Contents

Exceptions

Exceptions are a way for the programmer to capture expected or unexpected events that might occur during the duration of the program. Exceptions provide a way to safely recover from these unusual cases or take another execution path in the program. In java there are 5 keywords which are used in the context of exceptions `try catch throw throws finally` - these words are used to both define an exception handling case and emit an exception.

All exceptions in java are a subclass of `Throwable`, then there are two types of sub-classes, which are - `Exception` and `RuntimeException`, both are used in different scenarios, but briefly

- `Exception` - inheriting from `Throwable` and all other normal types of exceptions inherit from it
- `Error` - inheriting from `Throwable`, meant to represent a very serious exceptions or errors in the program's execution flow, usually coming from the run time not the actual program being executed. These are not supposed to be caught by the program and are very much unrecoverable

```
                           Throwable
                 /                            \
             Exception                         Error
        /            \                    /          \
RuntimeException  IOException   LinkerException  CompilerException
```

`The Throwable class has an overloaded constructor which takes the human readable message as String and also another Throwable as cause, the cause object is in case we would like to wrap or create a new exception in certain cases and retain information about the initial cause of the exception in the first place, that way we can wrap the cause inside the custom exception, and now lose context.`

Exception

The exception class is further sub-classed by others such as:

- `RuntimeException` - is meant to be sub-classes by/for unrecoverable exceptions in the process which are usually not expected to be handled by the program, this however depends on the scenario and the type exception, of the execution of the program. All subclasses of `RuntimeException` are what are called `unchecked`, as mentioned, not required to be caught
- By default all other classes which inherit the `Exception` class are considered `checked` exceptions, meaning that the compiler would emit errors if a checked exception is declared to be thrown by a method, and not caught by the program and handled accordingly

`Why are RuntimeException by default not required to be handled, it is not because they are somehow special, they are actually handled by the default exception handler provided by the java runtime, the compiler recognizes that a class is a sub-class of RuntimeException and does not require the program to handle it, rather it knows that the default handler will in the end do this, but all exceptions which inherit from the Exception class are technically handled`

Error

As mentioned classes inheriting this type of class are mostly related to exceptional cases produced by the JVM run time and not by the program being executed, these exceptions are not only not expected to be caught and handled, but most often they signal some sort critical error, which is unrecoverable, the program very well might terminate execution, or not even start to begin with

Handling

The default handler in the java runtime usually prints or displays what is called a `stack trace` this is the call stack of all functions between the start of the program and the place where the exception has occurred, this way we can track what flow the program has taken to arrive at the exceptional case. The call stack contains links and references to the original source code, including file names, and line numbers which help us discover exactly where the error has occurred in the source itself

```java
class Exceptional {
    public static void main(String args[]) {
        int d, a;
        try { // monitor a block of code.
            d = 0;
            a = 42 / d;
            // there is a division by 0 above, which will trigger an
               exception
            System.out.println("This will not be printed.");
        } catch (ArithmeticException e) { // catch divide-by-zero error
            System.out.println("Division by zero.");
        } finally {
            // will execute always, with or without captured exception
            System.out.println("Finally will always execute this block");
        }
        // will execute just fine since we do not terminate the execution
        System.out.println("After catch statement.");
    }
}
```

The example above shows how a custom handler is created, instead of relying on the default one which if reached will terminate the program execution, this way we can safely `divide by 0` without forcefully killing the program

`Throwable overrides the toString()method, which means we can pass the Throwable object/exceptio to a print procedure to display a human readable error description`

Here is the actual implementation lifted from the `Throwable` class in the java default library and run time, it usually prints the message along with the name of the class, in this case the name of the `Exception`

```java
public String toString() {
    String s = getClass().getName();
    String message = getLocalizedMessage();
    return (message != null) ? (s + ": " + message) : s;
}
```

Catching

Catching an exception can be done with single or multiple catch clauses, keep in mind that the multiple catch clause has a few caveats which have to be looked at carefully. When multiple catch clauses are present they are inspected in the order they appear in.

```
public static void main(String args[]) {
    try {
        int a = 0;
        int b = 42 / a;
    } catch(Exception e) { // exception is a top level super class and
       comes first
        System.out.println("Generic Exception catch.");
    }
    catch(ArithmeticException e) { // compile time error, this is not
       reachable
        System.out.println("This is never reached.");
    }
}
```

