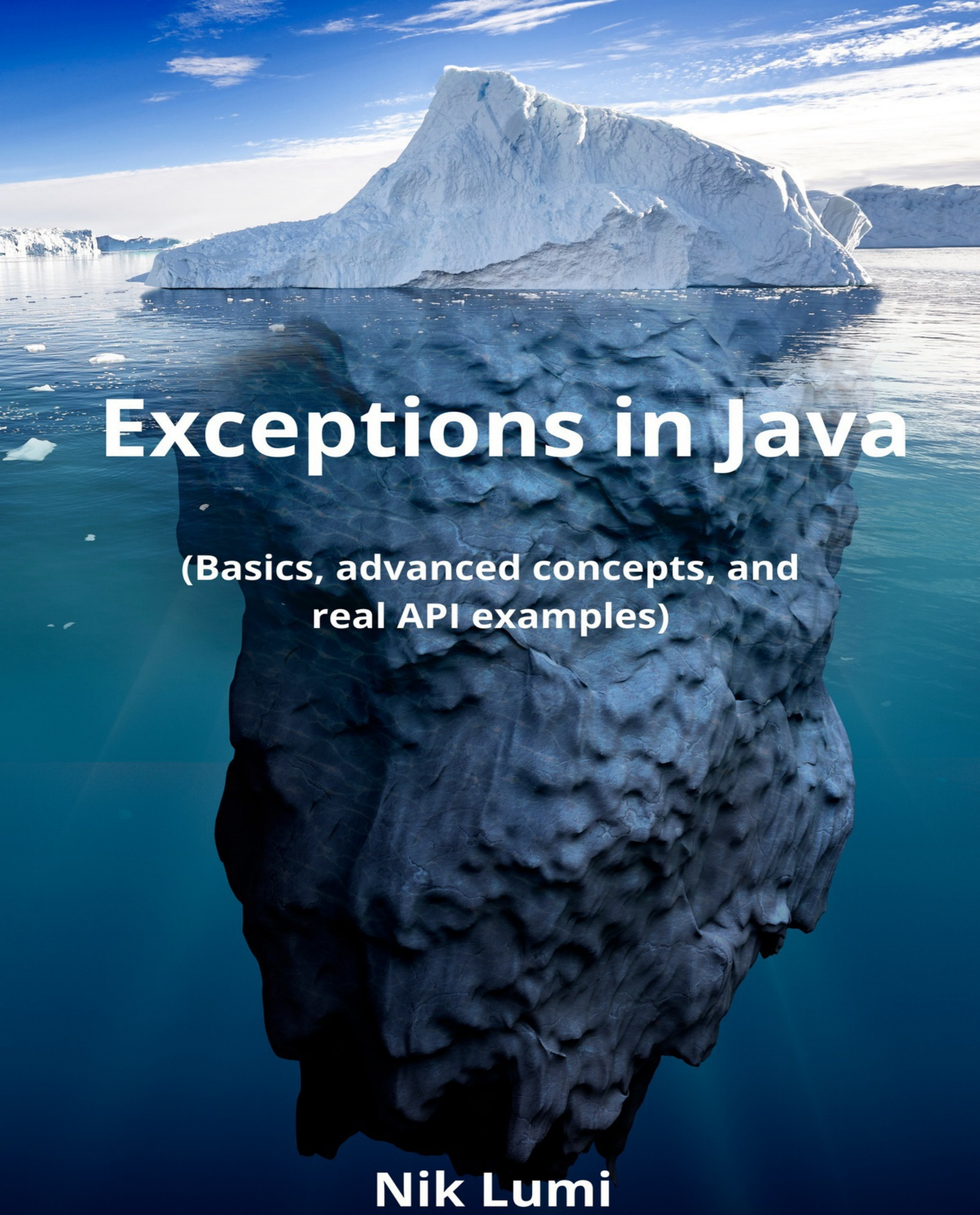
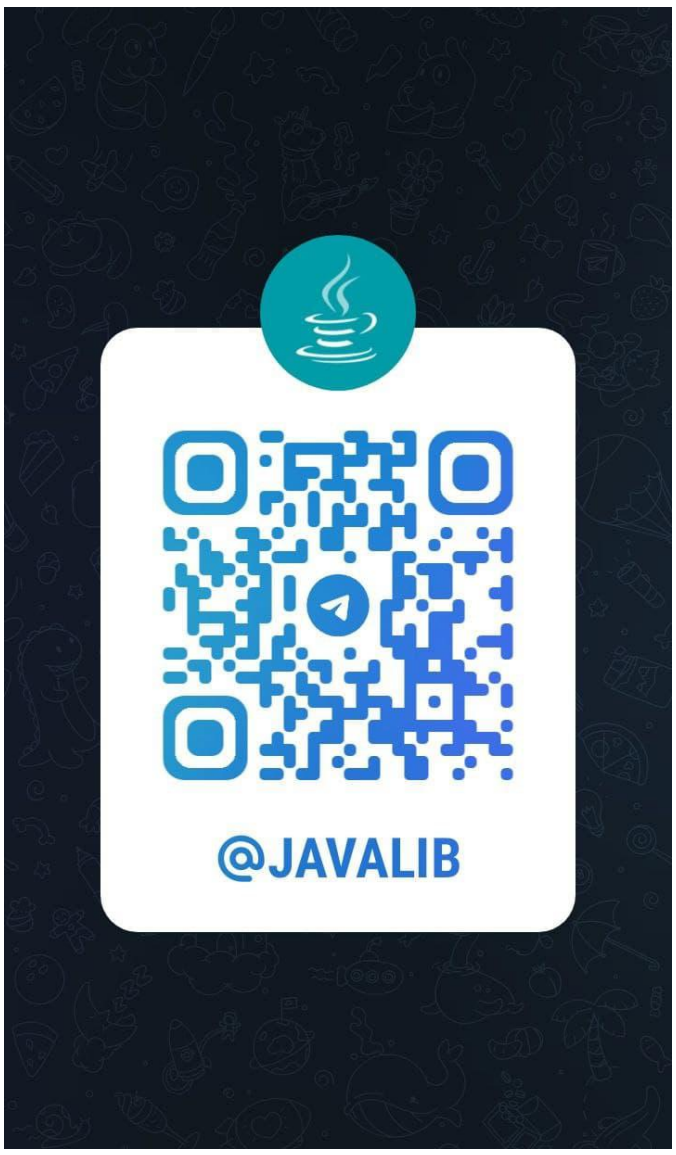

Exceptions in Java
(Basics, advanced concepts, and real API examples)
Nik Lumi

EXCEPTIONS IN JAVA

(Basics, advanced concepts, and real API examples)

Nik Lumi

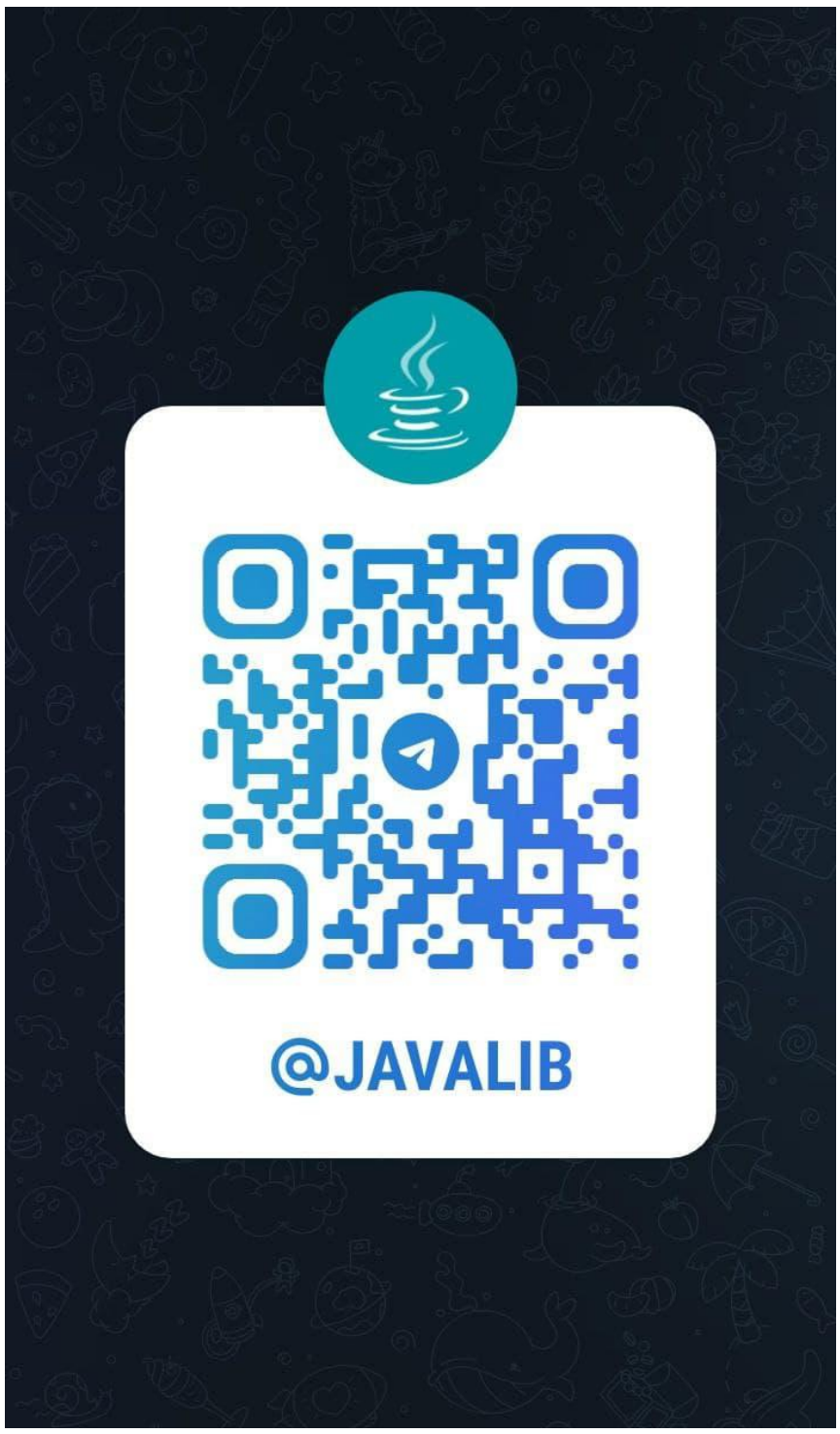


Published in the United States of America
First Published, January 2022

Contents

8. Modern Challenges

9. References



Preface

The concept of a software exception was introduced into programming in the early 60s. Since then, it has been adopted by many languages and has evolved significantly.

It is generally accepted that exceptions help separate different concerns in programming and therefore improve manageability of code. While very few people will deny the usefulness of exceptions, there is ongoing debate on *how* they should be used.

This book covers the basics of Java exceptions, advanced concept, and best practices, and gives a historical overview of how exception handling in Java was and is approached. In addition, this book provides examples of real APIs with analysis on how these APIs approach and handle exceptions.

This book assumes that the reader is familiar with the basics of the Java programming language, and is able to write, compile and execute a simple program. It also assumes that the reader is able to read and understand code excerpts.

We hope you both enjoy and learn from this book.

1. History

We all know well what an “exceptional situation” is in real life. It is when something unexpected happens that we must deal with. For most of us, it happens many times a day and often more frequently than expected.

You leave your garage and two blocks later your car alternator starts making a loud noise. What should you do? Drive home? Drive directly to your mechanic? Can your car make it? You must quickly answer these questions that run through your head and then you must make the corresponding decisions.

At a workplace — be it McDonalds or an airplane cockpit — people face these types of questions and often are given instructions on how to proceed.

In the IT world, these types of situations are extremely common.

The mechanism known as “exception handing” is supposed to solve these problems, or at least make them easier to solve. Exception handling is a systemic approach to exceptional situations.

Interrupts

The notion of interrupts, introduced in mid 1950s, is not exactly the same as modern exception handling but is close to it.

Interrupts refer to both hardware and software. In both cases there is a request to the processor to interrupt (or delay, or postpone – use any word you like) its current activity and handle this specific situation. After the specific situation is handled, the processor may resume its interrupted activity.

Lisp, Ada, COBOL, and others

Lisp was the first language that introduced software exception handling in a way similar to what we know today. It introduced key words like ERRSET, ERR, CATCH, and THROW. Their meanings and usages were similar to what they are today.

Computer languages like Ada, COBOL, Object Pascal, and others quickly adopted the concept. Its implementation may vary from language to language, but the core idea is the same.

C/C++

C did not have exceptions.

In 1985, C++ was released as an object-oriented extension of C. In addition to other things, exception handling was an integral part of the program. C++ added the power of object-oriented programming to it.

C++ is the language from which the architects of Java borrowed exception handling semantics. With C++, one may declare an exception in a method signature, but its usage will not be enforced by the compiler.

The architects of Java, as we will see later, tried to take things one step further. They added the idea of checked exceptions. This means that some exceptions may become a part of a method signature, thus requiring special handling.

Normal & Abnormal

The whole concept behind exception handling (and its conceptual predecessors) is that there are two ways that a program may be executed.

1) The program executes normally; all language elements work as they should; or
2) The program executes abnormally; there is not enough memory allocated, a null value is passed instead of a reference to a valid object, or there is no file by the location passed by the caller.

The separation of normal and abnormal execution sounds simple and intuitive, and normally does not cause controversy.

However, some experts go further and support the idea of a separation of abnormal behavior further into two categories - “recoverable abnormal” and “unrecoverable abnormal”. “Recoverable abnormal” conditions

can be intercepted by the program itself and tried for recovery. “Unrecoverable abnormal” however would normally require termination of the execution and human intervention. What is recoverable and what is not recoverable could have easily been given to the programmers discretion to be decided based upon each particular situation, however, the fathers of Java made it possible to enforce mandatory handling of “recoverable abnormal” as a part of the rigid programming contract. Many experts criticize this decision. Who is right and who is wrong will be discussed in a later chapter.

Return & Goto

Exceptions can be described as alternative return values, and exception handling as an alternative return mechanism. But, contrary to a return statement, they may return a call not to the next level up the call stack, but to an arbitrary level higher, bypassing all intermediate checks for the return value. So, another way to describe them is a non-local conditional goto.

Exceptions & OOP

Objects have very convenient semantics. In OOP languages (including Java) the information about exceptional behavior can be extracted, encapsulated, and separated from the normal flow. In addition to information about where, when, and how the exceptional conditions took place, information can be transparently added to the object that describes the exceptional condition.

In summary, in OOP languages exceptions combine the power of non-local transition of control with object-oriented semantics.

2. Essentials

In this chapter we explain the basic mechanism that the Java programming language offers for exception handling. This chapter intentionally does not cover what is considered right or wrong with regards to using exceptions; this will be covered in later chapters. The purpose of this chapter is to describe the technical aspects of exception handling provided in Java.

Types

Java is an object-oriented language, which means classes play an important role in it and its multiple concepts are represented in the form of classes. Exceptions are — no pun intended — not an exception to this rule.

Therefore, every exception is an object of a class with some specific behavior.

Hierarchy

There are several important classes in the exceptions hierarchy that are treated specially by the JVM.

Throwable

The **java.lang.Throwable** class is at the root of the Java exception hierarchy. Any exception used in the Java programming language should be its subclass.

The class has five constructors.

public Throwable()

public Throwable(String message)

public Throwable(String message, Throwable cause)

protected Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)

public Throwable(Throwable cause)

The first two create a simple exception with or without a corresponding message. Normally they are enough for most cases.

The latter three display more advanced concepts associated with exception handling: cause, suppressed exceptions and stack trace. We will return to these in the subchapter on exception analysis.

Checked & Unchecked

The three important subclasses (in the java.lang.* package) of the java.lang.Throwable in the exception hierarchy are:

public class Error extends Throwable

public class Exception extends Throwable

public class RuntimeException extends Exception

The three classes – **java.lang.Error, java.lang.Exception** and **java.lang.RuntimeException** – do not add anything to the programming logic as it is defined in the **Throwable**. They are “marker interfaces”, rather than fully functional classes.

Subclasses of **Error** and **RuntimeException** are called **unchecked** exceptions.

All other classes directly or indirectly inherited from **Throwable** are called **checked** exceptions.

As we will see later, the Java compiler treats checked and unchecked exceptions, in some respects, very differently.

Original Intent

The exception classes structure, as stated by Sun/Oracle, was intended to divide two types of possible

program failures, namely **faults** and **contingencies** [1].

Faults

A “Fault” is something that should never happen. Examples include misconfiguration of program environment, JVM failures, software bugs and similar.

Subclasses of java.lang.Error and java.lang.RuntimeException were reserved for this purpose.

Contingency

A “Contingency” is something that may be unpleasant but is expected and is even a part of a method return protocol. Examples include not enough funds, user not found, etc. These events should be expected, and the programming logic should be prepared to handle them.

Other subclasses of java.lang.Throwable (except subclasses of java.lang.Error and java.lang.RuntimeException, mentioned in the previous paragraph) were reserved for this purpose.

Raising an Exception

There are two stages in raising an exception in Java.

First, the exception should be created in the same way any other Java object is created. But simply creating an object of a Throwable class (or one of its subclasses) is not enough.

Second, the exception should be put in a specific state. Specifically, it should be thrown. Some other languages use the word raised. In Java the special statement **throw** serves this purpose.

Creating

If an exception is initiated by the user, as with any other Java object, it should first be explicitly created. It may be one of the existing core Java exception classes. The user may also decide to create their own class and use an exception of the corresponding class.

Therefore, the operator **new** should be called with one of the constructors available for the specific class:

```
Exception exception = new Exception(); // new MyException()
```

Throwing

Creating an exception is the first step of initiating an exceptional situation. For the second step, the exception should be thrown; i.e., a special statement should be used with it.

```
throw exception;
```

Throwing an exception can be combined with the previous creation step:

```
throw new Exception();
```

After an exception is thrown it begins its “flight” up the stack trace to the nearest handling block. If the JVM does not find an appropriate handler in the code, it will terminate the corresponding thread that has thrown the exception.

Users do not have to create an exception every time from scratch. They may use an already available exception. This process is called **re-throwing**.

Two points should be made regarding re-throwing.

First, if the reference used for it is null, then instead of your exception, an instance of java.lang.NullPointerException will be thrown. This happens because the JVM detects an attempt to throw a null exception object, which is not allowed. There are multiple situations when the JVM may decide to create and throw an exception of its own.

Second, a re-thrown exception may already have some information in it, which is irrelevant to the new context. That information should possibly be reset. This will be covered in detail in later chapters.

Detection by the JVM

As has already been mentioned, an exception may be created by the JVM. It may detect an exceptional situation and take care of it.

In this case the JVM will create an exception object (by calling an existing constructor) and throw it.

The JVM creates and throws this way only unchecked exceptions (Error, RuntimeException, and their subclasses).

According to the JVM specification [2] the following exceptions may be detected and thrown by the JVM.

16 **errors** from the java.lang package:

StackOverflowError,
AbstractMethodError,
AssertionError,
ClassCircularityError,
ClassFormatError,
ExceptionInInitializerError,
IllegalAccessError,
IncompatibleClassChangeError,
InstantiationError,
LinkageError,
NoClassDefFoundError,
NoSuchFieldError,
OutOfMemoryError,
UnsatisfiedLinkError,
VerifyError,
VirtualMachineError
10 **runtime exceptions** from the java.lang package:
ArithmeticException,
ArrayIndexOutOfBoundsException,
ArrayStoreException,
ClassCastException,
IllegalArgumentException,
IllegalMonitorStateException,
InstantiationException,
InterruptedException,
NegativeArraySizeException,
NullPointerException
1 **runtime exceptions** from the java.lang.invoke package:
WrongMethodTypeException
In total, there are 27 **unchecked** exceptions (16 **errors** and 11 **runtime exceptions**).

Interception & Processing

Interception and processing of exceptions are probably the most critical aspects of Java's exception handling mechanism. Understanding how this works is absolutely critical.

After an exception is created and/or (re-)thrown, sequential execution of the program is interrupted and the JVM starts looking in the local code and/or up the call stack for exception handling logic that may process the exception and continue the execution.

Basic Blocks

The **try, catch,** and **finally** blocks are the corner stones of the Java exception handling mechanism. The three serve as a guard, interceptor, processor, and finalizer for exceptions thrown inside of the corresponding construction. Since Java 7, two more major components have been added to the exception handling mechanism. These are the **Resource Specification** and the **AutoCloseable** interface.

try

The **try** block is also referred to as the "guarded region", "protected region", "guarded block" or other similar names. It serves as a guard for the exceptions thrown inside of it.

```
try {
    …
}
```

This is a regular **try** block.
The purpose of the **try** block is to execute and guard a code that may potentially throw an exception.

Resource Specification

There is another form of the **try** block. It is a **try** block with the **resource specification** statement.

```
try (
```

… // Resource specification

) {

…

}

The purpose of this **try** block is the same as the purpose of the regular **try** block, but in addition, it also provides the user with some facilities on how to initialize and automatically clean up resources used in the block.

The resource specification lets the user declare local (visible in the body of the **try** block) variables with initializers on the right. There are several limitations the compiler and the JVM impose on these variables.

First, a resource variable type should be a subclass of the **java.lang.AutoCloseable** interface.

Second, all of the resource variables should be initialized in the **resource specification**.

Third, the resource variables should be effectively final. Once initialized in the **resource specification** they cannot be reassigned in the **try** body (this stands true for resource variables initialized to null, they cannot be reassigned to a non-null object).

Example:

```
try (
    FileReader fileReader = new FileReader("myFile");
    BufferedReader reader = new BufferedReader(fileReader);
) {
    …
}
```

java.lang.AutoCloseable close()

As mentioned earlier, resources declared in the **Resource Specification** should implement the **java.lang.AutoCloseable** interface.

The interface has a method close().

void **close()** throws Exception;

The purpose of the method is to perform the corresponding resource clean up.

We will reference this activity as resource closure.

catch

The **catch** block looks as follows:

```
} catch (MyException caughtException) {
    …
}
```

The purpose of the block is to process a thrown and then intercepted exception.

The **catch** block is also called an “exception handler”. An exception parameter must be a valid Java exception.

The execution of the block starts immediately after the opening curly brace that follows the catch parameter declaration.

```
} catch (MyException caughtException) { // exception caught
    // <- the execution goes here
    …
}
```

The **catch** blocks may follow each other.

```
} catch (MyException caughtException) {
    …
} catch (MyAnotherException caughtException) {
```

…

}

The catch block may also use the “multi catch” construction.

Two catch blocks may often do a similar job, so it makes sense to reduce the redundancy in the code that handles the logic. In some situations, a more generic type can be used in the catch block and intercept both exceptions. But this approach is not always possible.

Since Java 7, two or more exceptions can be combined in one catch block.

Example 1 (pre-Java 7)

} catch (IOException e) {

…

} catch (SQLException e) {

…

}

Example 2 (post-Java 7)

} catch (IOException | SQLException e) {

…

}

The effective compile type of exception variable in the second example is the nearest common superclass of the exception classes in the multi-catch declaration.

The Java compiler enforces several rules for the **catch** block.

First, a **catch** block for a child exception class cannot go after a **catch** block for the parent class.

Example:

} catch (MyParentException caughtException) {

…

} catch (MyChildException caughtException) {

…

}

If you try to compile a code that has the above logic, you will get a compilation error that will look something like:

Unreachable catch block for MyChildException. It is already handled by the catch block for MyParentException

Second, in a “multi-catch” construction two exceptions cannot be in a parent-child relationship.

Example:

} catch (MyParentException | MyChildException caughtException) {

…

}

If you try to compile a code that has the above logic, you will get a compilation error that will look something like:

The exception MyChildException is already caught by the alternative MyParentException

finally

The last block is called **finally**. It appears as follows:

} finally {

…

}

The purpose of the **finally** block is to finalize the work done in the **try** block. Mainly, to release and/or clean up resources used in it.

try-catch-finally

The **try-catch-finally** construction starts with a regular **try** block.

```
try {
    …
}
```

The regular **try** block must be followed by at least one of the **catch** or **finally** block**.**

There may be many different combinations, including one **catch,** one **finally,** one **catch** and one **finally,** multiple **catch** blocks and one **finally** block, or multiple **catch** blocks and no **finally**. All of these combinations are valid.

