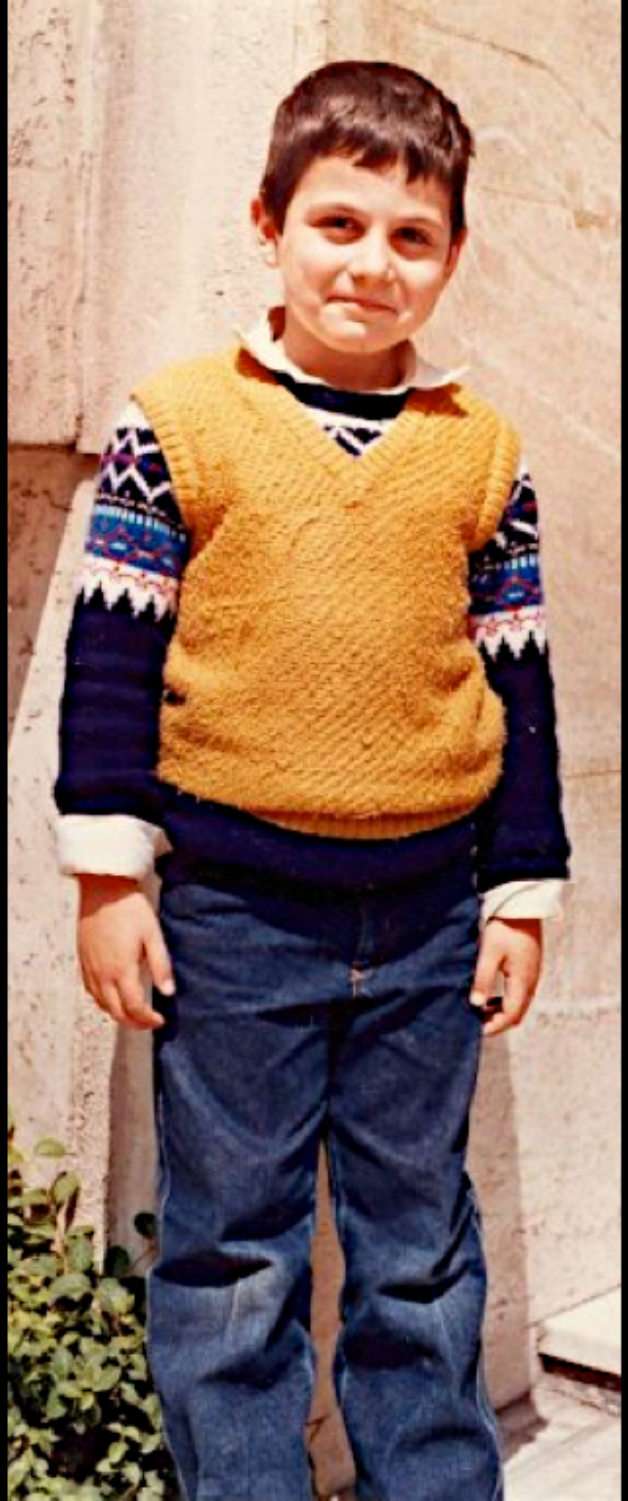
JAVA EXCEPTION HANDLING
BEST PRACTICES

Improved Second Version

LEMİ ORHAN ERGİN

@lemiorhan lemiorhanergin.com @lemiorhan

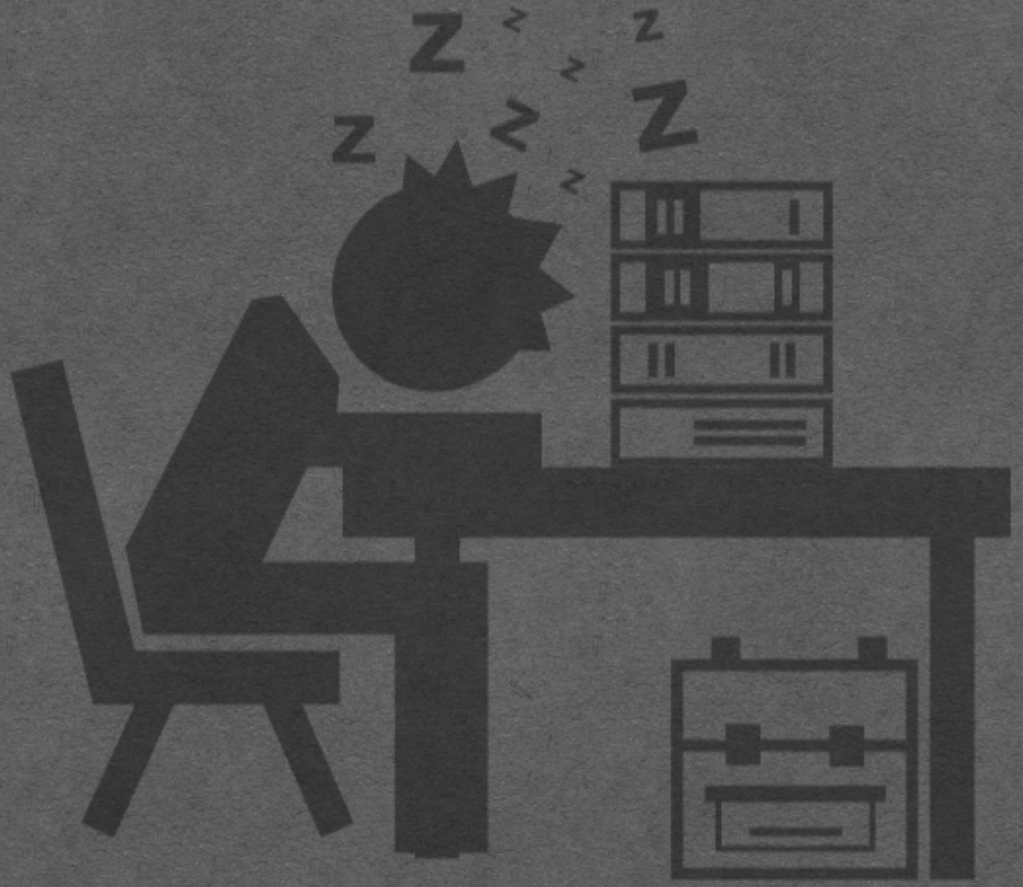


LEMİ ORHAN ERGİN

CERTIFIED SCRUM MASTER (CSM, PSM 1)
KANBAN AND XP PRACTITIONER

Developing since 2001
Principal Software Engineer at Sony Europe
Agile Software Craftsman
GittiGidiyor/eBay Alumni
