception?

5) What is throw keyword?

6) What is use of throws keyword?

7) Is there anything wrong with the following exception handler as written? Will this code compile?

```
try {
} catch (Exception e) {
} catch (ArithmeticException a) {
```

}

8)What exception types can be caught by the following handler?

catch (Exception e) {}

Why is it a poor implementation of Exception handling.

9) Is the following code legal?

try { }

finally { }

6. Error Zone - commonly made mistakes

Basic problems with exception handling

Some examples of basic programming mistakes regarding exception handling are:

•Swallowing exceptions;

•Logging the same exception in multiple places;

•Exceptions that are too general.

Below are some of the generally accepted principles of exception handling:

•If you can't handle an exception, don't catch it;

•If you catch an exception, don't swallow it;

•Catch an exception as close as possible to its source;

•Log an exception where you catch it, unless you plan to rethrow it;

•Structure your methods according to how fine-grained your exception handling must be;

•Use as many typed exceptions as you need, particularly for application exceptions.

Item 1 is obviously in conflict with item 3. The practical solution is a trade-off between how close to the source you catch an exception and how far you let it fall before you've completely lost the intent or content of the original exception.

Exception handling in large enterprise-class systems

In large systems a whole different kind of problems arise.

•Breaking encapsulation. Methods are implemented that throw exceptions that make no sense to the caller of that method;

•Loss of information. Wrapping the exceptions that make no sense in another general exception has the side effect of making it difficult to detect the distinction between different problems programmatically.

•Information overload. Letting the original exceptions propagate up the call stack leads to

incoherent sets of exceptions in the throws clauses of methods, until developers get fed up and introduce “throws Exception”.

•Overriding a method from a superclass that does not throw exceptions. This could be a valid situation when there is no sensible reason to the caller why the method should fail.

Imagine a method which receives a checked exception but it does not know how to handle it and the method can't declare the exception as 'checked' because it does not make sense to the caller of this method.

The method could do one of four things:

•The exception is a fault and should be hidden. The method should wrap the exception in a SystemException (a subclass of RuntimeException), effectively letting the exception disappear and resurface in the main routine that will log it and possibly display it to the user;

•The exception is a contingency and the method will handle it;

•The exception is a contingency and needs to be translated to something that the caller of the method understands. The meaning of the exception does make sense to the method but not to the caller because it is simply of the wrong class. The method should wrap the exception in a new exception that is of the right class.

•The exception is a contingency but can't be handled and can't be translated, so it must be treated as a fault. The method should wrap the exception in a SystemException (a subclass of RuntimeException), effectively letting the exception disappear and resurface in the main routine that will log it and possibly display it to the user.

So here are the additional rules:

1.The exceptions that a method declares in its throws clause should make sense to the caller of that method; (was: “It is the responsibility of the Class Designer to identify issues that would result in a checked exception being thrown from a class method. Those reviewing the class design check that this has been done correctly. Exception specifications are not changed during implementation without first seeking agreement that the class design is in error.”)

2.Exceptions that propagate from public methods are expected to be of types that belong to the package containing the method;

3.Within a package there are distinct types of exceptions for distinct issues;

4.If a checked exception is thrown (to indicate an operation failure) by a method in one package it is not to be propagated by a calling method in a second package. Instead the exception is caught and "translated". Translation converts the exception into: an appropriate return status for the method, a checked exception appropriate to the calling package or an unchecked exception recognised by the

system. (Translation to another exception type frequently involves "wrapping".)

5.Empty catch-blocks are not used to "eat" or ignore exceptions. In the rare cases where ignoring an exception is correct the empty statement block should contain a comment that makes the reasoning behind ignoring the exception clear.