The execution starts with the beginning of the **try** block, with the first line of code right after the opening curly brace that follows the **try** statement. If an exception is thrown inside of it, the rest of the code inside of the block is skipped and the program execution goes to the end of the block.

```
try { // <- the execution starts right after this line
    …
    throw thrownException;
    … // SKIPPED
} // <- the execution goes here
```

After that, the JVM starts looking for **catch** blocks that follow the **try**. The JVM checks if the type of the thrown exception is the type in the catch block declaration, which means the expression *thrownException instanceof MyException* is true.

If no corresponding **catch** block is found, the execution immediately goes to the **finally** (if any) block.

The first **catch** it finds will be used for the exception handling. In this case, the execution goes to the first line of the corresponding **catch** block. The exception in this case is called *caught*. The thrown exception is assigned to the reference within the **catch** block.

```
try {
    …
    throw new MySecondException();
    … // SKIPPED
} catch (MyFirstException exception) {
    … // SKIPPED
} catch (MySecondException exception) {
    … // <- the execution goes here
} catch (MyThirdException exception) {
    …
}
```

After the execution of the **catch** block is complete, it goes to the **finally** block.

The **finally** block is the last block that is executed.

If there is no **finally** block, the Java Language specification calls such construction **try-catch**, and if there is no **catch**, it uses the **try-finally** name. Both **try-catch** and **try-finally** are just particular cases of the **try-catch-finally** construction.

try-with-resources

This is a modified version of the **try-catch-finally** construction.

It starts with the **try** block with **resource specification**. For this construction, the **catch** and **finally** blocks

are optional.

As the first step, the resources in the **resource specification** are initialized.

```
try ( // <- the execution starts right after this line
    AutoCloseable ac1 = … ;
    AutoCloseable ac2 = … ;
    AutoCloseable ac3 = … ;
    AutoCloseable ac4 = … ;
    AutoCloseable ac5 = … ;
    …
) {
    …
}
```

ac1 … ac5 in the example above are called resources, they should implement **java.lang.AutoCloseable** interface and be final or effectively final and initialized inside the corresponding **resource specification**. They should be declared and initialized inside of the resource specification block.

The resource specification block execution then goes to the body of the **try** block, in the same way as it is done in a regular **try** block.

```
try (
    … // Resource specification
) { <- the execution continues right after this line
    …
}
```

If an exception is thrown inside of it, the rest of the code inside the block is skipped and the program execution goes to the end of the block.

```
try (
    …
) {
    …
    throw thrownException;
    … // SKIPPED
} // <- the execution goes here
```

This is the point where the behavior of the **try-with-resources** specification is different from the behavior of the **try-catch-finally** block.

At this point the JVM will start calling the **close()** methods of the resources successfully initialized in the **resource specification**. The order it will call the **close()** methods is the reverse order of how the resources were initialized.

It is important to note here that depending on the actual type of the declared resource variable the compiler will demand that the checked exception possibly thrown from the corresponding close() method to be intercepted in one of the consequent catch blocks, or intercepted in a surrounding try-catch, or passed up (a detailed definition of passing up will come later) by the enclosing method.

After the resources are closed, the behavior is the same as for a **try-catch-finally** specification. The execution goes to the first suitable (if any) **catch** and then (if any) to **finally**.

The Java Language specification calls **try-with-resources** to be a **basic try-with-resources** if it does not have **catch** and **finally**. If it has at least one **catch** or **finally**, it is called **extended try-with-resources.**

Transfer of Control

In the two previous paragraphs, the normal flow in the **try-catch-finally** and **try-with-resource** constructions was explained.

Challenges

The fundamental problem is that constructions intended to both guard and process exceptions may, in turn, throw their own exceptions or try to transfer control (with **break**, **continue,** and **return** statements) at any point during the processing. The execution flow may try to leave the **try**, **catch** & **finally** blocks before their end. With the introduction of **try-with-resource,** some additional implicit logic is added to the execution flow.

What happens in these cases?

Next, we cover the most common situations when an exception can be thrown, or a program execution control transmitted.

Resource Specification

There is no way transfer of control can happen in a **resource specification** via one of the following statements: **break**, **continue**, or **return**. These statements are not allowed in the **resource specification**. Therefore, the only possible way to transfer control is to throw an exception.

As previously mentioned, the JVM first initializes the resources declared in the **resource specification**.

If there is an exception during the initialization process, the rest of the process is interrupted, the body of the corresponding **try** block is skipped, and the JVM starts closing the resources by calling **AutoCloseable close()** in the opposite order of the resources initialization. The method is called only on the resources that were successfully open.

What happens during the process of closing resources is described in the paragraph below.

Example:

```
try (
    AutoCloseable ac1 = … ; // is good
    AutoCloseable ac2 = … ; // is null
    AutoCloseable ac3 = … ; // is good
    AutoCloseable ac4 = … ; // throws an exception
    AutoCloseable ac5 = … ; // will never happen
)
```

As already mentioned, some of the resources may be initialized to null. The JVM is fine with that. But it will not let you initialize them later; the variables in the resource specification are effectively final.

Therefore, in the code excerpt above the first resource is initialized to a valid object, the second is initialized to null, the third is initialized to a valid object, and while initializing the fourth resource, an exception is thrown.

The JVM skips the rest of the initialization process. It skips the body of the **try** block.

In the next step, the JVM will try to close the initialized resources, starting from the last to the first successfully opened.

Therefore, it will call **close()** on the ac3. It will not call anything on ac2, because it is null, and then it will call **close()** on the ac1.

try

As previously mentioned, the purpose of the **try** block is to guard thrown exceptions. The results of an exception being thrown from the block itself was covered in the sections on **try-catch-finally** and **try-with-resources.** This is their normal behavior.

Another challenge is the situation that occurs when control is transferred from the **try** block, using **break**, **continue,** and **return** statements.

There are two possible cases: a regular **try**, and a **try** with a **resource specification**.

First case: a regular **try** block.

If there is no **finally** block, the transfer of control is executed immediately.

If there is a **finally** block, the transfer of control is postponed, and control is transferred to the **finally** block for the final decision.

What happens in the **finally** block is described below.

Second case: a **try** with a **resource specification**.

The transfer of control is postponed, and the JVM starts closing the open resources in reverse order to how they were open, by executing the **close()** methods.

In the next paragraph we describe what happens during the closing of the resources.

AutoCloseable close()

The **close()** method of **AutoCloseable** interface is called only in case of a **try** with a **resource specification**. It is called on the resources that were successfully open in the **resource specification** section.

There are four cases to get to the process of closing resources.

1. The **resource specification** threw an exception.
2. The **try** block threw an exception.
3. The **try** block finished successfully.
4. The **try** block tried to transmit control out of itself.

First case: the **resource specification** threw an exception.

The JVM will skip the body of the **try** block and start closing successfully opened resources in the reverse order. If there is an exception thrown during the **close()** method call, that exception is considered **suppressed** and added to the list of suppressed exceptions associated with the original exception. When the process of closing is over, there may be a list of suppressed exceptions associated with the original exception. The beginning of the list will contain the first suppressed exception thrown; the end will contain the last.

Next, the JVM starts looking for a **catch** block that can handle the original exception. If such a block is found, the execution goes to it. If such a block is not found, the execution goes to the **finally** block.

What happens in the **catch** and **finally** blocks is described below.

Second case: an exception was thrown from a **try** block.

In this case the behavior is the same as in the first case above. The thrown exception is treated as the original one. All (if any) exceptions thrown during the method **close()** calls are added as **suppressed**.

Third case: the **try** block is finished successfully.

As with the first case, the JVM will start closing successfully opened resources in the reverse order.

Here we have two more subcases.

There was at least one exception thrown during the process of calling the **close()** methods.

No exception was thrown during the process of calling the **close()** methods.

First subcase: an exception is thrown during the process of closing, that exception is set aside as the original one. All other consequent exceptions (if thrown) are added as **suppressed** to the first one. After this, the JVM starts looking for a **catch** block that can handle the original exception. If such a block is found, the execution goes to it. If such a block is not found, the execution goes to the **finally** block.

Second subcase: no exception is thrown during the process of closing. The execution goes to the **finally** block.

Forth case: the **try** block tried to transfer control out of itself.

In this case the transfer of control is postponed. The JVM will start closing successfully opened resources in the reverse order.

Again, there are two subcases.

1. There was at least one exception thrown during the process of calling the **close()** methods.
2. No exception was thrown during the process of calling the close() methods.

First subcase: an exception is thrown during the process of closing; the exception is set aside. All other consequent exceptions (if thrown) are added as **suppressed** to the first one. The postponed transfer of control is discarded. After that, the JVM starts looking for a **catch** block that can handle the original exception. If such a block is found, the execution goes to it. If such a block is not found, the execution goes to the **finally** block.

Second subcase: no exception is thrown during the process of closing. The execution goes to the **finally** block. The postponed transfer of control waits for the decision to be made in the **finally** block.

catch

The only way to get into the **catch** block is by throwing an exception. One can either throw an exception from the resource specification, or from try block, or from one of AutoCloseable close() methods, during automatic resource closing.

Regardless of how we got into the **catch** block, its subsequent behavior is the same.

The **catch** block may decide to throw an exception or transfer control using a statement. In both scenarios the action is postponed, and control is transferred to the **finally** block for the final decision.

If the **catch** block ends naturally, there is no postponed action, and the control is transferred to the **finally** block.

finally

The **finally** block makes the final decision. If there is any postponed action from previous blocks, it can be overwritten by throwing an exception or by the explicit transfer of control using **break**, **continue,** and **return** statements.

If the **finally** block ends naturally, and there is a postponed action, that postponed action is executed.

If the **finally** block ends naturally, and there is no postponed action, the next line of code after the **finally** block is executed.

```
try {
        …
        throw new MyFirstExcepton();
        …
} catch (MyFirstExcepton exception) {
        …
        throw new MySecondExcepton();
        …
} finally {
        …
        throw new MyThirdExcepton();
        …
}
```

In the example above, an exception is thrown from the **try** block. But the transition of control is postponed. The execution goes to the **catch** block. The **catch** block overrides the postponed action. It throws its own exception. The throwing of the exception is postponed as well. The control is transited to the **finally** block. The **finally** block, in turn, overrides the action by throwing its own exception. Therefore, the last exception is thrown from the try-catch-finally block.

Special case(s)

There are several public methods that may interrupt JVM execution abruptly.

In the **java.lang.Runtime** class:

```
public void exit(int status)
public void halt(int status)
```

In the **java.lang.System** class:

```
public static void exit(int status)
```

This method simply calls the **java.lang.Runtime** corresponding exit() method. It is simply another publicly available alternative to accomplish the same action. Below, we discuss the first two.

Since the methods from the **java.lang.Runtime** class above are not static, one requires a Runtime instance to call them. Normally, it is done via calling the getRuntime() method.

```
public static Runtime getRuntime()
```

Example 1:

```
Runtime.getRuntime().exit(0);
```

Example 2:

```
Runtime.getRuntime().halt(0);
```

The difference between the two methods is that exit() provides a more graceful JVM shutdown by first calling registered shutdown hooks and finalizers.

Both exit() and halt() may throw **java.lang.SecurityException**, which is raised if the calling thread is not allowed to terminate the JVM. In addition, they may throw other runtime exceptions and errors. For example, **java.lang.OutOfMemoryError**.

Internally, both methods use package-private class **java.lang.Shutdown**, that has corresponding exit() and

halt() methods as well. The sequence of calls finally boils down to:

static native void halt0(int status);

This native method terminates the JVM. Nothing is executed after that, including any finalization defined in **try-catch-finally** or **try-with-resources** construction, no finally block is called, and no close() method is called on AutoCloseable resources.

Example3:

```
…
try {
    System.out.println("It happens #1");
    Runtime.getRuntime().addShutdownHook(
        new Thread(() -> {
            System.out.println("It happens #2");
        }));
    Runtime.getRuntime().exit(0);
    throw new Exception();
} catch (Exception e) {
    System.out.println("It never happens #1");
} finally {
    System.out.println("It never happens #2");
}
System.out.println("It never happens #3");
…
```

The outcome is:

```
It happens #1
It happens #2
```

As one can see, nothing is executed after the exit() method, including the finally block. The same is true about any other code, except for registered shutdown hooks and finalizers.

If in the code above, we replace:

```
Runtime.getRuntime().exit(0);
```

With:

```
Runtime.getRuntime().halt(0);
```

The outcome will be:

```
It happens #1
```

As one can see, nothing is executed after the halt() method, including the finally block. The same is true of any other code, including registered shutdown hooks and finalizers.

An important point to mention is that, if on the way to the JVM shutdown after the call of one of the two methods there is an exception thrown, for example in the above-mentioned scenario when the thread is not allowed to terminate the JVM, then the program behaves normally as with any other exception thrown.

Summary

In summary, using the above, we are able to come up with the following set of rules of the transfer of control in **try-catch-finally** blocks.

1) In a regular **try-catch-finally** block if there is an exception thrown in **try**, go to (4). If there is any other form of transfer of control or **try** ends successfully, go to (6).
2) In a **try-with-resource** if there is any form of transfer of control in **try** or **resource specification** or **try**

ends successfully, go to (3).

3) The JVM starts closing successfully initialized (to a non-null value) resources in the reverse order by calling the corresponding **AutoCloseable close()** methods.
   If there is an exception thrown during the process of closing the resources, those exceptions are set aside, until the JVM ends its attempt to close all eligible resources. After the process is finished and there was an exception thrown before (from **try** or **resource specification**) that exception is selected as the main one, the exceptions that were set aside (if any) are consequently attached to it as suppressed, and the main exception is thrown. If there was no exception thrown (from **try** or **resource specification**), but there were exceptions set aside, the first one is selected as the main one, all consequent ones are attached to it as suppressed, and the main one is thrown. Go to 4).
   If there was no exception thrown from **try**, **resource specification** and the closing procedure, go to (6).
4) The JVM starts looking for the first **catch** block, corresponding to the thrown exception. If such a block is found, go to (5), if not, go to (6).
5) The **catch** block is executed, after the process is over go to (6). If there was any form of transfer of control, it is postponed.
6) The JVM checks if there is the **finally** block. If there is one, go to (7). Otherwise, go to (8).
7) The **finally** block is executed.
   If there is any form of transfer of control, any previously postponed transfer of control is discarded and the current transfer of control is executed immediately.
   If the execution reaches the last line, go to (8).
8) If there is a postponed transfer of control, it is executed immediately.
   If there is no postponed transfer of control, the next operation after the **try-catch-finally** block is executed.
9) If any of the methods that that terminated the JVM is called only eligible shutdown hook and finalizers are executed, but no **try-catch-finally** or **try-with-resources** construction elements.

Passing Up

As we said above, the Java language in some aspects treats **checked** and **unchecked** exceptions very differently.

In the case of a checked exception, the compiler forces you to handle it. Surround it with a **try** that has the corresponding **catch** block, that handles this checked exception type. Alternately, one can pass up the responsibility to do that to the caller of the method.

Example 1:

```
public class NotEnougFunds extends Exception {

    …

}

public void transferFunds(

    Account accountFrom,

    Account accountTo,

    BigDecimal amount) {

    …

        throw new NotEnougFunds();

    …

}
```

If one tries to run and compile the code similar to the above, they will get a compiler message:

Unhandled exception type NotEnougFunds

You can surround the throw with a **try-catch** block.
Example 2:

```
try {

    …
```

```
    throw new NotEnoughFunds();
    …
} catch (NotEnoughFunds exception) {
    … // do something
}
```

But the method itself is often unable to make the decision on what to do in situations like this, so as an alternative the responsibility to deal with the exception may be transferred to the caller of the method.

throws

A method that passes the responsibility of what to do with an exception to the caller, does so by using the **throws** keyword.

Example:

```
public void transferFunds(
    Account accountFrom,
    Account accountTo,
    BigDecimal amount) throws NotEnougFunds {
    …
}
```

With this change the example given in the previous paragraph will compile.

Method Signature

Checked exceptions are a part of the method signature, unchecked exceptions are not.

Example 1:

```
public interface myInterface {
    void method();
}
```

When you try to compile the following code:

```
public class myClass implement myInterface {
    void method() throws RuntimeException {
    };
}
```

It will compile without a problem.

If you change the **RuntimeException** to **Exception** and try to compile it again:

```
public class myClass implement myInterface {
    public void method() throws Exception {
    };
}
```

You will get a compilation problem:

```
Exception myException is not compatible with throws clause in myInterface.method()
```

This means the compiler understands that when you add a checked exception to a method, you are significantly changing its behavior. And in some cases, the changed behavior is not compatible with the behavior or the parent interface. Which in turn violates the fundamental OOO principal – inheritance.

The general rule here is that you cannot add checked exceptions to a method in child classes or interfaces.

With an unchecked exception, there are no restrictions on what you can do.

Execution Mechanism

Thread, stack trace and associated notions are key concepts in a Java program execution mechanism. Understanding how they work is critical for failure analysis.

Thread

A thread is a unit of execution in Java. There are two ways to start a thread in Java.

First, one may start a thread (also called the **main thread**) by starting a Java program. In this case the main thread will call the method main() of the corresponding class:

```
public static void main(String[] args) {
    …
}
```

The method main() signature may include some checked exceptions.

```
public static void main(String[] args) throws MyException {
    …
}
```

Second, a thread may be started by calling the method start() on an instance of the class Thread (either by inheriting from it directly, or via the creation of an instance of java.lang.Runnable interface). In this case, the first method of the thread will be the method run() of the corresponding class:

```
public void run() {
    …
}
```

The threads of the second type can only be started by another thread and its method run() can only throw unchecked exceptions.

Uncaught Exceptions

If an uncaught exception is thrown from the method main() or run() that started the thread execution, it may still be handled. The thread at this point is already doomed, but there is an opportunity to provide some finalization logic.

Thread.setUncaughtExceptionHandler()

The first handler the JVM is looking for is the handler registered directly for the thread being executed.

public static void setUncaughtExceptionHandler (Thread.UncaughtExceptionHandler eh)

The class Thread.UncaughtExceptionHandler contains one method:

void uncaughtException(Thread t, Throwable e)

Therefore, you can define a handler for your thread and register it with the thread. When there is an unhandled exception thrown from the main() or run(), the JVM will call the uncaughtException() method of your handler.

ThreadGroup.uncaughtException()

If the handler registered with your thread is not found, the JVM starts looking for another option.

It takes the thread group and looks for the following method.

public void uncaughtException(Thread t, Throwable e)

If the method is not defined, the JVM takes the parent group and reapplies the searching logic. This way the JVM climbs the hierarchy of thread groups, trying to find the first one that defines the method.

Thread.setDefaultUncaughtExceptionHandler()

If no handler is found in the thread itself and its parent groups, the JVM goes to the last option: the handler registered for the Thread class in general in the following static method.

public static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)

If the handler is registered, it is called.

No handler

If no handler is found, then there are two final options.

If the exception is of the java.lang.ThreadDeath class, the thread dies silently. For all other exceptions, the JVM prints its tack trace before it dies.

Containers

Most of the time users do not start threads manually. Instead, they provide classes with specific methods that do the actual work. Those classes are loaded into a Java engine called container, which in turn (sometime after a complicated startup and initialization process) calls the corresponding methods.

In addition to startup, load, and initialization of require classes, containers provide multiple other facilities – logging configuration, disabling/enabling, monitoring and other.

One of the most important facilities provided by a container is the handling of uncaught exceptions thrown from the service methods provided by the user.

finalize()

If an exception is thrown from the java.lang.Object finalize() method, and is uncaught, it will be ignored and the rest of the finalize() method skipped.

No thread handler (even if defined) will be called.

This is true only if the finalize() method was invoked by the JVM FinalizerThread thread, as a part of garbage collection. If the method is called as a regular Java method, the same rules apply as in the case of a regular invocation.

Analysis

When an exception is thrown, there is important information associated with it. Some of the information is created and added to an exception by native code in the base class constructors. Some of the information may be passed by a user to the exception constructor. Some other important information may be added to an exception later.

This information may be used for further analysis.

JVM Stack

As a thread starts, it will start calling constructors, static methods, and methods on other objects. Those methods will, in turn, call other constructors and methods. Thus, there is an entire sequence of calls from the method that started the thread to the one that is currently being executed.

To support this chain of calls, the JVM on the background creates a complicated structure called **stack**. For each constructor/method call the JVM creates a structure that defines the call context, referred to as a frame. As a method is called, its frame is pushed onto the stack. When the method returns, its frame is popped off the stack. The JVM Specification gives a detailed account on how the stack, frame and other elements are created, what they consist of, and how they behave.

For our simplified purposes, we will view the stack as a sequence of calls from the beginning of the thread to the current call.

Stack Trace

For a given thread, at a certain point, its stack trace is a list of method calls from the beginning of the thread to the current execution.

Java provides the opportunity to access the entire stack trace at any point of a program. The stack trace is represented as a list of **StackTraceElement** Java objects. A **StackTraceElement** corresponds to a method call, and (in JVM Specification terminology) roughly to a stack frame.

Some of the important methods are:

*public String **getFileName()***

The getFileName() returns the full name of the corresponding source code unit (in most systems, the source file). For the inner class it will be the full name of its outer unit, plus name of the inner class separated by a dollar sign ($). It is not unusual that the java compiler excludes the source unit name during the compilation process. In this case it is not available, and the method does not return it.

public int getLineNumber()

The getLineNumber() returns the line number for the call in the source code unit. It is not unusual that the java compiler excludes the line number during the compilation process. In this case it is not available, and the method does not return it.

public String getMethodName()

As one can see, this is basic information about the context of the execution. The method **toString()** is called when an element is printed, it combines the information from the three methods above.

The **javac** utility [3] compilation flag **-g** specifies which debug information is included in the produced .class files. If it is set to **g:none**, nothing is included. The user may decide to set it to **source**, **lines** and **vars**. In this case, only the information on source file name, line number and local variables will be included, accordingly. By default, only line number and source file information is included.

To access the current thread stack trace (an array of StackTraceElement object), one can make this call:

Thread.currentThread().getStackTrace();

Stack-Walking API

Starting with Java 9, additional mechanisms were added to the Java Core API that allowed users to traverse the stack trace.

It is more flexible and in general provides more information to the user than just simple access to the stack trace. As of now, it is not integrated with the exception handling mechanism directly, but this could be possible in the future.

Cause

Sometimes one exceptional situation may lead to another. For example, an inability to connect to a bank may cause an inability to withdraw funds.

Similarly, one exception thrown may logically cause another exception to be thrown. In this case, the first exception is called the cause of the second one.

In an upcoming chapter we will discuss when this pattern is appropriate, but for now we will limit our discussion to the facilities offered by the core Java API.

Example:

```
public void purchaseABook(
        String ISBN,
        Account account,
        BigDecimal price) throws PurchaseException {
        …
        try {
                …
        }catch (StorageAccessException e) {
                throw new PurchaseException(e);
        }
        …
}
```

The code that calls purchaseABook() method may simply not know what to do with StorageAcessException. Therefore, the original exception should be reinterpreted. Like for a person who only speaks Chinese, every English message should be translated.

Recall that the **java.lang.Throwable** class has three constructors each that accepts another exception as a cause.

public Throwable(String message, Throwable cause)

protected Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)

public Throwable(Throwable cause)

Knowing the cause of the exception is important in exception analysis.

Suppressed

Another important concept in exception analysis is a suppressed exception.

This concept was already introduced during discussions of **try-with-resources** construction. It was described as an exception that was raised after the main exception was thrown, during the closing of the open resources. In order not to discard the exception, it was attached to the main one.

But this concept is wider and goes beyond the closing of resources. The case of the **try-with-resources** construction is just one case when it can be used. In the construction it is done automatically, however, based on user discretion, it may also be used selectively in other cases.

There are several scenarios where a suppressed exception exists.