Active Member of Agile Turkey AGILETURKEY.ORG
Founder of Software Craftsmanship TR SCTURKEY.ORG
Trainer on Agile, ALM, Scrum, Git, etc.
Doing code reviews since 2005
Public Speaker
Blogger about Agile AGILISTANBUL.COM
Passionate Developer

AGENDA



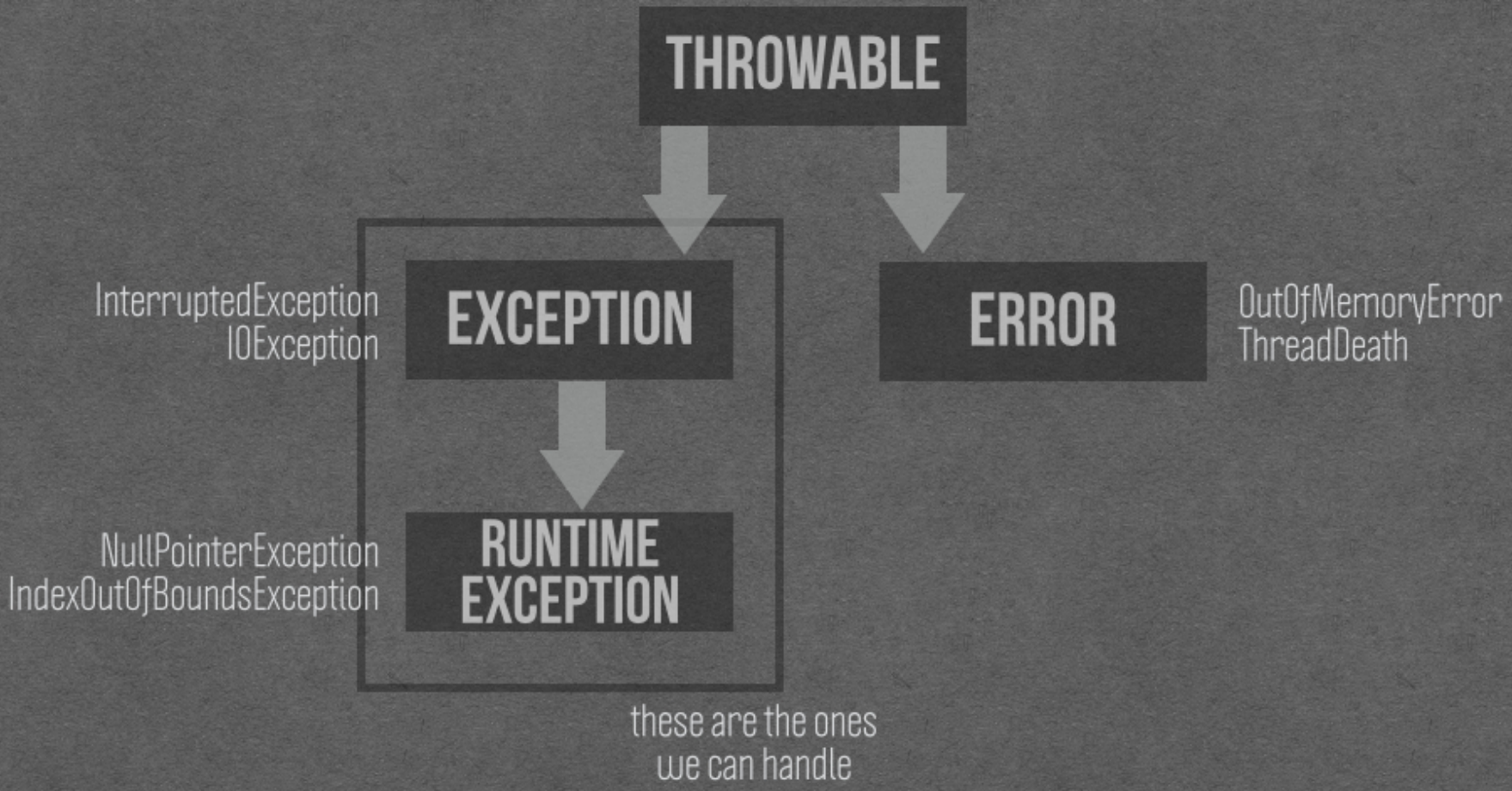
THE BASICS
TIPS & TRICKS
BEST PRACTICES
ENDLESS DEBATE

I ASSUME YOU ALL KNOW JAVA AND EXCEPTION HANDLING, RIGHT?