Example above is very important to understand, we can not define super class exceptions followed by sub-class exceptions which are in the same hierarchy, otherwise the compiler will throw errors, reason is pretty obvious since catch clauses are evaluated top-down, `Exception` (which is a super class of `ArithmeticException`) will be caught first even if `ArithmeticException` is thrown, meaning that the catch below is `dead code`

```
When you use multiple catch statements, it is important to remember that exception subclasses
must come before any of their superclasses. This is because a catch statement that uses a
superclass will catch exceptions of that type plus any of its subclasses.
```

Nesting

It is allowed to nest try - catch pairs within each other, it is also allowed to skip the catch blow of a try statement, meaning that the outer try - catch pair will then intercept whatever exception is emitted from the inner try (without catch).

```
public static void main(String args[]) {
    try {
        int a = args.length;
        int b = 42 / a; // division by zero if no args are present
        System.out.println("a = " + a);
        try { // nested try block that only handles ArithmeticException
            if(a==1) a = a/(a-a); // division by zero if no args are
               present
            if(a==2) { // enough arguments, no division by zero expected
                int c[] = { 1 };
                c[42] = 99; // generate an out-of-bounds exception
            }
        } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out-of-bounds: " + e);
        }
    } catch(ArithmeticException e) {
        System.out.println("Divide by 0: " + e);
    }
}
```

In the example above, the inner try-catch handles only the index out of bounds case, which means that if the inner one throws division by 0 exception, then the outer catch will actually capture this exception coming from the inner try block. Which is perfectly valid, but something to keep an eye out for

Throwing

Throwing an exception from a program is straight forward, it is required that we use the keyword `throw` which is then followed by an object / instance of a `Throwable`, which can be any custom or existing exception already defined in the run time.

```
throw new NullPointerException("demo");
```

Throws

A similar but very different keyword, which defines if a given method throws an exception but does not handle it. Now it is very important to note that if a method throws an exception which is `RuntimeException` or a sub-class of it, it needs no notify the caller with `throws` however if that is not the case and the exception it throws is an instance of `Exception` then it is considered by default to be checked exception, which means the method `must` specify all checked exceptions that it does not handle, otherwise `compile time error` occurs

```
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException { //
       IllegalAccessException is sub-class of Exception, not
       RuntimeException, correct
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    static void throwTwo() { // and here, NullPointerException is
       sub-class of RuntimeException no need to specify throws, NO compile
       time error
        System.out.println("Inside throwTwo.");
        throw new NullPointerException("demo");
    }
    static void throwThree() { // compile time error, this is not allowed
       since ParseException is a sub-class of Exception, needs `throws`
       clause
        System.out.println("Inside throwThree.");
        throw new ParseException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Finally

The finally clause provides a way to always execute a piece of code after try / catch block, regardless if the code within has thrown or not an exception, this is very useful to make sure we can release stored resources, reset temporary state and so on. `The finally block will execute right after the try or/and catch and before the code after the finally block`

```
    public static void main(String args[]) {
        try {
            throwsSomething();
        } catch (Exception e) {
            System.out.println("Caught " + e);
        } finally {
            System.out.println("Finally");
        }
        System.out.println("After");
    }
```

Chaining

Since java 1.4, the `Throwable` class has been expanded to support a way to provide a cause for an exception, as already mentioned, when constructing an exception we can use one of the overloaded constructors to provide a `cause` the cause is another instance of `Throwable` which is the cause of the current exception, usually this is done when we want to create our own wrapping exception with custom message and description, however still provide means of discovering and recovering the original information for the cause of our own exception.

```
public static void main(String args[]) {
    try {
        // do some work which might throw some generic exception, which we
           would like to wrap around our custom one
    } catch (NullPointerException e) {
        throw new CustomSpecialDomainException("Message", e); // do not
           lose information about the actual cause
    }
}
```

Autoclose

After java 8, it is also now possible to automatically close resources without having to worry about handling the case in a finally block, these have to implement the AutoCloseable interface

`The try-with-resources statement can be used only with those resources that implement the AutoCloseable interface defined by java.lang. This interface defines the close( )method. AutoCloseable is inherited by the Closeable interface in java.io.`

It is important to understand that the resource declared in the try statement is implicitly final. This means that you can't assign to the resource after it has been created. Also, the scope of the resource is limited to the try-with-resources statement.

```
// `fout` is not accessible outside of the try block and is also declared
   as final
try(FileOutputStream fout = new FileOutputStream("file.txt");
    FileOutputStream foutTwo = new FileOutputStream("file2.txt")) {
    // do something with the two file streams
```

```java
} catch(IOException e) {
    // capture potential issues with opening
}
// after exiting the try block, `fout` stream will be closed automatically
```