One example can be taken from the previous paragraph: There is a requirement to inform the support team anytime storage is unavailable. Therefore, programmers come up with a code similar to the following.

Example 1:

```
public void purchaseABook(
        String ISBN,
        Account account,
        BigDecimal price) throws PurchaseException {
        …
        try {
                …
        }catch (StorageAccessException e1) {
                PurchaseException newException = new PurchaseException(e1)
                try {
                        …
                        sendEmailToSupport();
                        …
                } catch (EmailException e2) {
                        newException.addSuppressed(e2);
                }
                throw newException;
        }
        …
}
```

In this scenario, the code fails to communicate with the storage. The corresponding exception is thrown. This exception is interpreted in terms of the calling code.

But the problem does not end here. The code tries to notify the support team by sending them an email. And this attempt fails. There is another exception. It did not cause the original problem, but it cannot be ignored. Only one exception can be sent to the caller, therefore the second exception is attached as a suppressed one to the exception that is sent to the caller.

Therefore, in this case: (1) an exception of the PurchaseException class is send to the caller, (2) the original StorageAccessException is its cause, and (3) the EmailException is attached as a suppressed one.

Therefore, in this first scenario there is a suppressed exception when some finalizing, post-failure activity fails.

Here is another example.

You write a program for the local 911 office. When there is information that a house is on fire, you are supposed to automatically inform both the police and fire departments.

Example 2:

```
public void houseOnFire(
    Address address,
    VoiceMessage callRecord) throws ProcessingException {
    …
    ProcessingException e = null;
    try {
        …
        notifyPolice(address, callRecord);
        …
}catch (NotificaionException e1) {
        e = new ProcessingException(e1);
}
try {
        …
        notifyFireDepartment(address, callRecord);
        …
}catch (NotificaionException e2) {
        if (e == null) {
                e = new ProcessingException(e2);
        } else {
                e.setSuppressed(e2);
        }
        …
}
```

In this situation, when both notifications fail, we have two exceptions, but can pass up only one. The second one is attached to the first one as a suppressed.

Therefore, in the second scenario, we have a suppressed exception when two (or more) parallel activities on the same logical level fail.

There are also some hybrid cases where the first parallel activity fails, the second succeeds, but the second activity clean up fails.

In the chapter on patterns and anti-patterns, suppressed exceptions will be examined further.

Exception Info

When an exception is created, it has some useful information associated with it. The **java.lang.Throwable** has constructors that are implicitly called and set this information.

getMessage()

This message is an optional, context dependent part of the exception. In an exception it is just a placeholder, the user is responsible for providing information for it. It is a good place to add some valuable information known from the context where the exceptional situation took place.

It is passed as a parameter during the exception creation. Three of the five constructors that are available for the **java.lang.Throwable** class have it.

public Throwable()

*public Throwable(String **message**)*

*public Throwable(String **message**, Throwable cause)*

*protected Throwable(String **message**, Throwable cause, boolean enableSuppression, boolean writableStackTrace)*

public Throwable(Throwable cause)

It can be assessed by the method of the **java.lang.Throwable**:

public String getMessage()

There also is a method that by design is supposed to be redefined in subclasses:

public String getLocalizedMessage()

It is supposed to return a localized version of the message. Its default implementation returns the same result as getMessage().

getCause()

This is the original exception, and the cause of the current one. It can be passed as a constructor's parameter. There are three constructors in the **java.lang.Throwable** class that accept it.

public Throwable()

public Throwable(String message)

*public Throwable(String message, Throwable **cause**)*

*protected Throwable(String message, Throwable **cause**, boolean enableSuppression, boolean writableStackTrace)*

*public Throwable(Throwable **cause**)*

An important method in the class is:

public Throwable initCause(Throwable cause)

It can be called only once; constructors internally call it, but it may be called from a user's code as well.

To access the cause there is a method:

public Throwable getCause()

There can be only one cause for a particular exception.

getSuppressed()

In a way similar to the cause, suppressed exceptions can be added to an exception. There is no constructor that can do this, but there is a method that can:

public final void addSuppressed(Throwable exception)

There can be more than one suppressed exception for a particular exception.

To access the array of suppressed exceptions associated with the current one, the following method is used:

public final Throwable[] getSuppressed()

Here is a way to prevent addition of suppressed exception:

protected Throwable(String message,

Throwable cause,

boolean enableSuppression,

boolean writableStackTrace)

The flag **enableSuppression** when set to false prevents the addition of any suppressed exception, addSuppressed(java.lang.Throwable) does not have any affect, and getSuppressed() returns an empty array.

Stack Trace

A StackTraceElement[] array associated with an exception indicates the stack trace state at the moment that the exception was created. Because every exception is a subclass of the **java.lang.Throwable**, it has to eventually call one of its constructors. Four out of five constructors in **java.lang.Throwable** do this automatically, when an exception is created. It is done by invoking the method:

public Throwable fillInStackTrace()

The fillInStackTrace() method may also be invoked at any time and it will reset the stack trace with its current state.

The stack trace is returned by the following method:

public SteckTraceElement[] getStackTrace()

The fifth constructor demonstrates more complicated behavior:

*protected Throw*able(String *message,*

Throwable cause,

boolean enableSuppression,

boolean writableStackTrace)

The flag **writableStackTrace,** when set to false, prevents the creation of any stack trace. It may make sense to save some system resources for the JVM. In this case, getStackTrace() returns an empty array.

The stack trace be may directly manipulated by the method:

*protected void setStackTrace(StackTrace*Element*[] stackTrace)*

However, it is strongly advised that you do not attempt this unless you are very knowledgeable on this topic. The usage of this method is reserved for advanced systems developers, for example when an exception received from a remote system is deserialized and must be restored in the local system with the stack trace elements from the local call added.

printStackTrace()

This method is a user-friendly version of printing the array of stack trace elements associated with the exception.

Professional Java programmers constantly deal with the results of this method (or its equivalents) and know very well what it looks like.

This method doesn’t just print elements of the stack trace associated with the exception. It actually prints three sections.

First, it prints the current stack trace elements. Second, it prints the stack traces of the suppressed exceptions, associated with the current exception. And third, it prints the stack trace of the cause. For the suppressed exceptions and the cause, the method recursively calls itself. Therefore, suppressed and the cause exceptions may have their own suppressed and cause exceptions in the print.

Handling

We have already covered some technical aspects associated with exception handling in Java. One of these was the mechanism of thread handlers that can handle an uncaught exception. But this is an extreme case and developers normally want to handle exceptions in a much more graceful way.

The catch that intercepts and finally handles the exception is normally located closer to the top of the stack call, near the method that is on the highest level that triggered the activity that may fail.

If it is a UI-based interactive application based on the MVC architecture, it normally happens in the controller. But this depends on the nature of the application. For example, if the UI application is something like a scheduler, where the user initiates and/or executes tasks, the actual handling of the exceptions thrown may happen down the call stack and will not necessarily involve explicit message notification shown to the initiator.

If it is a non-UI activity, exception handling may include simple logging of the exception and termination of the thread that executed the corresponding task. The tool that administers the execution is supposed to monitor the logs and notify the administrator who, in turn, is supposed to take some type of action and/or inform the developers.

In summary, what should end up being done with an exception thrown is very application/task and context dependent.

3. Advanced Aspects

In this chapter we cover advanced technical aspects associated with exception handling that go beyond basics. They are technical in nature and answer the question “what’s under the hood?”. Understanding these aspects is crucial for a professional Java programmer.

Overhead Costs

Handling exceptions is associated with some overheads.

There are optimization techniques used by modern JVM implementations to speed up exception handling, but they are still not perfect and cannot guarantee the absence of overheads.

Overhead cost is less visible if the code that uses it performs a single operation. An exception thrown here and there will most likely take more execution time than an operation performed without throwing the exception, but it is unlikely that it will take significant system resources.

But when a code does the same thing repeatedly, cycling or as part of a massive operation, the cost could be very high.

Creating

The creation of an exception is an expensive operation. Every Java exception inherits from the Throwable class, so when an exception is created one of its constructors is inevitably called.

Four out of five (in Java 7+) constructors in the Throwable in turn call the fillInStackTrace() method that fills in the stack trace. The fillInStackTrace() is the most expensive call made while creating an exception. The only constructor that does not call it is this:

protected Throwable(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)

If the writableStackTrace parameter is true, the stack trace will never be filled.

This constructor is rarely used, and most Java programmers never write code that utilizes it. But it may be considered as a lightweight alternative to other constructors. The obvious disadvantage of this approach is that the information about the stack trace will be missing.

Another approach is caching and/or reusing exception, but in this case the stack trace (filled in by fillInStackTrace() method) will not correspond to the new call, which is not significantly better than just being missing.

One more alternative is to simply override (it is not declared final) the fillInStackTrace() with something like the following:

```
public synchronized Throwable fillInStackTrace() {
    return this;
}
```

But this approach will work only for your custom classes, while again suffering from the issue of a missing stack trace.

In summary, the filling in of the stack trace is the most expensive part of an exception creation. The cost of it is proportional to the depth of the stack [4].

Stack Unwinding

Another costly part of handling exceptions is what is called unwinding the stack trace.

When an exception is thrown, if there is no local (in the current method) catch that can intercept and handle it, the JVM starts looking for a handler up the call stack. Meanwhile, it has to pop the frames off it. This process is called unwinding.

Unwinding the call stack in addition to filling in the stack trace, is another costly part of handling exceptions. The cost of it is roughly proportional to the number of frames the JMV has to pop off the call stack

in order to find the appropriate handler [4].

The JVM View

There are several important elements associated with exception handling in the structure of a *.class file.

Exceptions

Exceptions attribute is an element in the *attributes* list in a *method_info* structure. It has the following format [2]:

```
Exceptions_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions];
}
```

This is what the Java compiler looks at when generating Java code for classes that reference the class and its corresponding method.

The JVM specification mandates that a method should throw only runtime exceptions (those that inherit from java.lang.Error or java.lang.RuntimeException) or exceptions listed in the corresponding exception_index_table array.

Let us take a look at a particular example.

```
public class A {
    public void method() throws java.io.IOException {
    }
}
```

The code compiles.

```
javac A.java
```

Next let us try to compile the following code that utilizes the A class.

```
public class Test {
    public static void main(String[] args) {
        new A().method();
    }
}
```

It fails.

```
javac Test.java
Test.java:4: error: unreported exception IOException; must be caught or declared to be thrown
        new A().method();
                ^
1 error
```

The reason for failure is the fact that the compiler realized that the A.method() may potentially throw a java.io.IOException and this exception should be handled by being caught or thrown. The main method does not do it.

Let us decompile the A.class with the javap utility.

```
javap -l -c -v A.class
```

The outcome is:

```
Classfile A.class
  Last modified Sep 10, 2021; size 272 bytes
  MD5 checksum 43d3146c87021232b2d18b943278971c
  Compiled from "A.java"
public class A
  minor version: 0
  major version: 59
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #7 // A
  super_class: #2 // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref   #2.#3 // java/lang/Object."<init>":()V
   #2 = Class       #4 // java/lang/Object
   #3 = NameAndType #5:#6 // "<init>":()V
   #4 = Utf8        java/lang/Object
   #5 = Utf8        <init>
   #6 = Utf8        ()V
   #7 = Class       #8 // A
   #8 = Utf8        A
   #9 = Utf8        Code
  #10 = Utf8        LineNumberTable
  #11 = Utf8        method
  #12 = Utf8        Exceptions
  #13 = Class       #14 // java/io/IOException
  #14 = Utf8        java/io/IOException
  #15 = Utf8        SourceFile
  #16 = Utf8        A.java
{
  public A();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1 // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
```

```
            line 2: 0
    public static void method() throws java.io.IOException;
        descriptor: ()V
        flags: (0x0009) ACC_PUBLIC, ACC_STATIC
        Code:
            stack=0, locals=0, args_size=0
                0: return
            LineNumberTable:
                line 6: 0
        Exceptions:
            throws java.io.IOException
}
SourceFile: "A.java"
```

According to the JVM specification, *u2 attribute_name_index* should point to a constant pool entry that represents “Exceptions” string. In our case it is #12. In hexadecimal representation it should be 0C. Since the item is of u2 type, which corresponds to 16-bit, the sequence we are looking for is “00 0C”. The next item *u4 attribute_length* should correspond to 32-bits. In order to calculate its value, we must sum up the *u2 number_of_exceptions* item length, which is fixed 2 bytes, plus the number of bytes for the *u2 exception_index_table [number_of_exceptions]*, which in our case is 2 (because we have only one exception). Altogether, the field should have 00000004 value. As already said, the *u2 number_of_exceptions* should have 0001, and the value of *u2 exception_index_table [number_of_exceptions]* should point to #14, which is 000D.

Altogether, we should look for the following sequence:

00 0C 00 00 00 04 00 01 00 0D

If we want the compiler to not complain about the exceptions thrown from the method, this sequence should be removed. In addition, we should adjust the enclosing method definition. Now it has one attribute, not two. For the sake of brevity, we omit the corresponding calculation and come up with this:

00 01 00 0B 00 06 00 ***02***

The above should be replaced with:

00 01 00 0B 00 06 00 ***01***

Then we open the A.class file with a hex editor and “doctor” the file accordingly, saving the result.

The code compiles.

javac Test.java

We successfully removed the Exceptions attribute of the A.method() and, in doing so, “blinded” the compiler, which now “believes” that the method does not throw a checked exception.

In summary, the Exceptions attribute provides information about checked exceptions a method may throw. It is taken into consideration and enforced during the compilation, but not in runtime.

Exception Table

The Exception table is the next important exception handling topic to cover. This is information contained in a *.class file, or more formally *exception_table* array of the *code* attribute in *method_info* structure [2]. Every instance in it describes one exception handler.

```
Code_attribute {

    u2 attribute_name_index;

    u4 attribute_length;

    u2 max_stack;
```

```
        u2 max_locals;
        u4 code_length;
        u1 code[code_length];
        u2 exception_table_length;
        {   u2 start_pc;
            u2 end_pc;
            u2 handler_pc;
            u2 catch_type;
        } exception_table[exception_table_length];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

Let us compile the following example:

```
import java.net.MalformedURLException;
import java.net.URISyntaxException;
public class A {
    public static void check(String url) {
        try {
            new java.net.URL(url).toURI();
        } catch (MalformedURLException e) {
            System.out.println("MalformedURLException");
        } catch (URISyntaxException e) {
            System.out.println("URISyntaxException ");
        }
    }
}
```

It compiles just fine.

```
javac A.java
```

Now let us decompile the resulting A.class file and see what was generated by the compiler in terms of bytecode.

```
javap -l -c -v A.class
```

The result is the following listing.

```
Classfile A.class
  Last modified Sep 10, 2021; size 628 bytes
  MD5 checksum 2e6bfd0baa456d45c51c57689c84b589
  Compiled from "A.java"
public class A
  minor version: 0
  major version: 59
```

```
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #33                         // A
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Class              #8 // java/net/URL
   #8 = Utf8               java/net/URL
   #9 = Methodref          #7.#10         // java/net/URL."<init>":(Ljava/lang/String;)V
  #10 = NameAndType        #5:#11 // "<init>":(Ljava/lang/String;)V
  #11 = Utf8               (Ljava/lang/String;)V
  #12 = Methodref          #7.#13 // java/net/URL.toURI:()Ljava/net/URI;
  #13 = NameAndType        #14:#15 // toURI:()Ljava/net/URI;
  #14 = Utf8               toURI
  #15 = Utf8               ()Ljava/net/URI;
  #16 = Class              #17 // java/net/MalformedURLException
  #17 = Utf8               java/net/MalformedURLException
  #18 = Fieldref           #19.#20        // java/lang/System.out:Ljava/io/PrintStream;
  #19 = Class              #21            // java/lang/System
  #20 = NameAndType        #22:#23        // out:Ljava/io/PrintStream;
  #21 = Utf8               java/lang/System
  #22 = Utf8               out
  #23 = Utf8               Ljava/io/PrintStream;
  #24 = String             #25            // MalformedURLException
  #25 = Utf8               MalformedURLException
  #26 = Methodref          #27.#28        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #27 = Class              #29            // java/io/PrintStream
  #28 = NameAndType        #30:#11        // println:(Ljava/lang/String;)V
  #29 = Utf8               java/io/PrintStream
  #30 = Utf8               println
  #31 = Class              #32 // java/net/URISyntaxException
  #32 = Utf8               java/net/URISyntaxException
  #33 = Class              #34 // A
  #34 = Utf8               A
```

```
   #35 = Utf8               Code
   #36 = Utf8               LineNumberTable
   #37 = Utf8               check
   #38 = Utf8               StackMapTable
   #39 = Utf8               SourceFile
   #40 = Utf8               A.java
{
   public A();
      descriptor: ()V
      flags: (0x0001) ACC_PUBLIC
      Code:
         stack=1, locals=1, args_size=1
            0: aload_0
            1: invokespecial #1 // Method java/lang/Object."<init>":()V
            4: return
        LineNumberTable:
          line 4: 0
public static void check(java.lang.String);
   descriptor: (Ljava/lang/String;)V
   flags: (0x0009) ACC_PUBLIC, ACC_STATIC
   Code:
      stack=3, locals=2, args_size=1
            0: new           #7 // class java/net/URL
            3: dup
            4: aload_0
            5: invokespecial #9 // Method java/net/URL."<init>":(Ljava/lang/String;)V
            8: invokevirtual #12                 // Method java/net/URL.toURI:()Ljava/net/URI;
           11: pop
           12: goto          36
           15: astore_1
           16: getstatic     #18                 // Field java/lang/System.out:Ljava/io/PrintStream;
           19: ldc           #24                 // String MalformedURLException
           21: invokevirtual #26                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
           24: goto          36
           27: astore_1
           28: getstatic     #18                 // Field java/lang/System.out:Ljava/io/PrintStream;
           31: ldc           #24                 // String MalformedURLException
           33: invokevirtual #26                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
           36: return
```

Exception table:

from to target type

0 12 15 Class java/net/MalformedURLException

0 12 27 Class java/net/URISyntaxException

LineNumberTable:

line 8: 0

line 13: 12

line 9: 15

line 10: 16

line 13: 24

line 11: 27

line 12: 28

line 14: 36

StackMapTable: number_of_entries = 3

frame_type = 79 / same_locals_1_stack_item */*

stack = [class java/net/MalformedURLException]

frame_type = 75 / same_locals_1_stack_item */*

stack = [class java/net/URISyntaxException]

frame_type = 8 / same */*

}

SourceFile: "A.java"

In the listing above, the “Exception table” is what we are looking for.

Exception table:

from to target type

0 12 15 Class java/net/MalformedURLException

0 12 27 Class java/net/URISyntaxException

In the byte code, this excerpt of listing looks somewhat illegible.

00 02 00 00 00 0C 00 0F 00 10 00 00 00 0C 00 1B 00 1F

The purpose of the exception table is to contain the mark up and mapping of try-catch-finally blocks.

If a code contains try-with-resources and/or finally, the compiler may insert additional records into the “Exception table” which will serve the purpose of implementing the logic of those blocks.

For example, the following code performs basic url check.

```
import java.net.MalformedURLException;

import java.net.URISyntaxException;

public class A {

    public static void check(String url) {

        try {

            new java.net.URL(url).toURI();

        } catch (MalformedURLException e) {

            System.out.println("MalformedURLException");
```

```
            } catch (URISyntaxException e) {
                System.out.println("URISyntaxException");
            } finally {
                System.out.println("In Finally");
            }
        }
    }
```

After we compile the code above and produce the disassembly listing, for the “Exception table” we will get something like the following.

Exception table:

from	*to*	*target*	*type*
0	*12*	*23*	*Class java/net/MalformedURLException*
0	*12*	*43*	*Class java/net/URISyntaxException*
0	***12***	***63***	***any***
23	***32***	***63***	***any***
43	***52***	***63***	***any***

We can see that the compiler added at least three additional mappings. Note that “any” is almost equivalent to “java/lang/Throwable”, “any” wording is most likely chosen intentionally, to indicate that the mapping is for a finally block. In the case above, one mapping corresponds to the body of the try; and two more mappings correspond to the bodies of the two catch blocks. All of them, independent of the result, should end up in the finally block.

In summary, the “Exception table” is an important part the *.class file that exists for every method that has at least one try-catch-block (of any form) in it. It marksup and maps the try, catch, and finally block borders.

athrow

This instruction is a JVM instruction level representation of the Java language throw statement. The byte code that corresponds to the instruction is 0xBF.

Though this instruction explicitly throws an exception, other JVM instructions may do it as well, when an abnormal condition is detected.

JIT Compilation

The basic idea of JIT compilation and correspondingly JIT compilers is relatively simple. If there is a pattern of frequently executed byte code, it can be compiled into some native code and after that executed much faster.

Unfortunately, there is an inherent contradiction in the idea of using JIT compilers for exceptions.

Exceptions thrown from a program are not supposed to form any frequent pattern. The most probable candidate for it is the case where exceptions are used for flow control, but as we will discuss later, this is a well-known design anti-pattern and exceptions should not be used this way.

There were two papers published on this subject in the early 2000s [5] [6].

Although it is difficult to use the power of JIT compilation for exception handling directly, it still may be used indirectly. For example, methods used to create an adequate exception message may be between “hot” (as identified by the JVM) and there is nothing that prevents them from being compiled by the JIT compiler.

Generics

Generics were introduced in Java 5. This is a powerful mechanism that significantly added to the language type-safety.

It would be tempting to try to utilize generics within the exception mechanism. Unfortunately, generics were implemented in Java in a way that utilizes type erasure and thus has some intrinsic limitations that significantly restrict such usage.

Given this, how can generic classes be used with exceptions?

NOT as a Generic Type

One of the most natural appeals would be to create a parameterized exception class:

```
public class ParameterizedException<T> extends Exception {
    private final T t;
    public ParameterizedException(T t) {
        this.t = t;
    }
    public T getT() {
        return t;
    }
}
```

But an attempt to compile the code shown above will be rejected by the compiler based upon the following (or similar, depending on the Java version) complaint:

```
ParameterizedException.java:1: error: a generic class may not extend java.lang.Throwable
public class ParameterizedException<T> extends Exception {
                                                ^
1 error
```

The reason for this lies in the fact that the Java language uses type erasure to implement generics. There is no problem with using a parameterized type for exceptions. The real challenge is the catch block that becomes an insurmountable problem.

If generic exceptions were allowed to be used in the manner shown above, in addition to other things, it would lead to the following semantics of the catch block:

```
try {
    …
} catch (ParameterizedException<Class1> e1) {
    …
} catch (ParameterizedException<Class2> e2) {
    …
}
```

After the compilation, all information about the precise type of the types would be erased and the code would be converted into something equivalent to:

```
try {
    …
} catch (ParameterizedException e1) {
    …
} catch (ParameterizedException e2) {
    …
}
```

This, in turn, would make it impossible for the compiler to correctly construct an unambiguous exception table in the resulting class file.

The Java language architects could have allowed exception generic types to be used in **throws** and **throw**,

but without the ability to use them in **catch** it would be virtually useless.

As a Type Parameter

Although exceptions cannot be used as a generic type, they can be used, to some extent, as a (bounded) type parameter.

In throws

```
public interface MyInterface<T extends Throwable> {
    public void method() throws T;
}
public class MyException extends Throwable {
}
public class MyClass implements MyInterface<MyException> {
    public void method() throws MyException {
    }
}
```

The code above will compile without a problem.

Generics can be used in a **throws** clause, but there are some specifics to consider that we will discuss later.

In throw

```
public <T extends Throwable> void raise(T t) throws T {
    throw t;
}
```

The code above will also compile and work well without a problem.

NOT in catch

Type parameters cannot be used a **catch** block. To understand why, let us consider the following code.