LET'S REMEMBER FIRST

EXCEPTIONS

Exceptions are objects that define an abnormal condition that interrupts the normal flow of the program



CHECKED EXCEPTIONS

Instance of Exception class
Instance of a subclass of Exception
Not an instance of RuntimeException
Compile-time checking

```
public void readUrl(String url) throws IOException, MalformedURLException {
    // read contents of URL using
}

public void readUrls(String[] urls) {
    for (String url : urls) {
        try {
            readUrl(url);
        } catch(IOException ioe) {
            // handle exception
        } catch (MalformedURLException me) {
            // handle exception
        }
    }
}
```

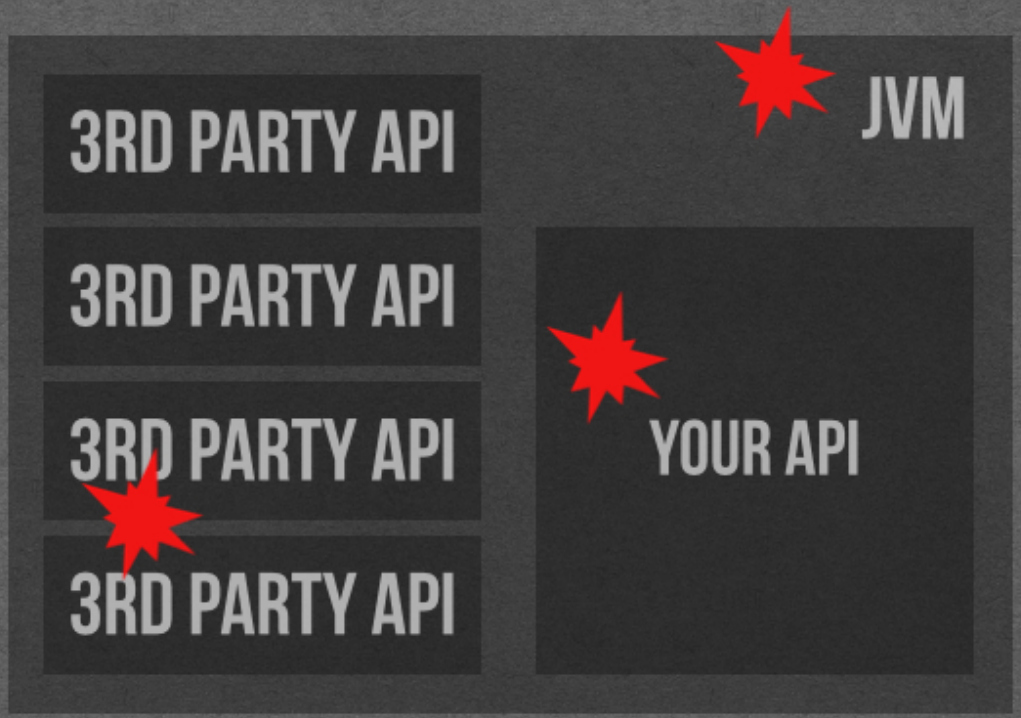
UNCHECKED EXCEPTIONS

Instance of RuntimeException class
No compiler check
No declaration of exceptions

```
public void readUrl(String url) {
    if (url == null)
        throws new RuntimeException("Error reading url");
}

public void readUrls(String[] urls) {
    for (String url : urls) {
        readUrl(url);
    }
}
```

EXCEPTION HANDLING

Designing exception handling for an application means deciding how the application should react to errors to assure the application stays healthy



TIPS AND TRICKS

TIP

CATCH AND FINALLY BLOCKS ARE OPTIONAL, AT LEAST ONE HAS TO BE AVAILABLE WITH TRY

AFTER CATCH BLOCK IS EXECUTED, EVEN IF AN EXCEPTION IS THROWN, EXECUTION JUMPS TO FINALLY BLOCK

```
try {
    // do some stuff here
    // that throws Exception1
} catch (Exception1 se) {
    // do some stuff here
    // that throws Exception2
} finally {
    // do some stuff here
    // that throws Exception3
}
```

TIP

IF EXCEPTION IS THROWN IN FINALLY BLOCK, THE EXCEPTION IS PROPAGATED UP THE CALL STACK EVEN IF THE EXCEPTION THROWN FROM THE TRY BLOCK IS PROBABLY MORE RELEVANT

```
try {
    // do some stuff here
    // that throws SomeException
} finally {
    // do some stuff here
    // that throws AnotherException
}
```

TIP

YOU DO NOT HAVE TO THROW IMMEDIATELY AFTER CREATING IT

```
Exception e = new Exception("An error ocurred");

// Do some stuff to enrich exception information
// like adding request id or error id

throw e;
```

TIP

YOU CAN CONVERT CHECKED EXCEPTION INTO UNCHECKED EXCEPTION AND VICE VERSA

```
try {
    riskyOperation();
} catch (IOException ioe) {
    throw new CustomRuntimeException(ioe);
}
```

TIP

CATCH BLOCK ORDER IS IMPORTANT FOR THE EXCEPTIONS OF SUBCLASSES AND CLASSES

```
try {
    readUrl(url);
} catch(Exception e) {
    // handle exception
} catch (MalformedURLException me) {
    // cannot access here
} catch (IOException ioe) {
    // cannot access here
}
```

TIP

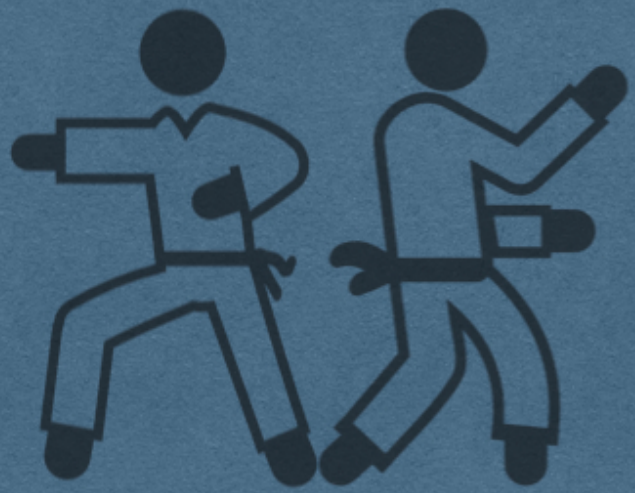
IN A MULTI-THREADED APP, EACH THREAD WILL HAVE ITS OWN CALL STACK. EXCEPTIONS THROWN DURING THE EXECUTION OF ONE THREAD DOES NOT INTERRUPT THE EXECUTION OF OTHER THREADS.