7. Best Practices

a)Use Specific Exceptions – Base classes of Exception hierarchy doesn't provide any useful information, that's why Java has so many exception classes, such as IOException with further sub-classes as FileNotFoundException, EOFException etc. We should always throw and catch specific exception classes so that caller will know the root cause of exception easily and process them. This makes debugging easy and helps client application to handle exceptions appropriately.

b) Throw Early or Fail-Fast – We should try to throw exceptions as early as possible. Consider above processFile() method, if we pass null argument to this method we will get following exception.

```
Exception in thread "main" java.lang.NullPointerException
    at java.io.FileInputStream.<init>(FileInputStream.java:134)
    at java.io.FileInputStream.<init>(FileInputStream.java:97)
    at
com.journaldev.exceptions.CustomExceptionExample.processFile(CustomExc
eptionExample.java:42)
    at
com.journaldev.exceptions.CustomExceptionExample.main(CustomExceptionE
xample.java:12)
```

c) While debugging we will have to look out at the stack trace carefully to identify the actual location of exception. If we change our implementation logic to check for these exceptions early as below

```
private static void processFile(String file) throws MyException {
        if(file == null) throw new MyException("File name can't be
null", "NULL_FILE_NAME");
//further processing
}
```

d) Then the exception stack trace will be like below that clearly shows where the exception has occurred with clear message.

```
com.journaldev.exceptions.MyException: File name can't be null
    at
com.journaldev.exceptions.CustomExceptionExample.processFile(Custom
ExceptionExample.java:37)
    at
com.journaldev.exceptions.CustomExceptionExample.main(CustomExcepti
onExample.java:12)
```

e) Catch Late – Since java enforces to either handle the checked exception or to declare it in method signature, sometimes developers tend to catch the exception and log the error. But this practice is harmful because the caller program doesn't get any notification for the exception. We should catch exception only when we can handle it appropriately. For example, in above method I am throwing exception back to the caller method to handle it. The same method could be used by other applications that might want to process exception in a different manner. While implementing any feature, we should always throw exceptions back to the caller and let them decide how to handle it.

f) Closing Resources – Since exceptions halt the processing of program, we should close all the resources in finally block or use Java 7 try-with-resources enhancement to let java runtime close it for you.

g) Logging Exceptions – We should always log exception messages and while throwing exception provide clear message so that caller will know easily why the exception occurred. We should always avoid empty catch block that just consumes the exception and doesn't provide any meaningful details of exception for debugging.

h) Single catch block for multiple exceptions – Most of the times we log exception details and provide message to the user, in this case we should use java 7 feature for handling multiple exceptions in a single catch block. This approach will reduce our code size and it will look cleaner too.

i) Using Custom Exceptions – It's always better to define exception handling strategy at the design time and rather than throwing and catching multiple exceptions, we can create a custom exception with error code and caller program can handle these error codes. Its also a good idea to create a utility method to process different error codes and use it.

j) Naming Conventions and Packaging – When you create your custom exception, make sure it ends with Exception so that it will be clear from name itself that it's an exception. Also make sure to package them like it's done in JDK, for example IOException is the base exception for

all IO operations.

k) Use Exceptions Judiciously – Exceptions are costly and sometimes it's not required to throw exception at all and we can return a boolean variable to the caller program to indicate whether an operation was successful or not. This is helpful where the operation is optional and you don't want your program to get stuck because it fails. For example, while updating the stock quotes in database from a third party webservice, we may want to avoid throwing exception if the connection fails.

l) Document the Exceptions Thrown – Use javadoc `@throws` to clearly specify the exceptions thrown by the method, it's very helpful when you are providing an interface to other applications to use.

8. Summary or Take Away Points

A program can use exceptions to indicate that an error occurred. To throw an exception, use the throw statement and provide it with an exception object — a descendant of Throwable — to provide information about the specific error that occurred. A method that throws an uncaught, checked exception must include a throws clause in its declaration.

A program can catch exceptions by using a combination of the try, catch, and finally blocks.

•The try block identifies a block of code in which an exception can occur.
•The catch block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
•The finally block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the try block.
The try statement should contain at least one catch block or a finally block and may have multiple catch blocks.

The class of the exception object indicates the type of exception thrown. The exception object can contain further information about the error, including an error message. With exception chaining, an exception can point to the exception that caused it, which can in turn point to the exception that caused it, and so on

9. References

http://www.ntu.edu.sg/home/ehchua/programming/java/J5a_ExceptionAssert.html
http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html
http://www.tutorialspoint.com/java/java_exceptions.htm
http://www.javaworld.com/article/2076700/core-java/exceptions-in-java.html
http://pages.cs.wisc.edu/~cs536-1/readings/JAVA/Exceptions.html
http://tutorials.jenkov.com/java-exception-handling/index.html
http://www.javatpoint.com/exception-handling-and-checked-and-unchecked-exception