```
public <T extends IOException,
    U extends IOException> void intercept() {
    try {
        …
    } catch (T t) {
        …
    } catch (U u) {
        …
    }
}
```

The Java compiler after erasure will replace both T and U with IOException. The code will be equivalent to something like:

```
public void intercept() {
    try {
        …
    } catch (IOException t) {
        …
```

```
        } catch (IOException u) {
            …
        }
}
```

Here, we hit a dead end (similar to what occurred with generic types). How will it generate the exception handler table for the class file? Which code handler should be placed first, and which second? The order is critically important in the table. As can be seen, there is no way to accomplish it.

Multithreading

There are two major concerns associated with exceptions in the context of multithreading.

The first concern is what happens and how to handle an exception if an uncaught exception (checked or unchecked) crosses the border of the main() or run() method that is on the top of user's thread call stack. This question was discussed in previous sections.

The second concern is how to intercept an exception and pass this information between threads in a multithreading environment. We discuss this issue below.

Manual Management

If a thread is started from another thread, exceptions thrown in it do not propagate to the parent thread.

If a thread is created and started manually, using core Java classes for concurrency, then exchanging exceptions between threads is not different to exchanging other objects between threads. An exception should be intercepted with an explicit try-catch block, then processed and passed to another thread using means of interthread communication. Or it can be handled using one of the handlers we discussed in the previous chapters with consequent processing and exchange of the information between interested threads.

Our discussion does not cover whether these actions are a good or bad idea; instead, we only discuss how it can be practically implemented. The task can be made easier by handlers, as already discussed in a previous chapter.

Executors

The Java executor services allow users to create and support a pool of threads for executing submitted tasks. It is less generic than manual threads management and provides a higher level of thread safety and convenience.

java.util.concurrent.**ExecutorService** interface is the workhorse of the Java executor services. The classes that implement it provide multiple facilities to run tasks in parallel.

There are two types of tasks that may be submitted: a task using java.lang.**Runnable** and a task using java.util.concurrent.**Callable**.

These two options take a significantly different approach to exception handling.

In the case of the **Runnable,** a user submits a task and can forget about it; the task lives its own life and is fully responsible for what information it exchanges with interested parties, how it handles exceptions thrown, etc.

In the case of the **Callable** interface a user submits a task and is able to get the result of the execution, including returned value and/or thrown exception. This is done via java.util.concurrent.**Future** interface, which is an important element of Java's concurrency framework.

Runnable

Using Runnable interface for executing tasks from the perspective of exception handling is not significantly different from manual creation and handling. A task starts as a separate thread, and if there is an unhandled exception, it either should be intercepted by the thread and handled manually in the thread itself, or if it crosses the border of the run() method it may be handled by the handlers registered for that purpose.

The interface has only one method:

```
public interface Runnable {
    public void run();
}
```

The method does not throw any checked exceptions.

Callable

Callable interface provides a significantly more flexible, simplistic and less generic way to handle exceptions.

```
public interface Callable<V> {
    V call() throws Exception;
}
```

Future

The interface provides access to the results of submitting a Callable interface to one of the Executors.

```
public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning)
    V get();
    V get(long timeout, TimeUnit unit);
    boolean isCancelled();
    boolean isDone();
}
```

The method get() is supposed to return the result of the asynchronous computation. If the work was finished by some exception thrown during the computation, it will be wrapped in an instance of the java.util.concurrent.**ExecutionException**. To get access to the original exception one has to call the getCause() method on the wrapper instance.

The method get() is a blocking call.

One also has to keep in mind that the execution can be interrupted, not finished, and may timeout, in this case one may get exceptions that are different to the ExecutionException.

4. Miscellaneous

There are different ways that exceptions can be used. Their use goes beyond core and advanced technical concepts. Some of these ways require user discretion and may or may not be used. Below, we cover the most popular uses of them.

Checked as Unchecked

Checked exceptions are verified at compile time, at runtime for the JVM they are indistinguishable from unchecked exceptions.

Therefore, if there is a way to trick the Java compiler into accepting a checked exception as an unchecked exception, it will propagate through all intermediate methods without being declared to the most enclosing adequate catch block or to the thread handler.

There are several ways this can be achieved.

Generics

The first way to throw a checked exception as unchecked is by using generics. This is the most recent and most popular way.

Take a look at the following code.

Example 1:

```
import java.io.IOException;

public class CheckedAsUnchecked {

    private static <T extends Exception> void f(Exception e)

            throws T {

        throw (T) e;

    }

    public static void main(String[] args) {

        f(new IOException("Thrown as unchecked"));

    }

}
```

When the code above is compiled, it will give you a warning:

```
Note: CheckedAsUnchecked.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.
```

But nevertheless, it will compile successfully, and after it runs you will get the following output:

```
Exception in thread "main" java.io.IOException: Thrown as unchecked

        at CheckedAsUnchecked.main(CheckedAsUnchecked.java:17)
```

In this case we clearly see a checked exception java.lang.IOException thrown from a method (which does not declare the IOException type in its throws clause) as an unchecked one.

Why is this?

If you change the java.lang.Exception class to java.lang.Throwable, compile and run the following code:

Example 2:

```
import java.io.IOException;

public class CheckedAsUnchecked {
```

```
    private static <T extends Throwable> T f(Throwable e)
            throws T {
        throw (T) e;
    }
    public static void main(String[] args) {
        f(new IOException("Thrown as unchecked"));
    }
}
```

It will work again:

```
Exception in thread "main" java.io.IOException: Thrown as unchecked
        at CheckedAsUnchecked.main(CheckedAsUnchecked.java:17)
```

But, if you slightly change the parameters definition in the method, it will stop working.
Example 3:

```
import java.io.IOException;
public class CheckedAsUnchecked {
    private static <T extends Exception> T f(
            T t) throws T {
        throw (T) t;
    }
    public static void main(String[] args) {
        f(new IOException("Thrown as unchecked"));
    }
}
```

You will get the following compilation error:

```
error: unreported exception IOException; must be caught or declared to be thrown
        f(new IOException("Thrown as unchecked"));
          ^
1 error
```

Why does this occur?

The official Java language Specification [7] explains why it happens in paragraph “18.4 Resolution” of the “Chapter 18. Type Inference.”

> *“Otherwise, if the bound set contains throws αi, and the proper upper bounds of αi are, at most, Exception, Throwable, and Object, then Ti = RuntimeException.”*

As the paragraph states, if a bounded type parameter proper upper bound is **java.lang.Throwable** or **java.lang.Exception**, in the throws clause it is inferred to **java.lang.RuntimeException**. This is what happens in Example 1 and 2 above: the compiler does not complain because a runtime exception should not be caught or declared to be thrown.

If there is another hint that allows the compiler to infer the parameter, it may do it to a different type. This is what happens in Example 3. The compiler used method parameter in the call to infer the throws clause exception type exactly to java.io.IOException, which should be either caught or declared to be thrown.

But why was this behavior introduced into the language? Java 5-7 and earlier versions of Java 8 compiler would infer <Throwable> and <Exception> correspondingly in the Example 1 and 2.

Later, as the author of the corresponding report [8] states, it was changed most likely because in some cases

(mentioned in his report) empty Lambda may be inferred to throw a checked exception.

This language feature, as we will see in later chapters, may be used in different situations to bypass compiler restrictions on checked exceptions.

Class newInstance()

This method is based on the fact that **java.lang.Class** method newInstance() transparently throws any exception raised during default constructor call, be it checked or unchecked. The check is not performed during the compile time.

public T newInstance() throws Instantiation InstantiationException, IllegalAccessException

The description of the method says the following [9]:

> *"Note that this method propagates any exception thrown by the nullary constructor, including a checked exception. Use of this method effectively bypasses the compile-time exception checking that would otherwise be performed by the compiler."*

In order to trick the code into throwing an arbitrary exception, the exception can be passed to the default constructor and thrown from there. Some implementations of this trick use static variable to store and pass it. In this case, the static raise() method should be synchronized. We use a slightly different approach, one that utilizes a thread local variable. In this case the method does not have to be synchronized.

```
import java.io.IOException;
public class CheckedAsUnchecked {
    private static ThreadLocal<Throwable> local
        = new ThreadLocal<>();
    private CheckedAsUnchecked() throws Throwable {
        if (local.get() != null) throw local.get();
    }
    public static void raise(Throwable throwable) {
        local.set(throwable);
        try {
            CheckedAsUnchecked.class.newInstance();
        } catch (InstantiationException
                | IllegalAccessException e) {
            }
    }
    public static void main(String[] args) {
        CheckedAsUnchecked.raise(
                new IOException("Thrown as unchecked"));
    }
}
```

Compile it:

```
javac CheckedAsUnchecked.java
```

Now, run it:

```
java CheckedAsUnchecked
```

Should get this (or similar) result:

Exception in thread "main" java.io.IOException: Thrown as unchecked

at CheckedAsUnchecked.main(CheckedAsUnchecked.java:17)

The code in newInstance() internally uses **sun.misc.Unsafe** method, but it is executed from the **java.lang.Class** class and this way has less security restrictions. We avoid all awkwardness associated with the use of Reflection API.

As of Java 9, this method is deprecated. There is no guarantee that in the future the method may not simply stop working. In addition, the fact that it utilizes the sun.misc.Unsafe class does not guarantee that all implementations will do the same. But the description given in the official API promises that the call to the default constructor will propagate any exception to the caller. As of Java 15 the method still works.

Unsafe throwException()

Although the official documentation does not promise (directly or indirectly) that the sun.misc.Unsafe class will be used in the future to back up the Class newInstance(), as of Java 16 it is still used. It can also be used on its own.

The call is made on the following method of **sun.misc.Unsafe** class:

```
/** Throw the exception without telling the verifier. */
public native void throwException(Throwable ee);
```

The class sun.misc.Unsafe contains multiple native (and non-native) methods that allow users to violate restrictions imposed by the Java language. This class should be used with great caution and a full understanding of how it works.

The method **throwException()** allows throwing any exception (passed as the method's parameter) as unchecked. The compiler will accept it and only at runtime will the exception be caught by the appropriate handler (be it try-catch or thread handler).

Although the method provides this capability, it cannot be called directly. Certain tricks should be utilized beforehand.

The method is not static, it should be called on an instance of the sun.misc.Unsafe class. There is only one, private constructor in the class. In addition, there is a private field of sun.misc.Unsafe, with the name theUnsafe.

```
private static final Unsafe theUnsafe = new Unsafe()
```

As one can see, the field is initialized during the static initialization of the class.

To assess the field there is a method getUnsafe(). But the problem with the method is that it has a security check:

```
public static Unsafe getUnsafe() {
    Class<?> caller = Reflection.getCallerClass();
    if (!VM.isSystemDomainLoader(caller.getClassLoader()))
        throw new SecurityException("Unsafe");
    return theUnsafe;
}
```

The meaning of this security check is that if the method is called not from core Java classes, the check will fail.

To bypass this security check Java Reflection API may be utilized. It is actively used in different Java frameworks and libraries, and normally is not disabled.

Example 1:

```
…
Field field = Unsafe.class.getDeclaredField("theUnsafe");
field.setAccessible(true);
Unsafe theUnsafe = (Unsafe) field.get(null);
theUnsafe.throwException(new CheckedException("Thrown as unchecked"));
…
```

Example 2:

```
…
Constructor<Unsafe> constructor = Unsafe.class
      .getDeclaredConstructor();
constructor.setAccessible(true);
Unsafe unsafe = constructor.newInstance();
unsafe.throwException(new CheckedException("Thrown as unchecked"));
…
```

In the first example above, we use Java Reflection API to access a private field. In the second example, we use it to access a private constructor. Two points should be mentioned in connection with these tricks. First, there is absolutely no guarantee that in the environment where such code will be running the security restriction will let it run easily and no security exception will be thrown. Second, the **sun.misc.Unsafe** class is already deprecated and there is absolutely no guarantee that in future Java releases the code will not be completely removed. As of Java 16, the methods still work.

In earlier versions of Java there was, at a minimum, another way to accomplish these tasks but without accessing the **sun.misc.Unsafe** class directly and explicitly calling the throwException() method. It is, however, called on the background.

Bridge throwException()

Similarly, the **sun.corba.Bridge** method throwException() throws any exception passed to it as a parameter without a compile time check. In order to access this method, Corba classes must be deployed.

```
public final void throwException(Throwable var1)
```

The method, in turn, calls sun.misc.Unsafe throwException(). In order to get a valid instance of the Bridge class one should make a call to the following method:

```
public static final synchronized Bridge get()
```

The method may fail due to security limitations. Other than this, the class has a very simple use.

```
import sun.corba.Bridge;
…
Bridge.get().throwException(
    new CheckedException("Thrown as unchecked"));
…
```

As of Java 9, Java EE and Corba modules were deprecated, and as of Java 11 they were removed from the JDK, so this method is no longer available.

Thread stop()

This method could have been used only in versions of Java prior to Java 8. Since then, it throws UnsupportedOperationException.

It has been deprecated since Java 1.2, but did work in some cases.

In order to throw an undeclared checked exception, one may use the following code:

```
…
Thread.currentThread().stop(new MyCheckedException());
…
```

Overall, in this method we do not explicitly rely on any vendor specific code, but it is deprecated and has not been supported since Java 8. We may refer to it as conditionally portable in earlier versions of Java.

Similar to Unsafe throwException() it cannot be recommended.

Bytecode Manipulation

As we have already stated, checked exceptions are verified at compile time, but not runtime. This fact may be used in order to trick the JVM to throw an unchecked exception as a checked one.

Let us write the following code:

```
public class CheckedAsUnchecked {
    public static void raise(Throwable t) {
        RuntimeException r = (RuntimeException)t;
        throw r;
    }
}
```

It successfully compiles:

```
javac CheckedAsUnchecked.java
```

In the next step we disassemble and dump the byte code instructions, using **javap** utility.

```
javap -c CheckedAsUnchecked.class
```

The result is:

```
Compiled from "CheckedAsUnchecked.java"
public class CheckedAsUnchecked {
  public CheckedAsUnchecked();
    Code:
      0: aload_0
      1: invokespecial #1 // Method java/lang/Object."<init>":()V
      4: return
  public static void raise(java.lang.Throwable);
    Code:
      0: aload_0
      1: checkcast #7 // class java/lang/RuntimeException
      4: astore_1
      5: aload_1
      6: athrow
}
```

The last two lines:

```
5: aload_1
6: athrow
```

Correspond to:

```
throw r;
```

We should modify the byte code, so that the last two lines would correspond to:

```
throw t;
```

In order to do this, let us take a look at the JVM specification and find instructions set for the corresponding operation.

We find that:

```
aload_1 -> 0x2B
athrow -> 0xBF
```

So, in the CheckedAsUnchecked.class we should look for the following sequence:

2B BF

Given that method parameters are treated as local variable, we have the following:

Since our method is static, not instance, the first reference argument is not **this** reference, but the first method parameter, which means aload_0 would correspond to passing t as an operand to throw statement. This means, in order to achieve our goal aload_1 should be replaced with aload_0.

Instead of:

aload_1 -> 0x2B

athrow -> 0xBF

We should have:

aload_0 -> 0x2A

athrow -> 0xBF

In terms of hex codes, we should find and replace:

2B BF -> 2A BF

Open CheckedAsUnchecked.class with your favorite hex editor (we used HxD), and search “2B BF”. Only one occurrence is found, replace “2B” with “2A” and save your changes.

Now, our class corresponds to the following Java source code:

```
public class CheckedAsUnchecked {
    public static void raise(Throwable t) {
        RuntimeException r = (RuntimeException)throwable;
        throw t;
    }
}
```

If you use one of the popular Java decompilers, it will produce something like the code above. However, the code cannot be compiled.

We can modify it a bit more, getting rid of this odd line of code:

RuntimeException r = (RuntimeException)t;

In the dump it corresponds to:

```
0: aload_0
1: checkcast #7 // class java/lang/RuntimeException
4: astore_1
```

Which corresponds to:

aload_0 -> 2A

checkcast #7 -> C0 00 07

astore_1 -> 4C

All together the sequence “2A c0 00 07 4C” should be found and replaced with five NOPs - “00 00 00 00 00”, (not removed, because in this case the file structure may become corrupted, and we should change other attributes). Again, open CheckedAsUnchecked.class with a hex editor, find this sequence, change it, and save the file.

Now CheckedAsUnchecked.class corresponds to:

```
public class CheckedAsUnchecked {
    public static void raise(Throwable t) {
        throw t;
    }
```

```
}
```

The resulting CheckedAsUnchecked.class when run through any popular decompiler will give a result equivalent to the one above, it will not compile as a Java code. But the class will work at runtime.

Let us test it:

```
import java.io.IOException;

public class Test {

    public static void main(String[] args) {

        CheckedAsUnchecked.raise(

            new IOException("Thrown as unchecked"));

    }

}
```

Compile it:

```
javac Test.java
```

Be sure, while compiling the code above, that CheckedAsUnchecked.class is on the classpath and not recompiled and overwritten. We need the modified version.

Now, run it:

```
java Test
```

You should get this (or a similar) result:

```
Exception in thread "main" java.io.IOException: Thrown as unchecked

        at Test.main(Test.java:4)
```

As one can see, although both methods raise() and method() do not declare IOException, the exception propagates through them to the nearest catch block able to intercept it.

.class Substitution

This method is similar to the previous one, but it does not require any disassembler and/or hex editor.

We use this source code to compile the CheckedAsUnchecked class:

```
public class CheckedAsUnchecked {

    public static void raise(Throwable t) {

    }

}
```

Next, we compile the code, identical to the code from the previous paragraph:

```
import java.io.IOException;

public class Test {

    public static void main(String[] args) {

        CheckedAsUnchecked.raise(

            new IOException("Thrown as unchecked"));

    }

}
```

The code compiles (make sure CheckedAsUnchecked.class is on the class path) without any problem.

If we run the Test class now, it will not throw any exception. Now let us change the CheckedAsUnchecked class as follows:

```
public class CheckedAsUnchecked {

    public static void raise(Throwable t) throws Throwable {
```

```
        throw t;
    }
}
```

After the CheckedAsUnchecked is compiled, we should run (again make sure, the recompiled CheckedAsUnchecked.class is on the classpath) Test without recompilation. If we try to recompile, it will fail.

But running Test works perfectly fine.

```
java Test
```

Should again get this (or a similar) result:

```
Exception in thread "main" java.io.IOException: Thrown as unchecked
        at Test.main(Test.java:4)
```

As one can see, although methods raise() declares Throwable in its throws, method() in Test is unaware of this fact. In runtime an IOException propagates through it to the nearest catch block able to intercept it.

Assertions

Assertions introduced in Java 1.4 provide an exception-based mechanism for testing purposes.

Purpose

The purpose of having assertions is to verify assumptions about a Java program.

This type of testing is different from the unit testing, which is somewhat close in terms of granularity but the two should not be confused. The approach and methods vary significantly.

During the unit testing the testing engine externally tests units with correct outcomes. The result (be it failure or success) of a test is recorded and the testing engine goes on to the next one. There may be more than one assertion in a unit, they test if the unit works consistently; if an assertion fails, this fact should lead to failure of the whole application (or the component).

Fail-Fast Concept

Fail-Fast is the concept that some failures should immediately lead to the failure of the whole system (or component).

Assertions are supposed to be actively used during the development stage. At this stage developers often have a local copy of the software environment, so as soon as a bug is found, the system is brought down, a fix applied, the system restarted and retested for the bug. All of this occurs without (or almost without) affecting other developers.

AssertionError & assert

The key class in the assertion mechanism that is used to raise an assertion is **java.lang.AssertionError**. The class does not have to be used directly, an instance of it created by the JVM and thrown if the assertion mechanism is enabled and a corresponding event happens.

The class has the following constructors:

```
public AssertionError()

public AssertionError(boolean detailMessage)

public AssertionError(char detailMessage)

public AssertionError(double detailMessage)

public AssertionError(float detailMessage)

public AssertionError(int detailMessage)

public AssertionError(long detailMessage)

public AssertionError(Object detailMessage)

public AssertionError(String message, Throwable cause)
```

Another important element of the assertion mechanism is the **assert** key word, which has two forms. The first one:

assert Expression1*;*

The second one:

assert Expression1 *:* Expression2*;*

Where Expression1 is a boolean expression, and Expression2 is a value expression.
The first form is equivalent to:

```
If(assertionsEnabled && Expression1)
    throw new AssertionError();
```

The second form is equivalent to:

```
If(assertionsEnabled && Expression1)
    throw new AssertionError(Expression2);
```

If the assertion mechanism is not enabled, the Expression1 is not evaluated.

To check if the assertions are enabled, the official Java documentation recommends using the following trick, based on side effects of the assert statement.

```
boolean assertsEnabled = false;
assert assertsEnabled = true;
```

Enabling & Disabling

By default, the assertion mechanism is disabled in Java. The reason for this is to provide backward compatibility and avoid problems with the Java code written in the pre-Java 1.4 world.

After the Java code is compiled, the assertion statements are translated, then injected into the compiled code and they stay there. There is no standard way to remove them.

The assertion mechanism is enabled at runtime. There are two ways to enable or disable assertions in Java. Also, there is a special case when assertions are never enabled. This case is rare but may occur therefore should be highlighted.

Compilation

Starting from java 1.4 “assert” became the Java language keyword. Earlier versions (after introduction of assertions) of javac utility required the new code that uses the keyword as an assertion statement to be compiled with a **-source 1.4** flag. Later the javac utility behavior changed, the old code that used “assert” as an identifier was supposed to use **-source 1.3** flag, meanwhile the code with “assert” as assertion statement would compile by default, without using any flag.

JVM startup

The first way to enable/disable assertions is by using special flags, when starting the JVM.

To enable assertions **-enableassertions** or **-ea** flags are used with java utility.