TIP

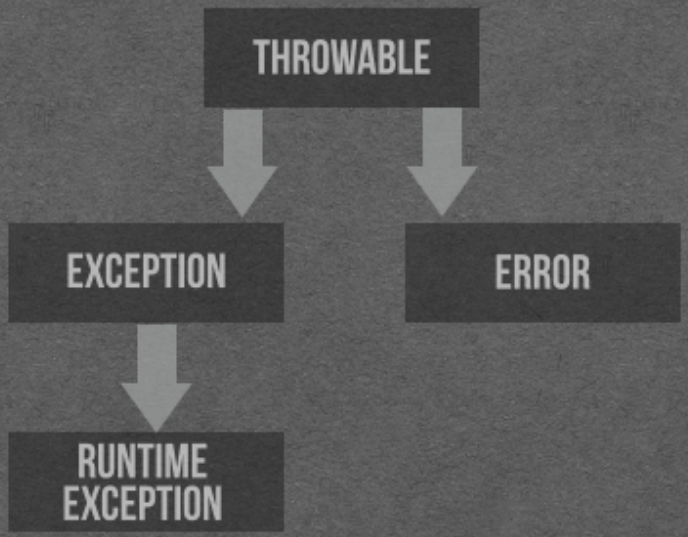
4 WAYS OF PROPAGATION

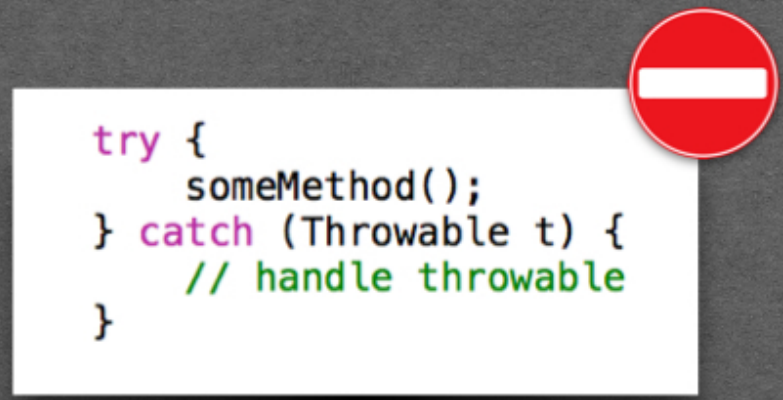
BY DECLARATION
BY RE-THROWING
BY WRAPPING
BY REPLACING

BEST PRACTICES



NEVER CATCH THROWABLE OR ERROR





```
try {
    someMethod();
} catch (Throwable t) {
    // handle throwable
}
```

Java errors are also subclasses of the Throwable.
Errors are irreversible conditions that can not be handled by JVM itself.
And for some JVM implementations, JVM might not actually even invoke your catch clause on an Error

USE NEW FEATURES OF JAVA7 TO HAVE CLEANER CODE

CATCHING MULTIPLE EXCEPTIONS

```
catch (IOException|SQLException ex) {
    logger.log(ex);
}
```

Since the multi catch block doesn't know exactly what type of exception it is catching, you can only reference the caught exception as an instance of Exception, or any other common superclass the exceptions in the multi catch block have.

USE NEW FEATURES OF JAVA7 TO HAVE CLEANER CODE

AUTOMATIC RESOURCE MANAGEMENT WITH TRY-WITH-RESOURCES

```
private static void printFileJava7() throws IOException {

    try( FileInputStream input = new FileInputStream(" file.txt");
        BufferedInputStream bufferedInput = new BufferedInputStream(input) ) {
        int data = bufferedInput.read();
        while (data != -1) {
            System.out.print(( char) data);
            data = bufferedInput.read();
        }
    }
}
```

- Catch and Finally blocks are not required
- Any class implementing AutoClosable can be used inside try block
- The resources will be closed in reverse order of creation
- If exception is thrown both from inside try, the exception thrown
- If exception is thrown while the stream is closing, the exception is suppressed

USE NEW FEATURES OF JAVA8 TO HAVE CLEANER CODE

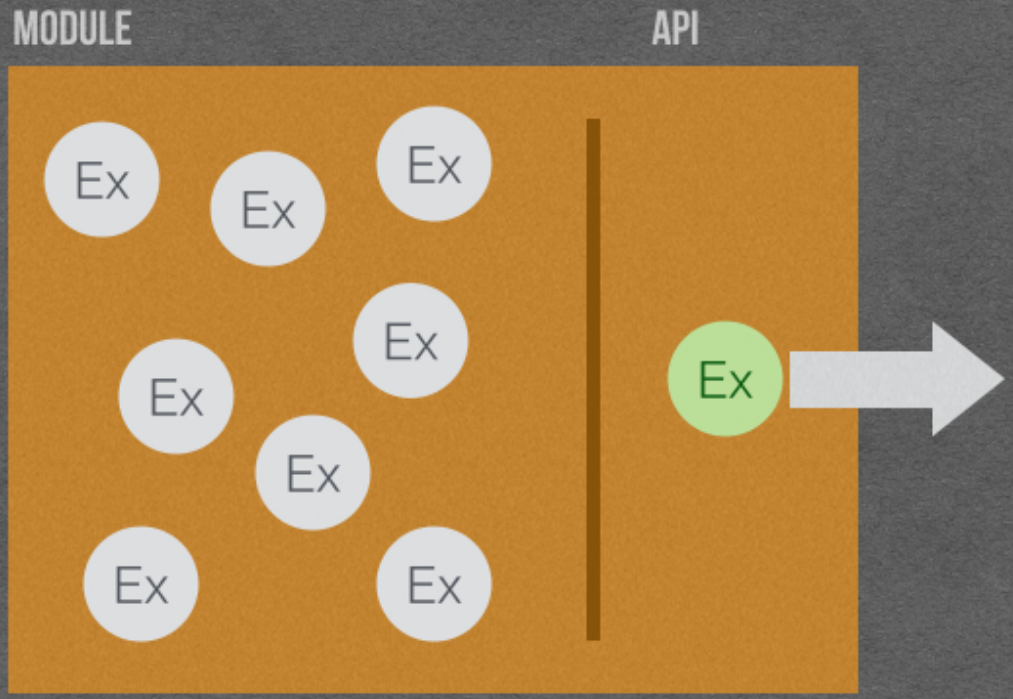
JAVA6 STYLE

```
public Response fulfill(HttpMethod method, String requestUri) {
    URI uri;
    try {
        uri = new URI(requestUri);
    } catch (URISyntaxException e) {
        throw new RuntimeException(e);
    }
    Request request = new Request(method, uri);
    return fulfill(request);
}
```