```
java [-enableassertions|-ea] [:<package name>"..." | :<class name>]
```

<package name> takes the form of package-tree expressions and affects the package with all its subpackages.

If the package name is followed by … the assertions are enabled not only in the package, but in its all subpackages as well.

<class name> is the name of the class that starts first after the JVM is up and running.

For example:

```
java -ea:javax.swing.text... MySwingTestApp
```

This example will start the MySwingTestApp with the assertion mechanism enabled in the following Swing packages:

javax.swing.text
javax.swing.text.html
javax.swing.text.html.parser
javax.swing.text.rtf

To disable assertions **-disableassertions** or **-da** flags are used with java utility.

java [-disableassertions|-da] [:<package name>"..." | :<class name>]

<package name> takes the form of package-tree expressions and affects the package with all of its subpackages.

If the package name is followed by … the assertions are disabled not only in the package, but in its all subpackages as well.

<class name> is the name of the class that starts first after the JVM is up and running.

For example:

java -da:javax.swing.text.html... MySwingTestApp

This example will start the MySwingTestApp with the assertion mechanism explicitly disabled in the following Swing packages:

javax.swing.text.html

javax.swing.text.html.parser

Switches that enable or disable assertions do not affect system classes.

To enable assertions in system classes, **enablesystemassertions** or **-esa** flags are used with java utility.

java [-enablesystemassertions | -esa]

To disable assertions in system classes, **disablesystemassertions** or **-dsa** flags are used with java utility.

java [-disablesystemassertions | -dsa]

Flags that enable or disable assertions are applied in the order they are passed to the java utility.

Example 1:

java -ea -da MyApp

This line, when executed, will start the MyApp class with the assertion mechanism disabled.

Example 2:

java -ea -da -ea:javax.swing.text.html... MySwingTestApp

This line, when executed, will start the MyApp class with assertion mechanism disabled for all classes, except for the classes that are in javax.swing.text.html package and its subpackages.

Programmatically

The second way to enable/disable assertions is by doing it programmatically.

The key Java class in enabling/disabling assertions programmatically is java.lang.ClassLoader. It is the key class that is responsible for loading classes to the JVM.

There are several methods that provider a user with the ability to disable/enable assertions or query the values of the corresponding settings.

public void setDefaultAssertionStatus(boolean enabled)

public void setPackageAssertionStatus(String packageName, boolean enabled)

*public void setClassAssertionStatus(*String *className, boolean enabled)*

public void clearAssertionStatus()

boolean desiredAssertionStatus(String className)

While setting the desired assertion status, one must make sure it happens before the corresponding class is loaded and initialized. Flags may be turned on and off, but the state of the class loader assertion settings, at the time when the corresponding class is loaded, will define its behavior.

Changing the assertion status using one of the methods above after the class is loaded and initialized will not have any affect.

Similarly, the method desiredAssertionStatus will return only the current value for the flag. There is no guarantee that the class was or will be loaded with the option corresponding to the flag.

Special case

An important point regarding programming that uses assertions is understanding that independent of the effective assertion status set at JVM startup or during loading the corresponding class the assertion mechanism will be always enable until the Java object is fully initialized.

There is a specific case described in the Java Language Specification where a method or a constructor can be called before the class is initialized.

In this case assertion mechanism is always enabled.

Example (a modified version from the Java Language Specification) [10] [11]:

```
public class A {
        static {
                B.f();
        }
}
public class B extends A {
        public static void main(String[] args) {
        }
        static void f() {
           boolean enabled = false;
          assert enabled = true;
           System.out.println("Asserts "
                        + (enabled ? "enabled" : "disabled"));
        }
}
```

Independent of which flags are used after the code is compiled and class B executed:

java -ea B

Or:

java -da B

It will always return:

Asserts enabled

It appears as if the logic used by the Java language architect was that the code similar to the above is dangerous and may execute in a not fully initialized class. Assertions are often used to check class invariants. When a class is not fully initialized, its invariants may not pass the validity check. Therefore, users should be made aware of this fact. An important point to notice in this context is that users may not even be aware that their code is being executed in such a context – the complexity of the code may unintentionally lead to this.

Guidelines

Design by Contract

Design by Contract is an approach to software development. Its key notions (amongst others) include preconditions, postconditions and invariants.

Java's assert mechanism guidelines recommended by Sun and Oracle follow a simplified version of the design by contract.

Preconditions

Preconditions are conditions that should be true before a method is called. Preconditions can be imposed on the method parameters as well as on the state of the object whose method is being called.

According to guidelines [10] and [11], it is recommended to assert preconditions only for private methods and data controlled privately, and not recommended for public methods and data controlled publicly. The logic behind this is that the developer fully controls the code that calls a private method (or access private data) and thus the fact that the preconditions are not met indicates that the code has mistakes. If the method is public (or data) is public, then throwing an informative exception is a better option. This way the responsibility to provide correct parameters (or take care of the external conditions) is on the calling side and does not indicate that the called code is internally inconsistent and/or broken.

In the case of a method’s parameters, the approach is very straightforward.

Example 1:

```
public class MyMath {
        …
        public double sqrt(double parameter) {
                if (parameter < 0)
                        throw new IllegalArgumentException(
                                "Parameter cannot be negative");
                return Math.sqrt(parameter);
        }
        …
}
```

Example 2:

```
public class MyMath {
        …
        private double sqrt(double parameter) {
                assert (parameter < 0) : "Parameter cannot be negative";
                return Math.sqrt(parameter);
        }
        …
}
```

In the case of the objects state it is a bit trickier.

Postconditions

Postconditions are conditions that should be true after a method is called. Postconditions can be imposed on the method parameters as well as on the state of the object whose method is being called.

The guidelines recommend assert postconditions equally on all types of methods, be they private or public.

Example 1:

```
public class MyBanker {
        …
        public void withdrawFunds(Accoount account,
                BigDecimal amount){
                …
                assert checkNegativeBalance(Account account) :
                        "Balance cannot be negative!";
        }
        …
}
```

Example 2:

```
public class MyBanker {
        …
        private void delete(Account account) {
                …
```

```
        assert !exists(Account account) :
            "Account is still in the system!";
    }
    …
}
```

Invariants

Invariants are conditions that stay the same over some period of time.

There are several types of invariants defined by the guidelines – class invariants, internal invariants, and flow-control invariants.

A class invariant is a condition that stays true, while an instance of the class is in a consistent state. In nature they are like conditions and preconditions at the same time. They should stay true during a method call.

The Java language provides multiple mechanisms to enforce a class instance consistency such as access modifiers, final keyword, monitors, locks, and other advanced mechanism. Nevertheless, sometimes it may make sense to assert a class consistency directly, especially during the development stage.

Example 1:

```
public class MyComplicatedWorker {
    …
    public void doSomething(){
        …
        assert checkConsistency() : "Instance inconsistent!";
        …
    }
    …
}
```

Internal and Control-Flow invariants indicate things should never happen.

Example 2:

```
public class MyCustomsOfficer {
    …
    public void check(){
        …
        If(isDomestic(Flight flight)) {
            …
        } else if isInternational(Flight flight) : (
            …
        } else {
            assert false : "No such thing!";
        }
        …
    }
    …
}
```

Example 3:

```
public class MyGame {
```

```
…
switch( a ) {
        case Choice.ROCK:
              …
                break;
        case Choice.PAPER
              …
                break;
        case Choice.SCISSORS:
                …
                break;
        default:
                assert false : "No such choice!";
        …
}
```

Spring AOP

AOP solutions provide facilities for operating on thrown exceptions as well as on the corresponding objects that throws those exceptions.

There are two major concerns associated with exception usage in AOP.

The first concern is how the aspects are applied when there is an exception thrown from the target method. The second concern is what happens if there is an exception not associated with calling the target method thrown from an aspect, when it is applied.

Legacy API

The pre-Spring 2.0 version of the AOP support was based on a lower-level API that used special interfaces. This API is now obsolete and rarely used. The Spring specification does not recommend it, although the API is still supported for the purpose of compatibility and its description is in the specification.

The exception handling in the legacy AOP API is similar to the handling in the schema based AOP and AspectJ annotation-based approach.

We mention the API, but since it is rarely used these days, we will not go through the details of the exception handling. If needed, the details of how it is implemented can be found in the corresponding version Spring specification and/or API documentation.

XML-based

Generic, schema-based Spring AOP since version 2.0 provides two major ways to intercept a thrown exception and add some extra logic or completely modify the call: **After throwing** and **Around** advice. Other advice (**Before**, **After returning** and **After (finally)**) although not called as a direct result of the target method throwing an exception, they can throw an exception themselves, thus changing the call behavior.

It is important with the XML-based Spring AOP to understand the precedence rules for advice methods in the same aspect. The precedence is determined by the order the advice elements appear in the corresponding <aop:aspect>. **Before** and **Around** advice go first (depending on the declaration order). Then, **After throwing, After returning** and **After (finally)** (they are called in the reverse declaration order), those of them that are defined after the **Around** advice, are called before the return from the **Around** advice (they are called in the declaration order). It is strongly advised to not use two (or more) advice of the same type in one aspect; rather, collapse the advice into one, or separate them into different aspects. We will not cover this option in detail.

If a checked exception is compatible with the throws statement of the advised method, from advice it will be thrown “as is”. If it is incompatible, it will be wrapped into the java.lang.reflect.**UndeclaredThrowableException**, which is a subclass of the RuntimeException.

With different versions of the Spring Framework, the behavior of advice after throwing an exception may

change, so what we describe here is true as of Spring 5.2.7 release.

Before

Before advice allows the user to intercept the moment before the joint point method is called. It is executed before any other advice. It is not a method that is called as a result of some exception thrown from the advised method. But it can throw its own exception, thus changing the execution flow.

If a **Before** advice throws an exception, no other **Before** and/or **Around** advice is started that in the same aspect. If there is an enclosing **Around** advice, it may intercept the exception and process it plus call the advice that should be called before the org.aspectj.lang.**ProceedingJoinPoint** method **proceed()** returns. If there is no enclosing **Around** aspect, the execution goes to available **After throwing** and/or **After (finally)** (depending on the declaration order), otherwise it is thrown back up the interceptor chain.

Around

Around advice let users intercept the whole call of a method/next advice from its beginning to end, including (if any) exception thrown.

The execution goes to a method that has a parameter of the org.aspectj.lang.**ProceedingJoinPoint** interface. In order to proceed with the intercepted method/next advice invocation the user is supposed to call the proceed() method on the parameter. If the call is successful for further normal flow, it should be returned from the proceed() method.

If there is an exception thrown while the proceed() method is called it does not have to be intercepted, and the case will correspond to the normal flow. If the exception is intercepted, some additional logic may be added to the flow and the exception swallowed or re-thrown.

In summary, the **Around** advice gives you full control over the intercepted method/next advice, either continue with normal flow, adding some behavior or completely interrupt further invocation.

After an unhandled exception is thrown from an **Around** advice, the control goes to available **After throwing** and/or **After (finally)** (depending on the declaration order) if it is applicable. If not, the exception is thrown back up the interceptor chain. Remember that at this point the proceed() can be called once, many times or never. Every time it was called all the corresponding aspects and advice were called as well as every time the proceed() method was called.

After returning

After returning advice allows users to intercept the moment the underlying call to the advised method (and all intermediate aspects advice) is returned successfully.

Although the method, by its nature, is not called as a consequence of some exception thrown from the advised method, it can throw its own exception.

If there is an exception thrown from the **After returning** advice it does not trigger an **After throwing** advice for the same pointcut**.** The exception thrown is delayed until it is actually thrown back up the interceptor chain or thrown from the org.aspectj.lang.**ProceedingJoinPoint** method **proceed()**. On that route it may be overridden by an exception thrown from a consequent advice or intercepted and suppressed (in an enclosing **Around** advice).

After throwing

After throwing advice allows users to intercept the moment when the underlying call to the advised method throws an exception.

The method is called as a consequence of some exception thrown from the advised method, it cannot suppress the original exception, but it can override it and throw its own exception.

The new exception is thrown down the interceptor chain, as if it was originally thrown.

After (finally)

After (finally) advice allows users to intercept the moment after the advised method is complete (independent of the result, be it exception or return). This advice does not have to follow the **After returning** or **After throwing** advice, it can have independent logic that does not rely on the method outcome.

There may already be a pending exception from the underlying calls. The method cannot suppress that exception but can override it by throwing its own.

If there is an exception thrown from the **After (finally)** advice it does not trigger an **After throwing** advice for the same pointcut**.** The exception thrown is delayed until it is actually thrown back up the interceptor chain or thrown from the org.aspectj.lang.**ProceedingJoinPoint** method **proceed()**. On that route it may be overridden by an exception thrown from a consequent advice or intercepted and suppressed (in an enclosing **Around** advice).

AspectJ-style

AspectJ is an aspect-oriented Java language extension, that is similar to XML-based Spring AOP with regards to what you can do with the result of a method invocation, but overall, it is more powerful in many aspects and gives you significantly more options regarding what you can do with thrown exceptions. In order to utilize it, one requires a special compiler and/or a weaver.

AspectJ is partially implemented in the Spring AOP AspectJ annotation style.

AspectJ defines three kinds of advice: **before**, **around** and **after** (plus two subtypes: **after returning** and **after throwing**). The order of the advice (for the same join point) in the same aspect is defined by the order of how they appear in the aspect, which is similar to the Spring AOP XML-based approach. The rules are more complicated than just following the lexical ordering, but this is the key factor taken into consideration. The order of aspects should be explicitly specified, or it is undefined.

Spring AOP AspectJ-style uses annotations to declare advice. The five annotations are: **@Before**, **@Around @AfterReturning**, **@AfterThrowing**, and **@After** that correspond to AspectJ **before**, **around**, **after returning**, **after throwing** and **after**.

Contrary to the AspectJ and Spring AOP XML-based approach, the order of the advice (for the same join point) in the same aspect is fixed. It is as follows: **@Around**, **@Before**, **@AfterReturning**, **@AfterThrowing** and **@After**. It is important to note that **@Around** (if present) embraces all other calls.

As in the case of XML-based Spring AOP, it is strongly advised to not use two (or more) annotated advice of the same type in one aspect, but rather collapse such advice into one, or separate them into different aspects. Therefore, this situation will not be covered.

Similar to schema-based advice, if a checked exception is compatible with the throws statement of the advised method, it will be thrown “as is”. If it is incompatible, it will be wrapped into the java.lang.reflect.**UndeclaredThrowableException**, which is a subclass of the RuntimeException.

With different versions of the Spring Framework, the behavior of annotated advice after throwing an exception may change. These details are true as of the Spring 5.2.7 release.

@Around

@Around annotation annotates the aspect’s method that (similar to Spring AOP Around advice) allows users to intercept the whole joint point method call, including (if any) exception thrown.

If there is an exception thrown from **@Around** advice, no other advice is called in the same aspect and the exception is thrown back up the interceptor chain.

@Before

@Before advice is executed before each joint point and is similar to Spring AOP **Before** advice. If there is an exception thrown from **@Before** advice, it may be still intercepted in the remaining part of **@Around** (if it is defined) and processed, if no **@Around** is defined, the exception is thrown back up the interceptor chain.

@AfterReturning

@AfterReturning advice is executed when the joint point method call returns successfully and is similar to Spring AOP **After returning** advice. If there is an exception thrown from **@AfterReturning** advice, the exception is delayed and the next method in an interceptor chain (**@After** or remaining part of **@Around**) is called. If there are no more advice methods in the current aspect to call, the exception is thrown back up the interceptor chain.

@AfterThrowing

@AfterThrowing annotation annotates the aspect’s method that let users intercept the moment when the joint point method call throws an exception.

Similar to its counterpart from Spring AOP, **After throwing**, the method is called as a consequence of some exception thrown from the joint point method call, it cannot suppress the original exception, but it can override it and throw its own exception.

The new exception is thrown down the interceptor chain, as if it was originally thrown.

@After

@After annotation annotates the aspect’s method that allows users to intercept the end of the underlying joint point method call and is similar to Spring AOP **After (finally)** advice. This method is called (if applicable) independent of the fact is there was or was not an exception thrown from the underlying joint point method call, it does not intentionally intercept any activity associated with any exception thrown.

It can, however, throw its own exception. The exception may still be intercepted in the remaining part of **@Around** (if it is defined) and processed, if no **@Around** is defined, the exception is thrown back up the interceptor chain.

Functional Programming

There are several aspects associated with Functional Programming. Amongst a number of purposes, functional programming when introduced into Java provided convenient syntax.

Three challenges

Unfortunately, checked exceptions do not fit well into the idea of compact and convenient code typical of functional programming.

And, in addition, functional programming also presents two major technical (and one aesthetic) challenges with regards to using exceptions.

Skip It

Let us take a look at the following example and call it Scenario 1:

You have a task of parsing a set of string tokens and printing them in a specific format. If one of them is in the wrong format, the program should skip it and try to process the next token.

Example 1 (Scenario 1):

```
private void print(List<String> input) {
    input.forEach(str -> {
        SimpleDateFormat format =
            new SimpleDateFormat("mm-dd-yyyy");
        System.out.println(format.parse(str));
        // Skip to the next on exception
    });
}
```

After you try to compile the code above, you will again get something like the following.

```
error: unreported exception ParseException; must be caught or declared to be thrown
        System.out.println(format.parse(str));
                                              ^
1 error
```

You should make sure that the code is compilable, and make sure that if there is an exception thrown while parsing the program does not stop but instead proceeds to the next record.

Therefore, the first challenge is how to skip to the next record in a cycling processing, while using functional programming.

Pass It Up

Another example is similar to the one above and can take place in scenarios that require processing to stop on failure to parse one of the tokens, and in addition, the problem should immediately be reported to the user. An exception thrown in the parsing method should be passed up to the level where it can be handled accordingly. Let us call this Scenario 2.

Example 2 (Scenario 2):

```
private void print(List<String> input) throws ParseException {
    input.forEach(str -> {
        SimpleDateFormat format =
            new SimpleDateFormat("mm-dd-yyyy");
        System.out.println(format.parse(str));
    });
}
```

After you try to compile the code above, you will get something like the following.

```
error: unreported exception ParseException; must be caught or declared to be thrown
```

```
        System.out.println(format.parse(str));
                                                ^
1 error
```

The code does not compile. This is because the abstract method defined in the functional interface Consumer accepted by the forEach() method does not throw any checked exceptions.

```
void accept(T t);
```

Although it is not self-evident from Example 2, we have the following chain of calls:
print() -> Consumer.accept() -> SimpleDateFormat.parse()
Therefore, if there is a java.text.ParseException (checked) exception thrown during the parsing, it is literally trapped inside the enclosing accept() method and cannot propagate up the call stack.

If it is an unchecked exception, everything goes smoothly, and it can be easily thrown up the call stack. But, if it is a checked exception, it is not so easily done.

In this scenario, you again have to make sure that the code compiles and the exceptional situation is reported up the call stack.

In summary, the second challenge is how to pass information about a checked exception up the call stack in a deeply nested process, while using functional programming.

Make It Compact

The third challenge, mentioned above, is how to make the code neat and compact. It is not an exact science, but rather an art. But still there are some widely accepted methods how to achieve this goal.

The beauty of using functional interfaces (in the same way as anonymous classes) is that the required programming logic may be implemented right in the place where it is needed and only there. If it is not needed anywhere else, there is no need to move it to a separate method or class. While we want to keep the processing logic where it is, the awkwardness associated with **try-catch** logic should be externalized and, if possible, generalized.

How to make the code compact and nice looking in general while using functional interfaces is a separate subject and is not the subject of this discussion. In the context of our current discussion, we are only talking about the corresponding exception handling **try-catch** logic.

We will go through some of the most common techniques used for this.

Lambdas

try-catch

Let us take the first example above and surround the call that may throw ParseException with the corresponding with try-catch.

Example 1 (Scenario 1):

```
private void print(List<String> input) {
    input.forEach(str -> {
        try {
            SimpleDateFormat format =
                new SimpleDateFormat("mm-dd-yyyy");
            System.out.println(format.parse(str));
        } catch (ParseException e) {
            e.printStackTrace();
            // Skip to the next on exception
        }
    });
}
```

We solved the main problem – the code compiles and does what is required. But the code still does not look neat and compact so this needs to be addressed.

Let us take the second example above and again surround the call that may throw ParseException with the

corresponding with try-catch.

Example 2 (Scenario 2):

```
private void print(List<String> input) throws ParseException {
    input.forEach(str -> {
        try {
            SimpleDateFormat format =
                new SimpleDateFormat("mm-dd-yyyy");
            System.out.println(format.parse(str));
        } catch (ParseException e) {
            // What shall we do here ???
        }
    });
}
```

The code compiles, there is some progress, but it does not do what it should. The information about the exception situation is not passed up the stack.

Re-Throw as Unchecked

In the previous example we surrounded the code that may throw ParseException with try-catch block, but still did not solve the problem of passing the exception information up the stack call.

One possible way to solve the problem is to do the following.

Example 1 (Scenario 2):

```
private void print(List<String> input) {
    input.forEach(str -> {
        try {
            SimpleDateFormat format =
                new SimpleDateFormat("mm-dd-yyyy");
            System.out.println(format.parse(str));
        } catch (ParseException e) {
            throw new RuntimeException(e);
            // throw new MyRuntieException(e);
        }
    });
}
```

The checked exception caught in the try-catch block is wrapped into an unchecked one and then re-thrown up the stack. For this purpose, the RuntimeException class may be used, or some other user defined runtime exception. The method signature is changed as well, it does not require ParseException in its throws clause.

What we actually did here is remove a checked exception and convert it into an unchecked one.

Regular Method

We still have to make the code in both examples look neat and compact. The most intuitive solution is to move it to a separate method.

Example 1 (Scenario 1):

```
private void print(List<String> input) {
    input.forEach(str -> parse(str));
}
```

```
private void parse(String str) {
        try {
                SimpleDateFormat format =
                        new SimpleDateFormat("mm-dd-yyyy");
                System.out.println(format.parse(str));
        } catch (ParseException e) {
                e.printStackTrace();
        }
}
```

Example 2 (Scenario 2):

```
private void print(List<String> input) {
        input.forEach(str -> parse(str));
}
private void parse(String str) {
        try {
                SimpleDateFormat format =
                        new SimpleDateFormat("mm-dd-yyyy");
                System.out.println(format.parse(str));
        } catch (ParseException e) {
                throw new RuntimeException(e);
        }
}
```

We got rid of the awkwardness associated with the try-catch block in the Lambda expression and moved it to a separate method.

Another code improvement may be done by using a method references.

Example 3 (Scenario 1):

```
private void print(List<String> input) {
        input.forEach(this::parse);
}
private void parse(String str) {
        try {
                SimpleDateFormat format =
                        new SimpleDateFormat("mm-dd-yyyy");
                System.out.println(format.parse(str));
        } catch (ParseException e) {
                e.printStackTrace();
        }
}
```

Example 4 (Scenario 2):

```
private void print(List<String> input) {
```

```
        input.forEach(this::parse);
    }
    private void parse(String str) {
        try {
            SimpleDateFormat format =
                new SimpleDateFormat("mm-dd-yyyy");
            System.out.println(format.parse(str));
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }
```

As already mentioned, one of the advantages of functional programming is that it provides facility to define logic that is only used in a particular situation, thus reducing the number of functions, classes and .java files. In our case, we moved the whole logic away from the forEach to a separate method. Ideally, we should move away only the awkwardness (associated with exception handling), leaving the core logic in its place.

Wrapping & Generics

In order to separate the exception handling awkwardness from the core logic we need both a separate method (as in the previous example), and also a functional interface. The interface encapsulates the logic that throws a checked exception, where as the method wraps the logic that throws exception and returns a functional interface that does not throw a checked exception.

Example 1 (Scenario 1):

```
…
@FunctionalInterface
public interface ParserConsumer<T, E extends Exception> {
    void accept(T t) throws E;
}
…
private void print(List<String> input) {
    input.forEach(wrapper(str -> {
        SimpleDateFormat format = new SimpleDateFormat(
            "mm-dd-yyyy");
        System.out.println(format.parse(str));
    }));
}
static <T> Consumer<T> wrapper(
        ParserConsumer<T, ParseException> consumer) {
    return i -> {
        try {
            consumer.accept(i);
        } catch (ParseException e) {
            e.printStackTrace();
        }
```

```
        };
}
…
```

Example 2 (Scenario 2):

```
…
@FunctionalInterface
public interface ParserConsumer<T, E extends Exception> {
        void accept(T t) throws E;
}
…
private void print(List<String> input) {
        input.forEach(wrapper(str -> {
                SimpleDateFormat format = new SimpleDateFormat(
                        "mm-dd-yyyy");
                System.out.println(format.parse(str));
        }));
}
static <T> Consumer<T> wrapper(
                ParserConsumer<T, ParseException> consumer) {
        return i -> {
                try {
                        consumer.accept(i);
                } catch (ParseException e) {
                        throw new RuntimeException(e);
                }
        };
}
…
```

The next step in further generalizing the approach is to move the wrapper method from the class where it is used to a more general utility class. This helps us not only get rid of a method in our class, but also reuse it (if needed) in other classes.

Further Improvement

In Java 8 static and default methods, we were introduced to interfaces that made possible further improvements in exception handling in Lambda functions.

The method described above can be moved to a static or default method of the functional interface shown in the previous section.

In the case of a static method, the logic can simply be moved there from the corresponding method.

Example 1 (Scenario 1):

```
…
@FunctionalInterface
public interface ParserConsumer<T, E extends Exception> {
        void accept(T t) throws E;
        static <T> Consumer<T> wrapper(
```

```
                ParserConsumer<T, ParseException> consumer) {
            return i -> {
                try {
                    consumer.accept(i);
                } catch (ParseException e) {
                    e.printStackTrace();
                }
            };
        }
}
...
private void print(List<String> input) {
    input.forEach(ParserConsumer
        .wrapper(str -> {
            SimpleDateFormat format = new SimpleDateFormat(
                "mm-dd-yyyy");
            System.out.println(format.parse(str));
    }));
}
```

Example 2 (Scenario 2):

```
…
@FunctionalInterface
public interface ParserConsumer<T, E extends Exception> {
    void accept(T t) throws E;
    static <T> Consumer<T> wrapper(
                ParserConsumer<T, ParseException> consumer) {
            return i -> {
                try {
                    consumer.accept(i);
                } catch (ParseException e) {
                    throw new RuntimeException(e);
                }
            };
        }
}
...
private void print(List<String> input) {
    input.forEach(ParserConsumer
        .wrapper(str -> {
            SimpleDateFormat format = new SimpleDateFormat(
                "mm-dd-yyyy");
            System.out.println(format.parse(str));
    }));
```

```
}
```

In the case of a default method, the logic is bit trickier.

The forEach() method calls accept() method of the corresponding Consumer interface. In all the examples above we redefined this abstract method, but it does not have to be so. Some other method can be used, from the accept() method.

Example 1 (Scenario 1):

```
@FunctionalInterface
public interface ParserConsumer<T> extends Consumer<T> {
    @Override
    default void accept(T t) {
        try {
            acceptWithException(t);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    };
    void acceptWithException(T t) throws ParseException;
}
...
private static void print(List<String> input) {
    input.forEach((ParserConsumer<String>) str -> {
        SimpleDateFormat format = new SimpleDateFormat(
            "mm-dd-yyyy");
        System.out.println(format.parse(str));
    });
}
```

Example 2 (Scenario 2):

```
@FunctionalInterface
public interface ParserConsumer<T> extends Consumer<T> {
    @Override
    default void accept(T t) {
        try {
            acceptWithException(t);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    };
    void acceptWithException(T t) throws ParseException;
}
...
private static void print(List<String> input) {
```

```
        input.forEach((ParserConsumer<String>) str -> {
            SimpleDateFormat format = new SimpleDateFormat(
                "mm-dd-yyyy");
            System.out.println(format.parse(str));
        });
    }
```

In the examples above, the class ParserConsumer interface has all the logic. The approach can be further generalized.

A Trick

Recall the previously mentioned trick based on the Java language generics feature. As we indicated, by using a generics mechanism, a checked exception may be thrown at runtime bypassing the Java compiler limitations exposed on such usage.

The easiest way to do this is to simply add this logic to the functional interface in a static method.

Example 1 (Scenario 2):

```
@FunctionalInterface
public interface ParserConsumer<T> extends Consumer<T> {
    @Override
    default void accept(T t) {
        try {
            acceptWithException(t);
        } catch (ParseException e) {
            rethrow(e);
        }
    };
    void acceptWithException(T t) throws ParseException;
    @SuppressWarnings("unchecked")
    static <T extends Throwable> void
        rethrow (Throwable t) throws T {
        throw (T) t;
    }
}
...
private void print(List<String> input) {
    input.forEach(
        (ParserConsumer<String>) str -> {
            SimpleDateFormat format = new SimpleDateFormat(
                "mm-dd-yyyy");
            System.out.println(format.parse(str));
    }
}
```

The advantage of this approach is that the exception will “travel” up the stack to the nearest (if any) corresponding **try-catch** block able to intercept it. The exception will bypass all intermediate methods even

those that do not declare the ParseException in their throws clauses.

This approach can be further externalized and generalized; the method that re-throws the exception as unchecked does not necessarily have to be in the same interface of class.

This approach is becoming more and more popular and there are different coding practices that rely upon it.

Another point to notice is that any techniques that throw checked exceptions as unchecked can be used, not necessarily only the one based on generics.

3rd Party Libraries

There are two stable (there were several that were introduced but then disappeared) 3r party libraries that provide (in addition to other features) facilities on how to handle exceptions in Lambda expressions.

Vavr (until April 2017 – **Javaslang,** www.vavr.io) is a Java library that facilitates the usage of data structures and functional programming.

Functional Java (www.functionaljava.org) is another Java library that serves the same purpose. It is an open-source product.

Streams

Using Java streams is similar to using Lambda expressions in cyclic operations. But there is a significant difference; in a cyclic operation you can simply skip to the next record, whereas in a stream you are forced to return a value, or as an alternative, to interrupt the streaming operation.

Let us slightly modify the examples from previous sections.

The new requirement for the first case is to take a set of string tokens, parse them in a specific format, print them and return the list of tokens that conform to the format. If one of them is in a wrong format, the program should skip it and try to process the next token.

The new requirement for the second case is almost the same as the first one: the list of string tokens that conform to the format should be returned, but if there is a parsing exception, the processing should be interrupted, and the exception should be passed up the calling stack.

try-catch

Example 1 (Scenario 1):

```
private List<String> print(List<String> input) {
    return input.stream().map(str -> {
        try {
            SimpleDateFormat format = new SimpleDateFormat(
                "mm-dd-yyyy");
            System.out.println(format.parse(str));
            return str;
        } catch (ParseException e) {
            e.printStackTrace();
            // return ?;
        }
    }).collect(Collectors.toList());
}
```

After you try to compile the code above, you will get something like the following.

```
error: method map in interface Stream<T> cannot be applied to given types;
        return input.stream().map(str -> {
                                  ^
  required: Function<? super String,? extends R>
  found:    (str)->{ t[...]; } }
  reason: cannot infer type-variable(s) R
```

(argument mismatch; ***bad return type in Lambda expression***

missing return value*)*

where R,T are type-variables:

R extends Object declared in method <R>map(Function<? super T,? extends R>)

T extends Object declared in interface Stream

2 errors

Surrounding the mapping with a try-catch block is the first intuitive step, but it does not solve the problem. What can we do if we want to proceed to the next record? It is evident from the verbose message above that the return value should be provided.

The most common approach is to return some magic value and then filter this value out.

Example 2 (Scenario 1):

```
private List<String> print(List<String> input) {
    return input.stream().map(str -> {
        try {
            SimpleDateFormat format = new SimpleDateFormat(
                "mm-dd-yyyy");
            System.out.println(format.parse(str));
            return str;
        } catch (ParseException e) {
            e.printStackTrace();
            return null;
        }
    }).filter(str -> str != null)
        .collect(Collectors.toList());
}
```

The problem here is that in some scenarios a null value is a valid value, therefore filtering it out may lead to a loss of information. To avoid this problem, some other specific value may be returned and records with this value filtered out.

Re-Throw as Unchecked

As in the case of Lambdas, re-throwing an intercepted checked exception as an unchecked works perfectly fine as long as you consider it to be a valid approach for your scenario.

Example 1 (Scenario 2):

```
private List<String> print(List<String> input) {
    return input.stream().map(str -> {
        try {
            SimpleDateFormat format = new SimpleDateFormat(
                "mm-dd-yyyy");
            System.out.println(format.parse(str));
            return str;
        } catch (ParseException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
```

```
                // throw new MyRuntieException(e);
            }
        }).collect(Collectors.toList());
    }
```

Regular Method

Further beautification of the streaming code may be again done with extracting some activity to a separate method.

Example 1 (Scenario 1):

```
private List<String> print(List<String> input) {
        return input.stream().map(str -> parse(str))
                .filter(str -> str != null)
                .collect(Collectors.toList());
}
private String parse (String str) {
        try {
                SimpleDateFormat format = new SimpleDateFormat(
                        "mm-dd-yyyy");
                System.out.println(format.parse(str));
                return str;
        } catch (ParseException e) {
                e.printStackTrace();
                return null;
        }
}
```

Example 2 (Scenario 2):

```
private List<String> print(List<String> input) {
        return input.stream().map(str -> parse(str))
                .filter(str -> str != null)
                .collect(Collectors.toList());
}
private String parse (String str) {
        try {
                SimpleDateFormat format = new SimpleDateFormat(
                        "mm-dd-yyyy");
                System.out.println(format.parse(str));
                return str;
        } catch (ParseException e) {
                e.printStackTrace();
                throw new RuntimeException(e);
                // throw new MyRuntieException(e);
```

```
    }
}
```

Wrapping & Generics

As in the case of Lambda functions, the next step to further improve the code is using wrappers and generics.

The difference is that in case of a Lambda function (or a method reference) in a cycle we used Consumer interface that operates through side effects and its main method accept() returns void, whereas in the case of streams Function interface is used, so its main method accept() has a return value.

We will again need a separate method and a separate functional interface, but as already mentioned, the type of interface will be different.

Example 1 (Scenario 1):

```
@FunctionalInterface
public interface ParserFunction<T, R, E extends Exception> {
    R apply(T t) throws E;
}
…
private static List<String> print(List<String> input) {
    return input.stream()
        .map(wrapper(str -> {
            SimpleDateFormat format = new SimpleDateFormat(
                "mm-dd-yyyy");
            System.out.println(format.parse(str));
            return str;
    })).filter(str -> str != null)
            .collect(Collectors.toList());
}
static <T, R> Function<T, R> wrapper(
        ParserFunction<T, R, ParseException> f) {
    return t -> {
        try {
            return f.apply(t);
        } catch (ParseException e) {
            e.printStackTrace();
            return null;
        }
    };
}
…
```

Example 2 (Scenario 2):

```
…
@FunctionalInterface
public interface ParserFunction<T, R, E extends Exception> {
```

```
        R apply(T t) throws E;
}
...
private static List<String> print(List<String> input) {
        return input.stream()
                .map(wrapper(str -> {
                        SimpleDateFormat format = new SimpleDateFormat(
                                "mm-dd-yyyy");
                        System.out.println(format.parse(str));
                        return str;
        })).collect(Collectors.toList());
}
static <T, R> Function<T, R> wrapper(
                ParserFunction<T, R, ParseException> f) {
        return t -> {
                try {
                        return f.apply(t);
                } catch (ParseException e) {
                        throw new RuntimeException(e);
                }
        };
 }
 …
```

Further Improvement

As with Lambda functions, since Java 8 similar facilities can be used for streams.

First, the logic from the static method that converts the functional interface can be moved to the interface class.

Example 1 (Scenario 1):

```
 …
@FunctionalInterface
public interface ParserFunction<T, R, E extends Exception> {
        R apply(T t) throws E;
        static <T, R> Function<T, R> wrapper(
                        ParserFunction<T, R, ParseException> f) {
                return t -> {
                        try {
                                return f.apply(t);
                        } catch (ParseException e) {
                                e.printStackTrace();
                                return null;
                        }
```

```
            };
        }
}
…
private static List<String> print(List<String> input) {
        return input.stream()
                .map(ParserFunction.wrapper(str -> {
                        SimpleDateFormat format = new SimpleDateFormat(
                                "mm-dd-yyyy");
                        System.out.println(format.parse(str));
                        return str;
        })).filter(str -> str != null)
                        .collect(Collectors.toList());
}
```

Example 2 (Scenario 2):

```
…
@FunctionalInterface
public interface ParserFunction<T, R, E extends Exception> {
        R apply(T t) throws E;
        static <T, R> Function<T, R> wrapper(
                        ParserFunction<T, R, ParseException> f) {
                return t -> {
                        try {
                                return f.apply(t);
                        } catch (ParseException e) {
                                e.printStackTrace();
                                throw new RuntimeException(e);
                        }
                };
        }
}
…
private static List<String> print(List<String> input) {
        return input.stream()
                .map(ParserFunction.wrapper(str -> {
                        SimpleDateFormat format = new SimpleDateFormat(
                                "mm-dd-yyyy");
                        System.out.println(format.parse(str));
                        return str;
```

```
        }})).collect(Collectors.toList());
}
```

Similarly, below we see the way the code may be rewritten when a default methods is used.
Example 1 (Scenario 1):

```
@FunctionalInterface
public interface ParserFunction<T, R> extends Function<T, R> {
        @Override
        default R apply(T t) {
                try {
                        return applyWithException(t);
                } catch (ParseException e) {
                        e.printStackTrace();
                        return null;
                }
        };
        R applyWithException(T t) throws ParseException;
}
…
private static List<String> print(List<String> input) {
        return input.stream().map(
                (ParserFunction<String, String>) (str -> {
                        SimpleDateFormat format = new SimpleDateFormat(
                                "mm-dd-yyyy");
                        System.out.println(format.parse(str));
                        return str;
        })).filter(str -> str != null)
                .collect(Collectors.toList());
}
```

Example 2 (Scenario 2):

```
@FunctionalInterface
public interface ParserFunction<T, R> extends Function<T, R> {
        @Override
        default R apply(T t) {
                try {
                        return applyWithException(t);
                } catch (ParseException e) {
                        throw new RuntimeException(e);
                }
        };
```

```
        R applyWithException(T t) throws ParseException;
}
…
private static List<String> print(List<String> input) {
        return input.stream().map(
                (ParserFunction<String, String>) (str -> {
                        SimpleDateFormat format = new SimpleDateFormat(
                                "mm-dd-yyyy");
                        System.out.println(format.parse(str));
                        return str;
        })).collect(Collectors.toList());
}
```

A Trick

The same trick used to bypass the Java compiler restriction on checked exceptions can be used with streams.

Example 1 (Scenario 2):

```
@FunctionalInterface
public interface ParserFunction<T, R> extends Function<T, R> {
        @Override
        default R apply(T t) {
                try {
                        return applyWithException(t);
                } catch (ParseException e) {
                        throw new RuntimeException(e);
                }
                return null; // Effectively unreachable
        };
        R applyWithException(T t) throws ParseException;
        @SuppressWarnings("unchecked")
        static <T extends Throwable> void
                rethrow (Throwable t) throws T {
                throw (T) t;
        }
}
…
private static List<String> print(List<String> input) {
        return input.stream().map(
                (ParserFunction<String, String>) (str -> {
                        SimpleDateFormat format = new SimpleDateFormat(
                                "mm-dd-yyyy");
```

```
                System.out.println(format.parse(str));
                return str;
        })).filter(str -> str != null)
                .collect(Collectors.toList());
}
```

If there is a checked exception thrown from the processing code, it will act in the same way as in case of the corresponding Lambda expression in a cycle. It will “travel” up the stack to the nearest (if any) corresponding **try-catch** block able to intercept it.

3rd Party Libraries

Both **Vavr** (until April 2017 – **Javaslang,** www.vavr.io) and **Functional Java** (www.functionaljava.org) provide facilities on how to handle exceptions in Streams.

5. Patterns & Anti-Patterns

The difference between a pattern and an anti-pattern is more emotional than practical. When an author wants to emphasize a correct approach to a problem, he says, “DO it like this” and he calls it a pattern. When the author wants to emphasize an incorrect approach to a problem, he says, “do NOT do it like this” and he calls it an anti-pattern.

Every pattern can be turned into an equivalent anti-pattern by negating its inversion. In the same way, every anti-pattern can be turned into a pattern. We will not really distinguish between the two and will rather use the names “pattern” or “anti-pattern” only for emphasis.

The Central Pattern

The central pattern of exception handling in Java is that a perfectly written Java program in a perfect world will never throw an exception in runtime. This is something that will never be achieved, but is something one should strive for.

Exceptions are to be avoided in the same way as one should avoid using a fire extinguisher. One should not do it on a regular basis, but rather, should do it only as a last resort when all other means have failed.

Structure & Design

Reinventing the Wheel

Isaac Newton stated that he was able to achieve a lot by “standing on the shoulders of giants”. He used the achievements of his predecessors to advance in mathematics and physics and, in doing so, is considered to be one of the greatest scientists of all time.

In a similar way, it does not make sense to invent something if it has already been invented before us.

Core Java classes have a large number of exceptions that may be used in different situations. Before creating a custom exception, it is always a good idea to check if there already is an exception that can be (re-)used. If the exceptional situation is generic in nature and not application specific, there may already be an exception class that can be used.

For example, if you detect that an object passed to a method is null, there is absolutely no need to create a class NullUserPassedException (or something similar), because there is already the NullPointerException that can be used.

Similarly, there may already be an exception suitable for the situation in your own or some third-party libraries.

Not Too Generic

Although reusing core Java exceptions in general is a good idea, it is important to note that base exception classes (**java.lang.Throwable, java.lang.Error, java.lang.Exception,** and **java.lang.RuntimeException)** cannot be used for this purpose. They are too generic in nature to describe any particular problem.

Logical Hierarchy

Exceptions should follow the same principles of building class hierarchy as pretty much all other Java classes. The exceptions in the hierarchy should logically reflect the nature of underlying problem (or a set of problems).

Close in functionality exceptions may inherit from a base class, that may be added to the base service method throws clause. This approach is widely used in Java core classes and in popular frameworks. A typical example of such a class is java.io.IOException and its subclasses.

An exception thrown by a method should correspond to the abstraction level of that method.

Adequate Anatomy

Exceptions are used rather as messages that indicate some atomic exceptional situation. They are not objects with complicated structures and/or complicated processing logic. The only two expensive and/or complicated methods are fillInStackTrace() and printStackTrace().

The exception hierarchy adequately reflects the underlying problems and has a separate class for each of them. Most of the time, information about an exception can by placed in its message and there is no need to add fields to the exception that convey more information.

But this is not always the case.

In some cases, an exception may reflect a problem with multiple sub-cases. Even worse, the subcases may change frequently over time making it impossible to have a separate class for each problem. In this case there may be some fields added to the exception class that contain additional information about the exception and its particular subcase. Sometimes the developer may consider using so called “error codes” (or their equivalents) in their exception, rather than using class hierarchy exclusively. Error codes may be also used for further exception processing.

A good example of the exception with multiple subcases is the java.sql.SQLException. Having a separate class for every database interaction problem would be excessive. Therefore, in addition to the standard fields from java.lang.Throwable it defines two additional fields: SQLState and vendorCode. These two fields provide further details of the underlying problem if this information is relevant.

Another case where an exception may have a field is when it is to be processed automatically by some processing engine or, for example, a UI controller processor, that selects the appropriate message to show it to the user.

In summary, the anatomy of exceptions used in a particular case is driven by the nature of the underlying problem; while there are certain guidelines that we describe above, overall, the correct decision regarding it is often more an art than a science.

Enough Details

An exception should have sufficient details regarding the context it was created in. It does no harm to include all parameters that were passed to the method that threw the exception.

It is not about the structure (that should not be overcomplicated as previously stated) of the exception class, but about information provided to the message.

Example 1:

```
public User findUser (long id) throws UserNotFoundException {
    …
    // WRONG!!!
    throw new UserNotFoundException(“User not found”);
    …
}
```

Example 2:

```
public User findUser (long id) throws UserNotFoundException {
    …
    // RIGHT!!!
    throw new UserNotFoundException(
            “User with id = ” + id + “ not found”);
    …
}
```

This has already been mentioned but it does no harm to repeat: if the exception thrown is the result of another exception, the latter should be added as the cause of the former. This information is an important detail and should not be lost.

Usage

Exceptions can be used in a variety of situations, from technical to business. But the situation should

always be exceptional in nature.

Not for Flow Control

As discussed earlier, creating and handling exceptions is expensive.

They also may transfer control across method calls, which in turn makes tracing and debugging a program execution flow less straight forward and more sophisticated.

Therefore, using exceptions is potentially associated with significant resource costs and transfer of control complexity.

Exceptions should not be used as means of simple transfer of control; they should only be used for bona fide exceptional situations.

One of the ways to avoid unnecessary exceptions is by checking the conditions that lead to the exception and if the conditions are met - use of other means of transfer of control.

For example, you call a method that withdraws funds from a users account.

The method:

```
public void withdraw(BigDecimal amount,
    String accountNumber) throws notEnoughFundsException {
…
}
```

The call is:

```
…
try {
    …
    withdraw(amount, accountNumber);
    …
} catch (notEnoughFundsException e) {
    // Inform the user
    …
}
…
```

The code above would be improved if it was changed to something like the following that is neater and more compact:

```
…
Boolean enoughFunds = true;
If(hasFunds(amount, accountNumber)) {
    try {
            …
            withdraw(amount, accountNumber);
            …
        } catch (notEnoughFundsException e) {
            enoughFunds = false;
            …
        }
} else {
    enoughFunds = false;
```

```
    …
}
```

The method hasFunds() should not be a big problem to add, if it is missing. The code for withdraw() definitely performs this kind of check and probably a similar method (or at lest a portion of code that does it) already exists.

The code above may look redundant, but its importance is the idea itself, that the conditions that may lead to an exception should be checked before the operation throws the exception. The exceptional conditions are intercepted and taken care of, using regular means of transfer of control, without using the exception mechanism.

One may say that the code that intercepts the exception with a catch block is still there. Yes, it is, but it is reserved for an exceptional situation, when within fractions of a second between the check for funds and the actual withdrawal someone withdraws the funds. This situation may happen, and it is not forgotten. But it is extremely exceptional. This is an example of what exceptions are for.

Let us illustrate it with an example from real life.

You had a party with a campfire in your backyard. At the end of the party, you want to put the fire out. You do not call 911 to do this. Instead, you take a bucket of water and pour it on the fire. If you fail, calling 911 becomes an option. It is an expensive option, but sometimes you have no choice. Throwing an exception is similar to calling 911. You do not call it just to accomplish what could have been done using less extreme methods.

Not to Correct Bugs

As already mentioned, certain exceptions clearly indicate the presence of programming bugs. They are not intended to be caught.

While it is clear that JVM and environment failures (normally represented by subclasses of java.lang.Error) are not to be intercepted under regular conditions, it is not always clear that programming bugs are not to be caught and fixed by the code itself.

Low level failures normally show up in a form of subclasses of RuntimeException: ClassCastException, NullPointerException, IllegalArgumentException, ArrayIndexOutOfBoundsException, NegativeArraySizeException, etc.

They originate at some upper logical or intermediate processing level, then they go down where they result in some invalid/illegal action.

For example, an algorithm that sorts an array of data has a logical bug, and one day it results in an incorrect index of an array element. An attempt to access it results in ArrayIndexOutOfBoundsException. The only reasonable reaction to the problem is to stop the program and rewrite the algorithm. If the code intercepts such an exception and tries to recover or do something about it, the code is actually trying to fix its own bug. It is much easier to write a code that correctly accesses an array index (or passes a non-null object) than an AI-like code that fixes its own problems. Chances that it does so correctly are much lower than the chances that it simply hides the bug even deeper in the code logic. Sooner or later, it will result in even worse problems and discovering the bug will be far more difficult.

The moral of the story is that clear low-level programming bugs, represented by runtime exceptions should not be intercepted by a catch block with the purpose of recovery.

Handling

Overly Broad

As already discussed, **java.lang.Throwable, java.lang.Error, java.lang.Exception,** and **java.lang.RuntimeException** are four base classes in the Java exceptions hierarchy. Using them directly is strongly discouraged when creating and throwing exceptions, and also while declaring, catching, and processing.

In throws

It is not unusual for a programmer who is tired of adding exceptions that can be thrown from the underlying class to the method signature, to switch to a simplified solution, such as:

```
public void method() throws Exception {
    …
}
```

This approach entirely eliminates the idea of a checked exception. Having java.lang.Exception in a throw clause is roughly equivalent to not enforcing checked exceptions handling at all.

This approach can be even more radical using java.lang.Throwable instead.

In catch

Another example of a bad practice is catching java.lang.Exception.

```
try {
    …
} catch (Exception e) {
    …
}
```

Note that every checked exception type potentially thrown from the underlying calls requires attention may lead to a radical solution. The user here decides to catch them all together, which is a bad idea.

Things can get even worse, when, as in the previous example, java.lang.Throwable is used instead.

Loosing

After an exception is caught, something should be done with it. It may be wrapped and re-thrown up the call stack or it may be processed, but in an attempt to process it, the information about the exception may be lost. There are a couple of scenarios on how this may happen.

Swallowing

This is probably the most common and universally accepted exception anti-pattern. It is known under different synonymous names such as hiding, swallowing, ignoring, suppressing, etc. According to some studies [12], up to 20% of catches in Java code do nothing. An exception is caught, and nothing is done with it. Sometimes it is done intentionally, for example to silence the complier, but it may also be done as a result of some unintentional (though erroneous) actions.

Waving It Away.

We have already acknowledged that the Java compiler may be very annoying about checked exceptions. It simply will not let you compile your code if you do not take care of the checked exceptions a method may throw.

Programmers often rush to finish a project and make it work for what is sometimes called “one true path” [13]. The program is tested on correct data, the exception is never thrown, and everything is assumed to be fine. The issue of what to do with the exception caught is delayed until later, but “later” never comes during the development/testing stage. It *does* come when the code is already deployed to production.

The easiest way to silence the compile is to do something like the following, which is how it is normally done.

```
try {
    …
    throw new SomeException();
    // Or
    // methodThatThrowsSomeException();
    …
} catch (SomeException e) {
}
```

The scenario above is quite common and is often the result of negligence due to lack of time.

While there may be other reasons why people take these types of actions, the only real remedy is proper exception handling. In the worst case scenario at least some TODO should be added to the code with further intent to take care of the intercepted exception.

When is it justified?

You may be surprised, but in some (although rare and exclusive) cases swallowing an exception may be justified. Or at minimum could be not considered a critical programming mistake.

The first case is an impossible catch situation.

For example, due to some specifics of the Java language (which we will discuss later), there may be situations where the compiler will require a corresponding handler for an exception that is never actually thrown

from a method.