USING LAMBDA IN JAVA8

```
public Response fulfill(HttpMethod method, String requestUri) {
    URI uri = Throwables.propagate(() -> new URI(requestUri),
                                   (e) -> new IllegalArgumentException("Problem!", e));
    Request request = new Request(method, uri);
    return fulfill(request);
}
```

CONVERT EXCEPTIONS TO PROVIDE BETTER ABSTRACTION



This is one of the technique used in many of frameworks like Spring, where most of checked exceptions are converted into unchecked exceptions. This Java best practice provides benefits, like collecting SQLExceptions in DAO layer and throwing meaningful RuntimeException to client layer.

PREFER FIXING THE ROOT CAUSE INSTEAD OF THROWING EXCEPTIONS

```
public boolean validatePackage(Package pkg){
    try{
        String msg = null;

        if (pkg == null) {
            msg = "Package is null";
        } else {
            if (pkg.isEmpty()) {
                msg = "Package " + pkg.getName() + " is not valid";
            }
        }

        if (msg != null) {
            RuntimeException ex = new RuntimeException(msg);
            if (logger.isErrorEnabled()) {
                logger.error(msg, ex);
            }
            throw ex;
        }
    } catch (Exception e){
        logger.error("Error when validating pkg" , e);
        return false;
    }
    return true;
}
```

NEVER USE EXCEPTIONS FOR FLOW CONTROL

PREFER FIXING THE ROOT CAUSE INSTEAD OF THROWING EXCEPTIONS

SIMPLE SOLUTION

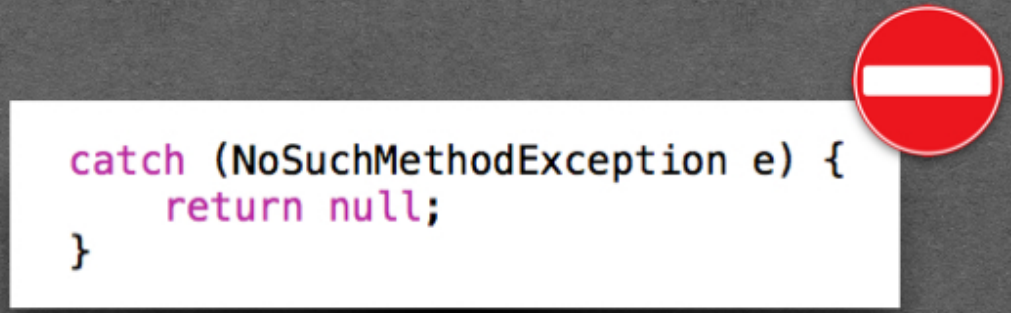
```
public boolean validatePackage(Package pkg){
    if (pkg == null) {
        logger.warn("Package is null");
        return false;
    }
    else if (pkg.isEmpty()) {
        logger.warn("Package " + pkg.getName() + " is empty");
        retun false;
    }
    return true;
}
```

YOU CAN PREFER TO RETURN OBJECTS INSTEAD OF THROWING EXCEPTIONS

```
public class OperationResult {

    public boolean isSuccess;

    public List result;

    public Error error;
}
```

In some cases, you might want to return a result object containing all the information about the result or the error instead of the exception.

NEVER SWALLOW THE EXCEPTION IN CATCH BLOCK



```
catch (NoSuchMethodException e) {
    return null;
}
```

Doing this not only return "null" instead of handling or re-throwing the exception, it totally swallows the exception, losing the cause of error forever. And when you don't know the reason of failure, how you would prevent it in future? Never do this !!

DECLARE THE SPECIFIC CHECKED EXCEPTIONS THAT YOUR METHOD CAN THROW

```
public void foo() throws Exception {
}
```

```
public void foo() throws SpecificException1, SpecificException2 {
}
```

Always avoid doing this as in above code sample. It simply defeats the whole purpose of having checked exception. Declare the specific checked exceptions that your method can throw.

AVOID TOO MANY CHECKED EXCEPTIONS IN METHOD DECLARATION

```
public void foo() throws SpecificException1, SpecificException2, SpecificException3,
                         SpecificException4, SpecificException5, SpecificException6,
                         SpecificException7, SpecificException8, SpecificException9 {
}
```

If there are just too many such checked exceptions, you should probably wrap them in your own exception and add information to in exception message. You can also consider code refactoring also if possible.

CATCHING EXCEPTION HIDES INFORMATION
THINK TWICE BEFORE USING IT

```
try {
    someMethod();
} catch (Exception e) {
    LOGGER.error("method has failed", e);
}
```

Catching Exception means catching both checked and unchecked exceptions.
Use it if you have to handle both checked and unchecked exceptions or you have no different handling plans for specific exception types.

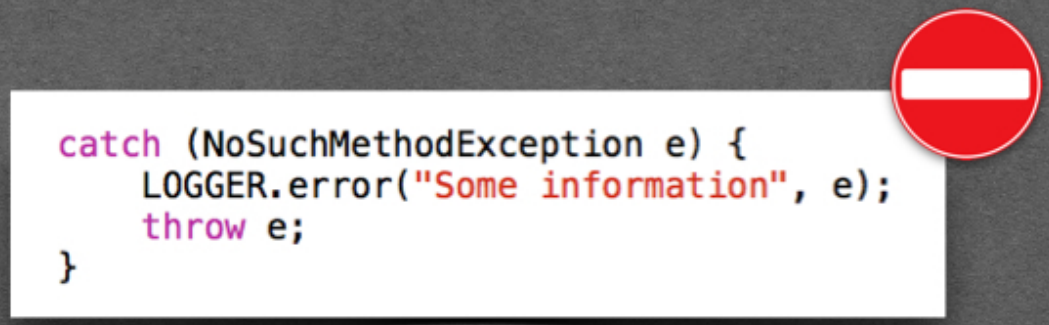
ALWAYS CORRECTLY WRAP THE EXCEPTIONS IN CUSTOM EXCEPTIONS SO THAT STACK TRACE IS NOT LOST

```
catch (NoSuchMethodException e) {
    throw new MyServiceException("Some information: " + e.getMessage());
}
```

```
catch (NoSuchMethodException e) {
    throw new MyServiceException("Some information: " , e);
}
```

Incorrect way of wrapping exceptions destroys the stack trace of the original exception, and is always wrong.

EITHER LOG THE EXCEPTION OR THROW IT BUT NEVER DO THE BOTH



```
catch (NoSuchMethodException e) {
    LOGGER.error("Some information", e);
    throw e;
}
```

Logging and throwing will result in multiple log messages in log files, for a single problem in the code, and makes life hell for the engineer who is trying to dig through the logs.

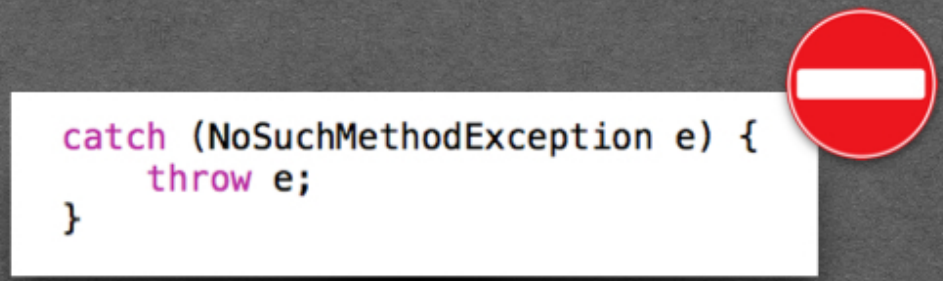
NEVER THROW ANY EXCEPTION FROM FINALLY BLOCK

```
try {
    // Throws exceptionOne
    someMethod();
} finally {
    // If finally also threw any exception,
    // the exceptionOne will be lost forever
    cleanUp();
}
```

This is fine, as long as cleanUp() can never throw any exception. In the above example, if someMethod() throws an exception, and in the finally block also, cleanUp() throws an exception, that second exception will come out of method and the original first exception (correct reason) will be lost forever.

If the code that you call in a finally block can possibly throw an exception, make sure that you either handle it, or log it. Never let it come out of the finally block.

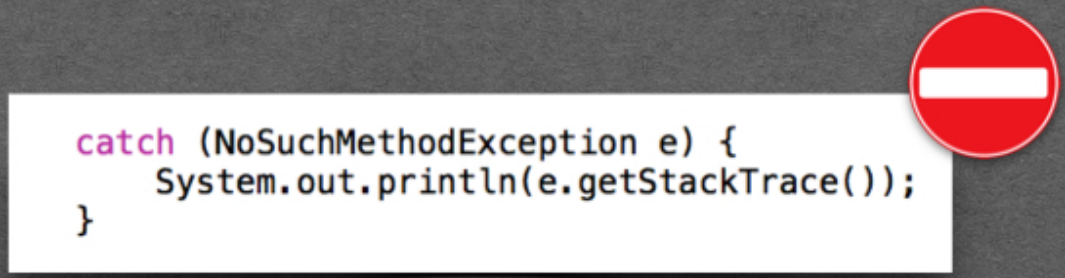
ALWAYS CATCH ONLY THOSE EXCEPTIONS THAT YOU CAN ACTUALLY HANDLE



```
catch (NoSuchMethodException e) {
    throw e;
}
```

Well this is most important concept. Don't catch any exception just for the sake of catching it. Catch any exception only if you want to handle it or, you want to provide additional contextual information in that exception. If you can't handle it in catch block, then best advice is just don't catch it only to re-throw it.

DON'T USE PRINTSTACKTRACE() STATEMENT OR SIMILAR METHODS



```
catch (NoSuchMethodException e) {
    System.out.println(e.getStackTrace());
}
```

Never leave printStackTrace() after finishing your code. Chances are one of your fellow colleague will get one of those stack traces eventually, and have exactly zero knowledge as to what to do with it because it will not have any contextual information appended to it.