```
OutputStream out = new ByteArrayOutputStream();
byte[] bytes = getBytes();
try {
    out.write(bytes);
} catch (IOException e) {
}
```

The ByteArrayOutputStream class in its write() method never actually throws an IOException instance. But this exception is declared in the parent (e.g., OutputStream) class and should be taken care of from the compiler perspective.

It is a paradoxical situation where we have to add a catch block that is never executed.

Although the code in the catch block example above is never executed and leaving the catch block empty cannot be considered a serious programming mistake, we still recommend having a logging record there. The code may change (it is unlikely for ByteArrayOutputStream but may not be unlikely for some other class) then the code inside the catch suddenly becomes executable making matters worse. Finding this fact may become a problem.

Something like this will protect from this possibility:

```
OutputStream out = new ByteArrayOutputStream();
byte[] bytes = getBytes();
try {
    out.write(bytes);
} catch (IOException e) {
    logger.debug("Should never happen", e);
}
```

The second case is failure while closing/cleaning up/finalizing cooperation with some external resources such as database connections, I/O sockets, interrupting threads, etc.

Professional Java programmers know the following code snippet very well.

```
Connection con = null;
PreparedStatement ps = null;
ResultSet rs = null;
try {
    …
} finally {
    try { rs.close();} catch (Exception e) {}
    try { ps.close();} catch (Exception e) {}
    try { con.close();} catch (Exception e) {}
}
```

Or:

```
try {
    socket.close();
} catch (IOException e) {}
```

What we have in this case is some secondary activity (finalization of resources/threads used) that may fail. The resources are normally not reused in the course of the main activity and failure to close them normally does not directly affect the outcome of the main activity.

External resources are often supported by some vendor specific library that has its own logging. Most of the information related to failure may be found there. For this reason, not logging the exception by the caller is not critical.

In summary, in the above cases, failure to resolve issues in the catch block cannot be considered a critical (or even serious) programming mistake. But we still recommend logging this type of exception.

```
Connection con = null;
PreparedStatement ps = null;
ResultSet rs = null;
try {
    …
} finally {
    try { rs.close();} catch (Exception e) {
        logger.debug(e);
    }
    try { ps.close();} catch (Exception e) {
        logger.debug(e);
    }
    try { con.close();} catch (Exception e) {
        logger.debug(e);
    }
}
```

Or:

```
try {
    socket.close();
} catch (IOException e) {
    logger.debug("Failed to close a socket", e);
}
```

Discarding

Next is the case when an exception may be lost while improperly using **catch** and **finally** blocks. As discussed previously, **catch,** and **finally** blocks may throw their own exception or may transfer control.

Tricky logic and much activity in both blocks may cause this to happen. In this case, as already covered in previous chapters, the new exception (or transition of control) takes precedence, and the original exception is discarded.

Example 1:

```
public void doSomething(String[] args) {
    try {
        …
        throw new SomeException(args);
        …
    } catch (SomeException e) {
        Logger.debug(
            (parse(getTemplate(args).getCaption(), e));
        …
```

```
        );
        …
    }
```

As soon as getTemplate(args) returns null, an attempt to call getCaption() on it will throw a NullPointerException. The instance of SomeException will be discarded and the instance of NullPointerException will start its “flight” to the nearest catch block that is able to intercept it.

The situation may be even more complicated.

Example 2:

```
    public void doSomething(String[] args) {
        try {
            …
            throw new SomeException(args);
            …
        } catch (SomeException e) {
            Logger.debug(
                (parse(getTemplate(args).getCaption(), e));
            …
        ) finally {
            Logger.debug("Done with: ", args[1]);
        }
        …
    }
```

The args may consist of only one element, and in this case the instance of ArrayIndexOutOfBoundsException will take precedence over all other exceptions thrown before it, both SomeException and NullPointerException.

The moral of this story is that one should avoid complicated logic in **catch** and **finally** blocks. It is best to keep these blocks simple.

If processing logic that may throw its own exception in **catch** and/or **finally** blocks is unavoidable, the new exception can be considered to be added as a suppressed to the original exception.

Layer-crossing

The universal approach to software (including Java) development implies that, according to the principal of separation of concerns, the code consists of logical layers (also called tiers). Each layer uses its own logic and operates on its own domain objects.

Transformation

The common pattern here is that when an exception crosses the border between two logical layers, it should be transformed to a new exception that corresponds to the new layer semantics.

```
public Account loadAccount(String accountNumber)
        throws FinderException {
    try {
        …
    } catch (SQLException e) {
        …
```

```
            throw new FinderException(
                "Failed to load account for number : "
                + accountNumber, e);
        }
    }
```

The layer that loads account information takes an account number and loads the account information from an underlying storage. Today it may be some relational database (as in the example), and tomorrow it may be some NoSQL storage. But the caller of the method is not meant to know all of these details. It should be transparent to them. Therefore, the exception they get should be one that denotes the problem in the service terms – "account not found" or something similar, without disclosing the details of why and how.

Simply transforming the exception and throwing a new one is not enough. It should be done in a proper way. The next section describes how.

Before we move on to the next section, we should emphasize an important point: the rule described in this section should be applied only to exceptions that, by their nature, belong to the corresponding logical layer. For example, java.lang.OutOfMemoryError does not belong to any specific logical layer, it belongs to the whole JMV logical space. One should not try to catch it and convert into something more specific. The same applies to java.lang.NullPointerException most of the time. It typically indicates a programming bug, is related to the application (or its component) as a whole and requires developers' attention.

Missing Cause

Another pattern associated with this issue is that when we catch an exception, transform it to a new one and pass it up to the caller, we should make sure that no important technical information is lost. The most important information in this context is the information about the original exception. This exception is called the cause. The java.lang.Throwable class has three constructors that let you attach the cause to an exception. The process of transformation associated with attaching the cause is called **wrapping**.

In the example above, we called the constructor that accepts another exception as the cause.

```
throw new FinderException(
                "Failed to load account for number : "
                + accountNumber, e); // CORRECT
```

Calling a constructor that does not accept the cause would be an erroneous approach.

```
throw new FinderException(
                "Failed to load account for number : "
                + accountNumber); // INCORRECT
```

There may be an objection on the grounds that the information about the cause is still available to the caller (through the getCause() method) and the logical layers are not fully separated. The reply to this objection is that the purpose of a good design is not to absolutely, whatever the cost, separate the layers in technical terms, but to separate layers logically and avoid unnecessary overexposure of the implementation details.

At the point where the exception is finally processed and logged, its stack trace is printed. By including the information about the cause, the information will be visible for technical analysis.

Logging

The logging of Java code is a large subject unto itself and therefore we cannot cover all aspects associated with it in this publication. However, some mistakes associated with the usage of logging for exceptions are quite common and therefore should be addressed.

Log or Re-Throw?

There is no need to log and/or print stack trace every time an exception is caught and/or re-thrown. There is also no need to catch an exception simply for the purpose of logging it.

This is wrong:

```
public methodA() throws MyException {
```

```
        try {
            methodB();
        } catch (MyException e) {
            logger.debug("MyException caught", e);
            throw e;
        }
}
public methodB() throws MyException {
        …
        throw new MyException();
        …
}
```

Printing stack trace too often does not add any information, is confusing, and makes log analysis more difficult. The stack trace should be printed only once and in the appropriate place.

If you are not planning to do anything useful with the exception, (e.g., convert it to another exception), show a message to the UI user, or process it by notifying another thread. Most of the time, you should just leave it.

Therefore, the code above should look like:

```
public methodA() throws MyException {
        …
        methodB();
        …
}
public methodB() throws MyException {
        …
        throw new MyException();
        …
}
```

The approach we describe above is generally correct, but it should be done with discretion, depending on how and where the exception is to be processed.

First, there may be some kind of automatic exception processing up the stack (for example in a servlet) that will take care of showing the exception page to the end user, but the automatic processing will not log the exception. Or, for example, the method invocation may be a result of a remote call, or even multithreading invocation on the same JVM in which case there will be no guarantee that the caller/invoker will receive the exception. Therefore, the exception should be logged before it is thrown and leaves the current JVM (or even current JVM thread if the thread is isolated).

Second, even in the same thread, your component may be isolated from the caller code, which may be written by a complete stranger. There is no guarantee that the calling code will log the exception your code throws. As a non-perfect solution you may try to log the exception somewhere closer to the point where the component is called by the caller. A good example of this approach can found in core Java classes, in java.sql package.

Up the Stack

The most suitable place for logging is normally around the place where the activity that failed was initiated, which most of the time is around the same place where the exception thrown is handled, typically somewhere up the stack.

First, the knowledge of the context is maximal and maximum available information can be logged.

Second, logging down the stack is problematic, because deeply nested methods do not know in advance in which context they will be used, and thus which logger should be used.

printStackTrace()

The printStackTrace() method is a highly informative way to print the stack trace of an exception with all of its causes and suppressed inwards formatted. New Java programmers often worry about how they can log the stack trace and try to utilize the familiar method for the purpose.

The pattern is that they should use the corresponding logging framework instead. All popular Java logging frameworks know very well how to handle java.lang.Throwable and what to do, when it is passed as an argument to the corresponding logging method.

```
logger.debug("An exception occurred.", e);
```

The sample code above will print the stack trace to the logging console.

For example, in **ch.qos.logback.classic.Logger**, which is the logger class for Logback framework, provides the following method.

```
public void debug(String msg, Throwable t)
```

The loggers do not necessarily use printStackTrace() on the background, to print the stack trace, but they follow the same logic as the printStackTrace() method, meanwhile optimizing the output overhead.

Cleaning Up

Cleaning up and/or releasing resources at the end of a method that uses them is a good programming practice. The problem is that there may be an exception thrown before the clean-up code is even reached. Therefore, the clean-up code is simply skipped and never executed. Using finally block is the way to avoid this problem.

Incorrect:

```
…
// main logic (start)
…
// main logic (end)
// clean up (start)
…
// clean up (end)
```

Correct:

```
try {
    // main logic (start)
    …
    // main logic (end)
} finally {
    // clean up (start)
    …
    // clean up (end)
}
```

If there already is a catch block that surrounds the main logic, it should be accomplished with a corresponding finally block, similar to the one above.

Needless to say, one should certainly make sure that no exception is lost if the finally block throws its own exception.

Miscellaneous

Document them

As previously stated, checked exceptions are self-documented. They are always in the throws clause of the corresponding method and simply running javadoc utility adds them to the API description.

With unchecked exceptions the situation is different. Unless they are added to the throws clause of a method (or tagged with @throws) they will not be visible in the documentation or in your IDE hints.

It is evident that one cannot add all unchecked exceptions that may be possibly thrown by a method to its signature. Almost any method can throw NullPointerException, or (from deeply nested calls) ArrayIndexOutOfBoundsException, or ArithmeticException.

Therefore, which unchecked exceptions should be added to a method signature?

The answer is only those that are explicitly thrown in the method and/or thrown from the underlying methods should be added plus any the caller should be reasonably expected to recover from. Interestingly, this sounds like the definition of a checked exception. It is true that many modern frameworks use unchecked exception in the same way the official documents by Sun/Oracle recommend using checked exceptions. Later we will discuss why they do this.

For now, if your code uses unchecked exceptions to transmit recoverable logic, they should be added to the method throws clause. The compiler will not force you to handle them, but at least they will be visible to the caller in the method signature, generated API documentation and IDE hints.

6. Two Major Views

As discussed in previous chapters, there are two major types of exceptions: checked and unchecked. The Java language itself offers a mechanism that differentiates between the two.

Official documentation by Oracle (and previously Sun) on the Java language provides a vague definition of the difference between the two.

The real question with checked exceptions is not how to technically use them, but rather, how to decide when an exception should be declared checked and when it should not.

Overview

The Java language treats checked exceptions differently than unchecked ones. The technical aspects of this differentiation has been described in previous chapters.

But what was the motivation behind introducing these concepts to the language? How is a programmer supposed to decide which exception to deem checked and which unchecked? What are the guidelines?

Although Oracle (and previously Sun) engineers give some guidelines in the official documents, there is a strong voice in programming society to not treat checked exceptions differently than unchecked. Which essentially means to treat all exceptions equally. The natural conclusion from this preposition is that the concept of checked exceptions was a mistake, made by the Java language architects.

In this chapter we will go through the most common arguments for and against checked exceptions.

Pro Checked Exceptions

The idea that in Java checked exceptions should be classified and treated in a specific way is based on two major postulates, and there are corresponding arguments supporting these postulates, as outlined below.

Two Categories

The first postulate is that all exceptions, by their nature, can be divided into two major categories (which does not preclude exceptions from being further divided into sub-categories).

The first category includes exceptions that irrecoverably break the program execution flow (such exceptions are called **errors**) or indicate programming mistakes (such exceptions are called **runtime exception**) that may lead to such breaks. An example would be that there is not enough memory for the JVM to run. Or a thread has already exhausted the call stack. Or a method tries to access a field of a null object because of a software bug. All that can be done in most of these situations is the program execution must be interrupted and human intervention is required. A significant number of such exceptions is detected and raised by the JVM. Names used for this type of exceptions include runtime exceptions and system exceptions. As mentioned, in their official guidelines Sun/Oracle states that the first type of exceptions should be used for **faults**.

The second category includes exceptions that do not naturally break the program, although they may alter its execution logic. As an example, say your application tried to withdraw funds from an account associated with a subscriber's account. The operation failed with a **NotEnoughFundsException** thrown. Nothing dramatic has happened, and the system may try to use the second (reserved) account registered by the user, which may work perfectly. An exception thrown, but it was expected, and the program knew how react to it. It is often said that the second category represents "recoverable" conditions. However, this term does not always correctly describe what is going on. Often there is nothing to recover from. If the user already exists and the corresponding exception **UserAlreadyExistsException** is thrown, what can be recovered in this situation? A better term would be "non-irrecoverable conditions", in the contrast to "irrecoverable conditions". But, since "recoverable" is an already established term, we will use it through the rest of this book. Names used for this second type of exception include application exception and business exception. Earlier, we mentioned that the official guidelines by Sun/Oracle state that the second type of exceptions should be used for **contingencies**.

Often the exceptions from the first category are called unchecked and the exceptions from the second category are called checked. But these names do not reflect the nature of the exceptions. These names, instead, reflect the fact that the Java language treats the two categories differently. The next paragraph explains this in more detail.

Enforced Handling

Following the principal of separation of concerns, the two categories of exceptions are good candidates to be treated differently. Although the principal states that different concerns should be treated differently, it does not state precisely how it should be done.

Static Control

The second postulate is that static control is better and should be enforced as much as is reasonably possible. Thus, applying it to exception handling, it makes sense to impose it on recoverable exceptions.

The exception that indicates recoverable conditions is always created by the user code and can be much easier indicated at the compile time, the user can be forced to handle them, or pass up the stack. Strict static control will force users to not forget about the possibility that such exceptions can be thrown. Therefore, the user will be forced to take preventive measures.

The checked exceptions declared in a method's throws clause are becoming part of the method signature.

Positive Side Effect

The positive side effect (that is not denied even by active opponents of the model) of static control is that checked exceptions are self documented.

In order to generate a list of checked exceptions thrown, all you need to do is to run the **javadoc** utility.

Unchecked exceptions lack this helpful facility. The javadoc utility will add them to the generated documentation, but only if they are declared in a method's throws clause. But, adding them to the throws clause is optional.

Summary

The theory of checked exception model as it is implemented in Java consist of the two postulates described above plus arguments that support them.

The opponents of the model offer arguments against it, trying to disprove it. The proponents of the checked exceptions model, in turn, try to defend their position and present their counter argument.

In legal terms, in the case against checked exceptions, the pro-checked exceptions camp acts as defendant, and their opponents act as plaintiff. This means that the plaintiff presents their evidence, and the defendant presents their defense.

In the next chapter we will go throw the most common arguments against the checked exceptions model in Java and the corresponding counter arguments.

Contra Checked Exceptions

According to Bruce Eckel, the idea of enforced handling of this type of exceptions is based on the assumption that strict static control is always better [14]. He calls this assumption unchallenged and there appears to be no evidence that he is wrong. The whole concept of Java is that it checks and enforces statically as much as it can. There are some exemptions to this rule, but overall, it is true.

A significant number of arguments against checked exceptions correlates with the disadvantages of static type checking. In addition, there are some other arguments of a different nature.

The important point to understand is that different experts often have different opinions as to why checked exceptions are bad. The sets of arguments they use may be different and may vary significantly. The sets normally do not match completely, but they may overlap. There is no universal institution that broadcasts the opinion of the contra checked exceptions community and there is no united view.

In the following section we outline the major arguments and counterarguments against the checked exceptions concept.

Non-Existing Arguments

There are several statements that those who are opponents of checked exceptions do not (contrary to what some of their opponents think) deny. Both sides agree on these statements. We think it is important to emphasize them.

Do Not Use Them

Opponents of the checked exceptions model (with few exceptions) do not deny the fact that exceptions should be used. Exceptions, as previously highlighted, are a very good OOP abstraction that represents an abnormal situation.

An alternative to using exceptions in some technical and business situations could be the use of return

values. The problem is this would take us back to C, Fortran and Pascal programing styles and times. There is nothing wrong with these languages, and they each have their own niches. But in the case of Java, using this style would mean completely eliminating all of the advantages of object-oriented programming with its encapsulation and compact, human-like syntax.

Therefore, nobody (with the exception of some radicals) denies the fact that exceptions can and should be used for both technical and business problems.

Do Not Recover

Some people believe that opponents of Java's checked exception model deny the fact that exceptions should be used for anything other than termination of the execution flow and exiting the running process. This is not true. Most of the opponents of the checked exceptions model do not deny the fact that in many situations the program can intercept an exception and try to recover (or use any other appropriate word) then continue its regular flow. They do not deny the fact that some exceptions are recoverable (do not signify irreparable damage to the program environment or execution flow) by nature.

What the opponents of checked exceptions do believe is that although exceptions can be both recoverable or not, this fact should not be engraved into the class semantics and the users should not be forced to treat exceptions in a specific way, predefined at the compilation time.

Ambiguity

Oracle documentation provides the following guidelines on how to define which exception should be declared checked and which unchecked.

> *"If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception."* [15]

If a program execution threw an **OutOfMemoryError** exception, there is a little that can be done by the program itself. The most natural approach would be logging the problem, terminating the execution, and giving the problem over to a human being. On the other hand, if there is a **NotEnoughFundsException**, terminating the program would be excessive, and instead the program should address the execution to the branch of code that will politely inform the user of the situation and offer him another method of payment or delay the operation until there are sufficient funds.

The two examples above demonstrate that some exceptions may be very clearly classified as either recoverable or not. But this is not always the case. Sometimes it cannot be done easily.

The guidelines by Oracle shown above explain that in case of reasonably recoverable exceptional conditions the exception should be made checked, but it does not explain the meaning of the words "reasonably recover" in this context. Therefore, the problem is addressed to another level, namely the definition (or understanding) of what it means "to reasonably recover". Or alternatively, its cognate "reasonably recoverable". The emphasis in some cases is on the word "reasonably", but most of the time when Oracle/Sun documentation/articles use the word "recoverable" they mean "reasonably recoverable". We will follow this pattern as well.

Let us take a look at some core Java exceptions. This might give us more insight as to how Java architects and engineers intended these words.

The first group of exceptions are those that inherit from java.lang.Error. These are also called errors, and normally represent serious JVM/environment problems that cannot be fixed without termination, fix, and restart. Examples (all from java.lang package) include LinkageError, ErrorOutOfMemory, NoClassDefFoundError, etc. Although most errors represent JVM/environment problems, they may also represent programming bugs and can be fixed by changes to the code and a consequent rerun of the program. An example would be when an incorrectly written algorithm leads to an infinite recursion, which in turn leads to StackOverflowError thrown. Errors are universally considered unrecoverable in nature.

Let us take a look at the second group of exceptions. Those that represent programming bugs. They inherit from java.lang.RuntimeException.

A good example is java.lang.NullPointerException, let's say your program called a method and passed a null object to it. The method expectedly throws an NullPointerException.

Can you catch it? Yes. Can you recover from it? Technically, yes. The exception can be intercepted and the program may attempt to pass a non-null value. But will this recovery be reasonable? No. At least, this is how Sun/Oracle engineers interpret their own guidelines. In addition, we are unaware of any Java expert that would recommend this. The logic is obvious: it is more reasonable to write a clean code than code that "on the fly" fixes its own lack of cleanliness. Put simply, it is easier to write clean code, than self-cleaning code.

The same stands true for a whole group of similar core Java exceptions (from java.lang). These include: ArithmeticException, ClassCastException, ArrayIndexOutOfBoundsException, etc. These classes represent

straightforward programming bugs and their status of being unrecoverable normally is not discussed.

In summary, we have two prominent groups of what is almost universally considered unrecoverable by Sun/Oracle, Java experts and the larger Java community: JVM/environment errors and programming bugs.

Unfortunately, this is not the end of the story.

Let us go to the next example.

java.io.IOException: your program connects to a server locate overseas. The server is down, and you get an IOException.

Can you catch it? Yes. Can you recover from it? Yes. You can definitely try to reconnect several more times or (if your program has a user interface) you can inform the user that your are experiencing some connection issues and ask him to try again later. This appears to be the logic that Sun/Oracle engineers followed and declared the IOException as a checked exception (direct subclass of java.lang.Exception).

But what if you cannot connect to the server because the utility class you use incorrectly concatenates the parameters for the URL? It is a programming bug, and you may retry as many times as you want but it will never help.

Shall IOException be declared as an unchecked exception in this case?

Nature & Context

This is where we come to the evident conclusion that some exceptions, by their nature, may be recoverable in some context and non-recoverable in another. Unfortunately, this is a source of confusion.

To illustrate this, let us provide more examples.

Say that you would like to use core Java SQL facilities to read a set of records from a table. If you make a typo in the table name, you will get an instance of **java.sql.SQLException**, which is a checked exception.

If for the same purpose you would like to use Spring JDBC template classes, you will get an instance of **org.springframework.dao.DataAccessException** which is an unchecked exception.

As you can see, in identical situations, Java experts from different teams may have a different understanding of what is recoverable and what is not.

Some posit that Spring architects were against checked exceptions and intentionally did not use them. But Rod Johnson (the leading Spring architect) explicitly confirmed that he does not support the idea of eliminating them. So, it appears that this is his true understanding of how exceptions should be declared.

Another example: SAX and DOM are two types of XML parsers available in Java SE.

The base exception class used for DOM parsing **org.w3c.dom.DOMException** extends java.lang.RuntimeException and thus is declared as a runtime exception. Whereas the base exception class used for SAX parsing **org.xml.sax.SAXException** extends java.lang.Exception and is an application exception.

One more example: If you want to parse a string into a numeric value, there are several similar methods in core Java that you can use.

java.lang.Integer method parseInt():

Integer.parseInt(string);

Or java.text.NumberFormat, parse():

NumberFormat.getInstance().parse(string);

The difference is that the parseInt() throws **java.lang.NumberFormatException** which is an unchecked exception. The parse() method throws **java.text.ParseException** exception, which is a checked exception.

In summary, this means that even for Java language architects and engineers it was a challenge to decide in which of the two categories to include an exception. And as a result, some exceptions that are similar in nature fell into both.

The fact that a program can reasonably recover from throwing a certain exception or not, generally speaking, may heavily depend on the context in which the exception is used and often does not have a universal answer even between top level experts.

If we decide that the conditions that correspond to the exception are always recoverable, and the exception should be declared as a checked one, there still may be a case, when this kind of exception is non recoverable. We may decide that the exception should be declared as unchecked, but in some context it may be recoverable. It depends on what the calling code is doing. We are forced to somehow declare the exception, and whatever we select it is, a no-win situation.

It is clear that there exists a significant gray area, where it is impossible to decide upfront whether the consumers of a given service will be able to “reasonably recover” from a certain exception.

The conclusion and the proposed solution are as follows: the border between the two types of exceptions is not always clear and thus Java’s language to avoid any issues should not really differentiate between them. If the language does not differentiate between the two types, and there is no way to force strict handling of unrecoverable exceptions, recoverable exceptions should be treated as unrecoverable, i.e., without enforced

handling.

Overexposure & Phantoms

This phenomenon is described in different forms and words, but its nature is the same, so we use what we think is the most descriptive definition for this phenomenon.

Before we discuss the details, let us recall some basic principles of OOP and how they are implemented in Java.