CATCH AND LOG ALL EXCEPTIONS AND FOLLOW LOGGING BEST PRACTICES

LOGGING WITH MISSING INFORMATION AND WRONG SEVERITY LEVEL

```
try {
    someMethod();
} catch (OperationException e) {
    LOGGER.debug("Error occurred");
}
```

```
try {
    someMethod();
} catch (OperationException e) {
    LOGGER.info("Error occurred", e);
}
```

PROPER WAY OF LOGGING THE EXCEPTIONS

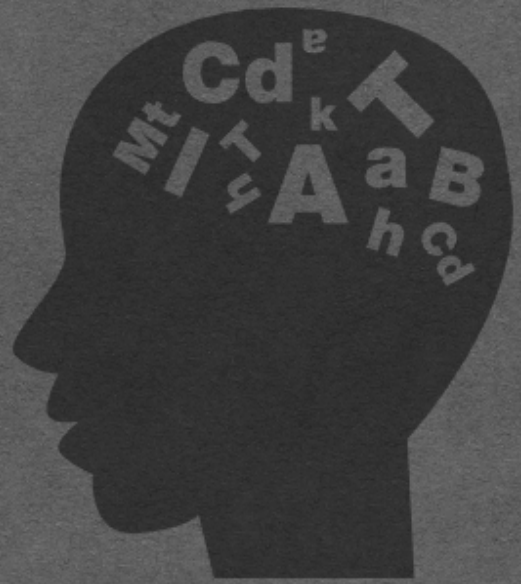
```
try {
    someMethod();
} catch (OperationException e) {
    LOGGER.warn("Sending cannot proceed for user {} due to operation errors", user.getName(), e);
}
```

- proper severity level
- correct operation type
- information about the context
- original exception

DO NOT MESS ACCESS/ACTION LOGS WITH ERROR STACK TRACES

Stacktraces mess-up logs and have negative impact on diagnosis and troubleshooting.

Keep access/action logs in separate log files and keep them clean



ALWAYS INCLUDE ALL INFORMATION ABOUT AN EXCEPTION IN SINGLE LOG MESSAGE

```
try {
    someMethod();
} catch (OperationException e) {
    LOGGER.debug("some message");
    // handle exception
    LOGGER.debug("some another message");
}
```

Using a multi-line log message with multiple calls to LOGGER.debug() may look fine in your test case, but when it shows up in the log file of an app server with 400 threads running in parallel, all dumping information to the same log file, your two log messages may end up spaced out 1000 lines apart in the log file, even though they occur on subsequent lines in your code.

REMEMBER “THROW EARLY CATCH LATE” PRINCIPLE

This principle says that you will be more likely to throw it in the low-level methods, where you will be checking if values are null or not appropriate. And you will be making the exception climb the stack trace for quite several levels until you reach a sufficient level of abstraction to be able to handle the problem.

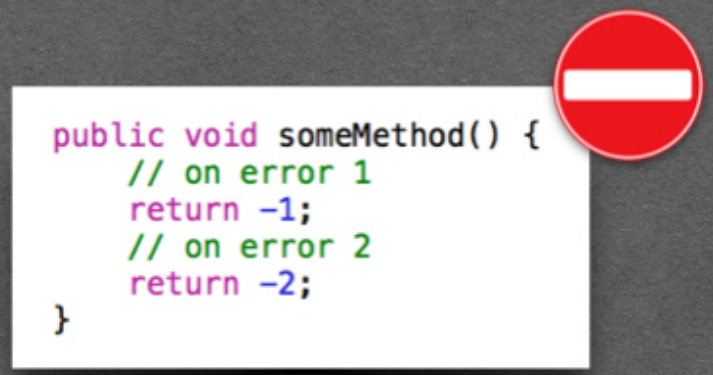
ALWAYS CLEAN UP AFTER HANDLING THE EXCEPTION

- Thread Locks
- Database connections
- Channels and Sockets
- Files and Streams
- Readers and Writers
- RowSet and ResultSets

EXCEPTION NAMES MUST BE CLEAR AND MEANINGFUL

Name your checked exceptions stating the cause of the exception. You can have your own exception hierarchy by extending current Exception class. But for specific errors, throw an exception like "AccountLockedException" instead of "AccountException" to be more specific.

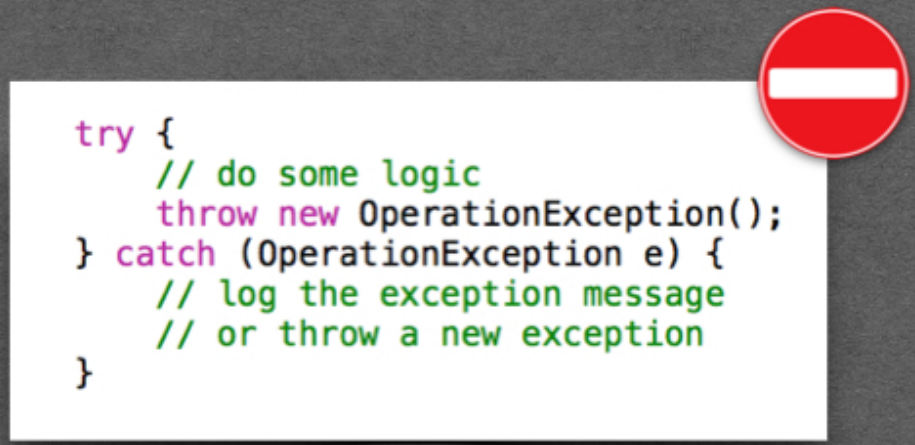
THROW EXCEPTIONS FOR ERROR CONDITIONS WHILE IMPLEMENTING A METHOD



```
public void someMethod() {
    // on error 1
    return -1;
    // on error 2
    return -2;
}
```

If you return -1, -2, -3 etc. values instead of FileNotFoundException, that method can not be understand. Use exceptions on errors.

NEVER THROW IN TRY BLOCK AND CATCH IT IMMEDIATELY



```
try {
    // do some logic
    throw new OperationException();
} catch (OperationException e) {
    // log the exception message
    // or throw a new exception
}
```

It is useless to catch the exception you throw in the try block. Do not manage business logic with exceptions. Use conditional statements instead.

ONE TRY BLOCK MUST EXIST FOR ONE BASIC OPERATION

TRY

```
try {
    ProductSku secondAvailabilityStatusSku = null;
    ProductSku thirdAvailabilityStatusSku = null;
    for (ProductSku sku : productSkuList) {
        try {
            Availability availabilityStatus = getSkuAvailability(sku);
            if (null != availabilityStatus) {
                if (availabilityStatus.equals(Availability.AVAILABLE)) {
                    logger.debug("Main available sku is found {}", sku.getMaterialName());
                    return sku;
                }
                else if (availabilityStatus.equals(Availability.NOT_IN_STOCK)) {
                    if (secondAvailabilityStatusSku == null)
                        secondAvailabilityStatusSku = sku;
                }
                else if (availabilityStatus.equals(Availability.PRE_ORDER)) {
                    if (thirdAvailabilityStatusSku == null)
                        thirdAvailabilityStatusSku = sku;
                }
            }
        } catch (Exception e) {
            logger.warn("Exception occurred while getting main sku. Exception: {}", e, e.getMessage());
            return productSkuList.get(0);
        }
    }

    if (secondAvailabilityStatusSku != null){
        logger.debug("Secondly available sku is found {}", secondAvailabilityStatusSku.getMaterialName());
        return secondAvailabilityStatusSku;
    }
    else if (thirdAvailabilityStatusSku != null) {
        logger.debug("Thirdly  available sku is found {}", thirdAvailabilityStatusSku.getMaterialName());
        return thirdAvailabilityStatusSku;
    }
    else {
        T firstSku = productSkuList.get( 0 );
        logger.debug("No available sku is found, so return first sku {}", firstSku.getMaterialName());
        return firstSku;
    }

    if (productSku == null) return null;
    TransactionalProductSku tSku = productSku.as(TransactionalProductSku.class);
    SSSTransactionalInformation sssTransactionalInformation = (SSSTransactionalInformation)
            tSku.getTransactionalInformation(TransactionalInformationType.SSS);
    if (sssTransactionalInformation == null) return null;
    Availability availabilityStatus = sssTransactionalInformation.getAvailabilityInformation().getAvailability();

    Comparator<ProductSku> c = new Comparator<ProductSku>() {
        public int compare(ProductSku o1, ProductSku o2) {
            Double p1 = getTransactionalProductSkuPrice(o1);
            Double p2 = getTransactionalProductSkuPrice(o2);

            if (p1 == 0 && p2 == 0)
                return o1.getCatalogName().compareTo(o2.getCatalogName());
            else if (p1 == 0 && p2 > 0)
                return 1;
            else if (p1 > 0 && p2 == 0)
                return -1;
            else
                return p1.compareTo(p2);
        }
    } catch(exception e) {
        logger.log("An issue occurs while detecting packages", e);
    }
```

CATCH

Granularity is very important.
One try block must exist for one basic operation.
So don't put hundreds of lines in a try-catch statement.

USE TEMPLATE METHODS FOR REPEATED TRY-CATCH

```
class DBUtil{
    public static void closeConnection(Connection conn){
        try{
            conn.close();
        } catch(SQLException ex){
            throw new RuntimeException("Cannot close connection", ex);
        }
    }
}
```

```
public void dataAccessCode() {
    Connection conn = null;
    try{
        conn = getConnection();
        ....
    } finally{
        DBUtil.closeConnection(conn);
    }
}
```

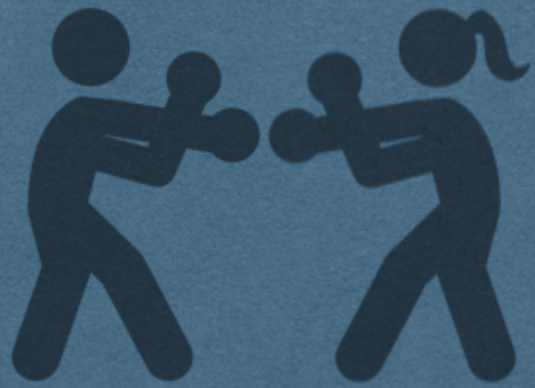
There is no use of having a similar catch block in 100 places in your code. It increases code duplicity which does not help anything. Use template methods for such cases.

CATCH ALL EXCEPTIONS BEFORE THEY REACH UP TO THE UI

You have to catch all exceptions before they reach up to the UI.
Don't make your user sad. This means on the "highest level" you want to catch anything that happened further down.

ENDLESS DEBATE FOR YEARS

USE CHECKED EXCEPTION IF NEEDED



ALWAYS USE UNCHECKED EXCEPTIONS

USE CHECKED EXCEPTION IF NEEDED

Checked ones are for recoverable errors
Unchecked ones are system/programming errors only
Without compiler enforcement, developers forget to handle them
You can catch unchecked ones and throw checked if needed

ALWAYS USE UNCHECKED EXCEPTIONS

Unchecked ones are not system errors like NullPointerException
Unchecked ones also need to be catched and fixed
No one wants to shut down in case of NullPointerException
Developers rely too much on compiler and compile time exceptions
Checked exceptions make method declarations cluttered

DETAILS WILL MAKE YOU A GOOD DEVELOPER

REFERENCES

http://www.amazon.com/Java-Exception-Handling-Jakob-Jenkov-ebook/dp/B00BG9FGFI/ref=sr_1_1

REFERENCES

JAVA EXCEPTION HANDLING BEST PRACTICES BY LOKESH GUPTA

http://howtodoinjava.com/2013/04/04/java-exception-handling-best-practices/

15 BEST PRACTICES FOR EXCEPTION HANDLING BY CAGDAS BASARANER

http://codebuild.blogspot.co.uk/2012/01/15-best-practices-about-exception.html

10 EXCEPTION HANDLING BEST PRACTICES IN JAVA PROGRAMMING BY JAVIN PAUL

http://javarevisited.blogspot.co.uk/2013/03/0-exception-handling-best-practices-in-Java-Programming.html

LEMİ ORHAN ERGİN

lemiorhan@agilistanbul.com
Founder & Author @ agilistanbul.com

@LEMIORHAN

@LEMIORHAN

@LEMIORHAN

AGILISTANBUL.COM

LEMIORHANERGIN.COM