```
public class MyParentClass {
    void method() {
    }
}
public class MyChildClass extends MyParentClass {
    int newField = 0;
    void newMethod() {
    }
}
```

In the example above MyParentClass represents a broader set of objects than the MyChidClass. The definition of a child class should narrow the definition of its parent class. Adding a field and/or a method in a child class narrows the set of objects that fall into the class definition compared to the parent class.

In case of a checked exception this logic works in a different way.

```
public class MyParentClass {
    void method() {
    }
}
public class MyChildClass extends MyParentClass {
    void method() throws Exception {
    }
}
```

In the case above you will get a compilation message, something similar to the following:

```
MyChildClass.java:3: error: method() in MyChildClass cannot override method() in MyParentClass
    void method() throws Exception {
         ^
  overridden method does not throw Exception
1 error
```

The reason for such behavior is that by adding an exception to a method signature, it does not narrow, but instead broadens, the definition of the method. This is why it is not allowed in subclasses. Adding a checked exception is similar to adding another return type, which means the subclass is less restrictive. This, in turn, violates the Liskov Substitution Principle. In simple words, adding a checked exception to a method signature increases the number of implementations that conform to it and thus it broadens the definition given in the parent class.

If we have a reverse situation – a checked exception is removed from the definition of a method in a subclass — we are narrowing the definition of the method and the complier is absolutely fine.

```
public class MyParentClass {
    void method() throws Exception {
```

```
        }
    }
    public class MyChildClass extends MyParentClass {
        void method() {
        }
    }
```

The code above will compile without any errors.

In summary, while overriding a method adding an exception (or using an exception of a broader type) is not allowed in subclasses.

This fact leads to some counterintuitive results.

Let's say we have a parent class with a method and a couple of subclasses and want to do something like the following:

```
public class MyService {
    void init() {
    }
}
public class MyWebException extends Exception {
}
public class MyWebService extends MyService {
    void init() throws MyWebException {
    }
}
public class MyDBException extends Exception {
}
public class MyDBService extends MyService {
    void init() throws MyDBException {
    }
}
public class MyInMemoryService extends MyService {
    void init() {
    }
}
```

The classes above will not compile, because of reasons we have already discussed, namely that adding a checked exception to an overridden in a subclass method is not allowed.

Let us refactor the code and move all exceptions to the parent class:

```
public class MyService {
    void init() throws MyWebException, MyDBException {
    }
}
public class MyWebException extends Exception {
}
public class MyWebService extends MyService {
```

```
        void init() throws MyWebException {
        }
}
public class MyDBException extends Exception {
}
public class MyDBService extends MyService {
        void init() throws MyDBException {
        }
}
public class MyInMemoryService extends MyService {
        void init() {
        }
```

The code above will compile, but there are several problems with it. If you want to use any of the implementations, you will have to take of an exception that is never (even potentially) thrown.

```
MyService service = getMyWebSevice();

service.init();
```

In the example, above the compiler will request the handling of not only the MyWebException, which may be actually thrown, but also the MyDBException. The same is true if you want to use the database implementation. But in this case, one will have to handle MyDBException which is never thrown.

The situation worsens if one decides to use the in-memory service.

```
MyService service = getMyInMemoryService();

service.init();
```

In this situation, the invocation of init() method according to the compiler may throw both MyWebException and MyDBException exceptions, but never actually does it.

We have a problem: exceptions are exposed to the methods that do not use them, thus disclosing implementation details; plus, in some situations we are required to handle phantom exceptions (i.e., exceptions that are never actually thrown).

Some refactoring can be done in order to improve the situation.

```
public class MyServiceException extends Exception {
}
public class MyService {
        void init() throws MyServiceException {
        }
}
public class MyWebService extends MyService {
        void init() throws MyServiceException {
        }
}
public class MyDBService extends MyService {
        void init() throws MyServiceException {
        }
```

```
}
public class MyInMemoryService extends MyService {
    void init() {
    }
}
```

Now, instead of the two exceptions — MyWebException and MyDBException — we have one umbrella exception MyServiceException. It helps us partially solve the problems of overexposure. But it will not help us in solving the problem of having a phantom exception.

```
MyService service = getMyInMemoryService();
service.init();
```

The compiler will still demand that we handle the MyServiceException while invoking init() method. Therefore, the user will know that some implementations may actually throw it, but the knowledge will be less precise compared to the first case, when we used two exceptions. In addition, the user will still be forced to handle an exception that is never actually thrown.

As an alternative to the above refactoring, we may introduce and make the inherit from it, changing the code to the following:

```
public class MyServiceException extends Exception {
}
public class MyService {
    void init() throws MyServiceException {
    }
}
public class MyWebException extends MyServiceException {
}
public class MyWebService extends MyService {
    void init() throws MyWebException {
    }
}
public class MyDBException extends MyServiceException {
}
public class MyDBService extends MyService {
    void init() throws MyDBException {
    }
}
public class MyInMemoryService extends MyService {
    void init() {
    }
}
```

But this situation will not be significantly different to the previous one. The only difference will be that the implementations now will throw a more precise exception instead of throwing a more generic one.

Let us take a look at some real-life examples.

As it was pointed out by many experts and Java programmers, one of the best examples to demonstrate this phenomenon is in the java.io package.

The class **java.io.OutputStream** has a method:

public void write(byte[] bytes) throws IOException

The reason why the IOException is added to the method is that some of the OutputStream subclasses may perform input/output operations that may be disrupted and throw some of the IOException exceptions.

The class **java.io.ByteArrayOutputStream** implements **java.io.OutputStream** and thus must have the same signature for the above method. But the implementation never actually throws the IOException. This means that anyone who uses the ByteArrayInputStream has to deal with this issue.

Example 1:

```
OutputStream toStream(byte[] bytes) {
    OutputStream out = new ByteArrayOutputStream();
    out.write(bytes);
    return out;
}
```

For obvious reasons, this code will not compile and will give something like:

```
error: unreported exception IOException; must be caught or declared to be thrown
            out.write(bytes);
                      ^
1 error
```

The IOException never thrown in the ByteArrayOutputStream class write() method must still be handled. (i.e., added to throws clause of the toStream() method or surrounded by a corresponding try-catch block).

In summary, the proponents of the checked exceptions model offered a way (through using an umbrella exception) to partially solve the problem, but there is no way to completely solve the problem of phantom exceptions. This fact definitely speaks in favor of the contra checked exceptions camp.

Cluttered Code

An exception is normally thrown at some point and then consumed somewhere up the call stack. If it is a checked exception, all the methods and objects in the middle are aware of the exception and should explicitly confirm that they pass it up the stack. Therefore, every code between the point where the exception is thrown to the point where it is processed should know about the exception. It creates undesired tight coupling that has several aspects.

Often those aspects are described as separate arguments, again checked exceptions, but because they share the same cause we place them under one umbrella term.

Excessive Dependencies

As we wrote above, every method's call from the point where the exception is thrown to the point where it is supposed to be processed should know about the exception class. This means that every class involved should explicitly import the corresponding exception class and if it is in a separate jar file, all necessary dependencies should be taken care of.

As is well known, multiple imports introduce additional challenges: conflicting, circular, diamond and other unwanted dependencies become more possible. It is not always the case that these situations occur, but the probability increases.

Refactoring Challenges

Any significant change in the checked exceptions (used in a module) structure may require changes to be introduced in all methods that declare the exception in their throws clause.

In other words, using checked exceptions makes refactoring more difficult.

Scalability Issue

This concern was expressed by Anders Hejlsberg [16] and some other lesser-known IT experts.

The idea is that a method that is close to the top of the stack call in its throws clause will aggregate multiple checked exceptions from the underlying subsystems. This is because the calls will, in turn, call other methods in other subsystems. So collectively, the exceptions will show up in the described method.

The objection to this, made by Bill Venners, is that by following just a few healthy design patterns one may

completely eliminate this problem. And most developers do follow these healthy design patterns. First, when an exception crosses the border of its subsystem, it should be converted into an exception that makes sense in the context of the new subsystem. Second, the exceptions (as any other Java objects) may be organized into a hierarchy, so the throws clause may contain only the basic class.

Awkwardness

Handling checked exceptions is often associated with awkwardness, and this is especially true in relation to some Java language features that were introduced later (e.g., Lambdas and Streams) . It is not the same as bad practices — the programmer may follow all recommendations and good practices — but the code can still be cumbersome.

This set of arguments is subjective and definitely not the strongest, but still may add some strength to the opponents of the checked exception model.

A “Trapped Exception”

The problem we describe may affect more than one design pattern and is often mentioned in such context. But this it is not limited to classic design patterns; it may as well affect the code that does not follow any classic design pattern. Its core issue is that directly or indirectly there is a predefined method in the middle of the supposed call chain that does not declare a specific checked exception in its throws clause and thus an exception of that specific type cannot pass through the method up the call stack.

It may happen, for example, while using some third-party library or code the user does not have control over and is not able to change.

Here is an example to help explain it.

You want to parse an XML file using Java built-in SAX parser. For every attribute with (case insensitive) name “URL” the value should be a reachable internet address. As soon as this condition is not met, the file should be discarded.

Example 1:

```
import org.xml.sax.SAXException;

import javax.xml.parsers.ParserConfigurationException;

import javax.xml.parsers.SAXParser;

import javax.xml.parsers.SAXParserFactory;

import java.io.IOException;


public class XMLParser {

    private static final String FILENAME = ".\\file.xml";

    public static void main(String[] args) {

        SAXParserFactory factory = SAXParserFactory
                .newInstance();

        try {

            SAXParser saxParser = factory.newSAXParser();

            MyXMLParserHandler handler = new MyXMLParserHandler();

            saxParser.parse(FILENAME, handler);

        } catch (ParserConfigurationException | SAXException
                | IOException e) {

            e.printStackTrace();

        }

    }

}

…
```

```
import java.io.IOException;
import java.net.URL;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;

public class MyXMLParserHandler extends DefaultHandler {

    @Override
    public void startElement(String uri, String localName,
            String qName, Attributes attributes) {
        if (attributes != null
                && attributes.getLength() > 0) {
            for (int i = 0; i < attributes
                    .getLength(); i++) {
                if ("url".equalsIgnoreCase(
                        attributes.getQName(i))) {
                    new URL(attributes.getValue(i))
                            .openStream().close();
                }
            }
        }
    }
}
```

The method parse() in javax.xml.parsers.SAXParser throws IOException, in a similar way methods openStream() and close() respectively in java.net.URL and InputStream throw the same IOException. It is therefore natural to assume that we should let this exception bubble up to the main() method in our XMLParser file.

But we cannot. The method startElement() does not declare IOException in its throws clause. We cannot add it, because the method is inherited from the parent class org.xml.sax.helpers.DefaultHandler where it does not have the exception in its throws clause. And as you may recall, a checked exception cannot be added to a method in subclasses.

Therefore, the IOException is literally trapped inside the startElement() method.

As stated previously, more than one design pattern may be distracted by the fact that a checked exception cannot easily propagate through the code. In the case of the SAX parser, it is the Observer design pattern. We should notice that normally a notification method from the Observer should not throw any exception, but in this case, it seems perfectly justifiable.

It can also be the Visitor, Strategy, Template Method, or any other pattern, where you pass in “something to do”.

Functional Programming

We already mentioned the awkwardness associated with using checked exceptions in functional programming. It is similar to that of “trapped exceptions” outlined in the previous section.

However, because functional programming is a special case of a “syntactic glue” and because it is especially affected by the problem, we mention it as a separate problem.

The fact that it is a non-trivial task to transparently convey a checked exception through the layers of calls in functional programming is a pretty strong argument against checked exceptions.

The opponents object that having checked exceptions is an old feature of Java, whereas functional programming is a new one. Therefore, maybe something should be (or should have been) done with the

functional programming syntax and/or the way it is implemented in Java rather than with checked exceptions.

Bad Practices

This set of arguments states that there are some bad practices associated with the handling of checked exceptions; they probably should not exist, but *de facto* they do.

Many bad practices are driven by the desire to bypass enforced checked exceptions handling and suppress an annoying compiler. Some of them have already been mentioned while describing the list of patterns and anti-patterns. Eliminating checked exceptions in the language or not using them would help eliminate some of these bad practices.

The reasonable counter argument is that there are always bad practices and not all of them are attributed exclusively to checked exceptions. For this reason, eliminating checked exceptions will not solve all problems. It appears that following good practices is the only real antidote.

Nevertheless, some practices are especially attributed to checked exceptions.

Swallowing

We already described swallowing in the chapter on patterns and anti-patterns. This is probably the most prominent problem associated with the desire to silence the compiler on the subject of checked exceptions.

Programmers swallow not only checked exceptions but unchecked as well. They swallow checked exceptions to silence the compiler, and unchecked ones to silence the JVM in runtime.

Too Broad throws/catch

Using overly broad types (java.lang.Throwable and java.lang.Exception) in throws and catch is another example of a bad practice with the goal of silencing the compiler.

It is associated mainly with checked exceptions.

Final Argument

As we can see, there are strong arguments for and against checked exceptions, although it appears that most experts and programmers have already decided to which camp they belong. There is no way to mathematically prove if checked exceptions should be used or not used in the Java programming language.

The pro checked exceptions camp appeals to more static control (plus self-documentation) advantages, and the contra checked exceptions camp appeals to the fact that, collectively, disadvantages associated with checked exceptions outweigh the advantages and the checked exceptions model was a failed experiment.

Who is right?

In the author’s opinion, the fact that in the 25 years following the release of the Java language, no other language has decided to use checked exceptions, is the final argument that the checked exceptions model was a mistake.

Even the experts that theoretically supported the idea of using checked exceptions in some cases, in practice, avoid using them in their own frameworks. And moreover, they do not use them in all of the other languages they have developed, including JVM based languages – Groovy, Kotlin, Scala and Clojure.

Discussion

As discussed, the Java language introduced some new features, and checked exceptions was one of the most prominent. It took some time (approximately 5 years) for the programming society to begin actively discussing if it was a useful feature.

Experts

Anders Hejlsberg

Anders Hejlsberg is a prominent software expert of Danish origin. Worked at Borland (1989 - 1996) and then at Microsoft (since 1996).

At Microsoft, since 2000 he was the lead architect of the C# development team.

Anders is a strong opponent [16] of the idea of checked exceptions in programming languages.

In addition to the C# design that does not support checked exception and thus implicitly reflects Andreas view on the problem. He gave a couple of interviews where he explicitly confirmed his views.

Joshua Bloch

Joshua J. Bloch is a software and technology author. Worked at Transarc Corporation (1989 - 1996), Sun Microsystems (1996 - 2004), Google (2004 - 2012), Carnegie Melon University (since 2016).

He is one of the Java designers, designed some features that were introduced into the language. Between them - Java Collections API, the java.math package and assertions. His book Effective Java (2001) is one of the most popular Java language guides.

The first edition (2001) of the book and the third edition (2018) have the same recommendation on exception use. It looks like during the 17 years the authors view on the subject did not change much.

Though in his book Joshua Bloch warns against excessive usage of checked exceptions, his views on exceptions are pretty much in line with Sun/Oracle orthodoxy.

James Gosling

James A. Gosling is a Canadian IT expert. Worked at Sun Microsystems (1984 - 2010), Google (2011), Liquid Robotics (2011 - 2017) and cooperated with/consulted Lightbend, Jelastic, Eucalyptus and DIRTT Environmental Solutions.

He was one of the lead Java designers. His influence on the language was so huge, that some people even call him the father of Java.

James is a strong advocate of checked exception model as it is implemented in Java.

His views are reflected in the Java language design, official documentation by Sun/Oracle, books, and interviews.

Rod Johnson

Roderick ("Rod") Johnson is an Australian IT expert. Founded and worked at SpringSource (2002 - 2009), worked at VMware (2009 – 2011), Neo4j (since 2011), Typesafe/Lightbend (2012 - 2016), Hazelcast (since 2013), founded and worked at Atomist (since 2016). He is a well-known expert on Java, Scala, and TypeScript. Considered to be the father of Spring Framework.

First, as a J2EE expert and then as an architect of Spring Framework he opposed the idea of excessive use of checked exceptions, though not completely rejecting the idea. Spring was one of the first Java framework/library that almost completely got rid of checked exceptions in its own classes (though it has some facilities to support them, as it is impossible to void them while working with legacy code and third-party classes).

His views are expressed mainly in the Spring Framework design/documentation, his books, and comments on forums.

Brian Goetz

Brian Goetz is a programming expert and architect. Worked at Quiotix (1992 - 2006), Sun Microsystems (2006 - 2010) and Oracle (since 2010).

As a Sun/Oracle expert/architect he participated in the development of several Java features, including concurrency utilities. As a specification lead, he oversaw the development of Java Lambda Expressions.

Brian’s views on the checked vs unchecked exceptions seems to be somewhat closer to the “middle ground” [17]. He discourages programmers from excessive use of checked exceptions; at the same time, he does not support the idea of completely getting rid of them.

Bruce Eckel

Bruce Eckel is an IT expert, independent researcher, contractor, and author.

Founded and worked at MindView, LLC (since 2011). Also worked at Fluke and was a member of ANSI/ISO C++ standard committee. Wrote multiple books on C++, Java, Python, Scala, Kotlyn, and OOP design.

In the beginning Bruce Eckel supported Sun/Oracle orthodox view on checked exceptions in Java, but then his views changed to being rather an opponent of the idea [14].

His views are expressed in books, articles, interviews, and his personal blog.

Gavin King

Gavin King is an Australian programming expert and architect.

He worked at TARMS(1999 - 2000), Cirrus Technologies (2000 - 2002), Expert IS (2002 - 2003), JBoss (2003 - 2006), Red Hat (2006 - 2019) and IBM (since 2019). He is the father of Hibernate and Seam Frameworks, as well as Ceylon language.

Based on the fact that Hibernate was one of the first frameworks that almost completely got rid of checked exceptions; Seam does not use them and Ceylon as a language does not have them at all, it looks like Gavin is not a big proponent of checked exceptions. Nevertheless, in his blog he still advocates their usage in certain cases [18].

His views are expressed in books and blogs as well can be seen in the architectural decisions he made.

Stephen Colebourne

Stephen Colebourne is an IT expert of British origin, blogger, and conference speaker.

He worked at RSA (1994 - 2000), SITA (2000 - 2010), Joda.org (since 2000) and OpenGamma (since 2010). He was a co-lead of Oracle JSR-310.

Stephen is a strong opponent of the idea of checked exceptions in Java [19].

His views can be mainly found on his blogs, where he between other things actively discussed the checked vs unchecked exceptions controversy.

Bill Venners

Bill Venners is a programming expert and architect. Artima, Inc president (since 2000).

As a Java expert he actively published articles in Java World magazine, then on Artima website. Now his major point of interest is Scala.

Originally Bill fully supported and reproduced Sun/Oracle view on checked exceptions [20], but later started questioning it.

His views can be mainly found in Java World magazine archives, Artima website, and his books (finished and unfinished).

Others

The opinion on the subjects was also expressed by many lesser known in Java world (some of them though are more known in the non-Java world) IT experts and programmers in their blogs and articles.

Including, but not limited to Tomas Whitmore (“Checked exceptions: Java’s biggest mistake”), Misko Hevery (“Checked exceptions I love you, but you have to go”), Howard Lewis Ship (“The Tragedy Of Checked Exceptions”), Philipp Hauer (“Checked Exceptions are Evil”), Jakob Jenkov (“Checked or Unchecked Exceptions?”), Ohad Shai (“It’s almost 2020 and yet… Checked Exceptions are still a thing”), Jeff Friesen (“Are checked exceptions good or bad?”), Yegor Bugayenko (“Checked vs. Unchecked Exceptions: The Debate Is Not Over”), Rodney Waldhoff (“Java's checked exceptions were a mistake (and here's what I would like to do about it)”).

Websites & Forums

Sun/Oracle

Java started as a project by Sun Microsystems in 1996. In 2009 Sun Microsystems was bought by Oracle.

Archived Sun, and modern Oracle official web sites published several tutorials, guidelines, and articles on the subject of exceptions. We reasonably assume, that whatever is published there can be considered the official position of the Java developing team on certain topics.

The authors of the documents by Oracle acknowledge that there is a controversy among programmers over the usage of checked exceptions [1]. But they stick to their own orthodox views and advocate use of checked exception [15].

Artima

Though www.artima.com web site belongs to Artima Inc, which main purpose is popularization of Scala programming language, in addition to Scala related topics it often publishes and discusses a broad set of programming topics and problems.

In 2003, the website published two interview with the programming world luminaries Bruce Eckel, Anders Hejlsberg, and James Gosling [16] [13]. The subject of the interviews was checked vs unchecked exceptions controversy. The interviews sparked an active discussions between the website users and users on other platforms.

Stack Overflow

Stack Overflow (www.stackoverflow.com) is a popular question and answer website for IT professionals.

There was more than one discussion on the subject of checked vs unchecked exceptions [21]. Though the website users did not come to an agreement, the discussion is interesting because both camps were presented there with various arguments [22].

The Server Side

The Server Side (www.theserverside.com) is a Java community discussing server.

Soon after the Artima interviews with Bruce Eckel, Anders Hejlsberg, and James Gosling there was a discussion concerning these interviews on the Server Side. Interesting fact is that Rod Johnson was a participant of the conversation and expressed his opinion on the matter.

Reddit

Reddit is a discussion platform. There were multiple discussions published on Reddit website concerning checked exceptions [23].

Some of them were independent, some of them referenced other articles and post published outside of

Reddit.

Hacker News

The Hacker News (news.ycombinator.com) is another discussion platform that published multiple discussions on articles on the subject of checked exceptions [24].

Evolution of Views

In the beginning, when Java was released experts in their books and articles normally simply repeated Sun orthodox view on and definition of checked and unchecked exceptions. They almost copied the guidelines provided by the company for checked exception use.

But, as time elapsed and more and more real projects were completed, programmers and experts started to question the dogma. This started to happen around 2000. The discussion fairly quickly reached its maximum and then went down. It does not look like one of the sides was able to convince the opponents, but rather like everyone made their own choices and decided to follow them. From time to time though, the debate flares up again.

Thus, Bruce Eckel was fully supportive of the checked exceptions in his book “Thinking in Java”. But later he changed his opinion and started questioning this practice.

In a similar manner Bill Venners was in line with Sun/Oracle on the subject, but later started questioning it as well.

Though Rod Johnson, Gavin King and Brian Goetz admit that checked exception may be useful, they warn against nondiscretionary use of them. In their own projects Johnson and King pushed the border between checked and unchecked exceptions far towards the checked once. Therefore, even if they theoreticize about possible usefulness of the former, they prefer to avoid them. Spring Framework originally released in 2003, as well as Hibernate version, released in 2005 almost did not use checked exceptions. Other Java project the two participated in, we well as non-Java projects do not promote the idea of checked exceptions.

Oracle sticks to their orthodox view and does not change it. But the ratio of unchecked exception to all exceptions in core Java classes tends to slightly increase as newer versions are released.

Experts associated with Sun/Oracle tend to support the idea of checked exceptions, but they as well warn against excessive use of them.

Majority of popular frameworks created after 2000 either do not use checked exception, use them very little, or use them mainly for backward compatibility and integration with third party libraries.

The last thing to emphasize is that frameworks, libraries, and the corresponding developers teams that do not use checked exceptions are able to avoid enforced handling and all of the inconvenience associated with it. Other than that, the conditions transmitted by the corresponding exceptions may be more than recoverable.

Summary

The fact that no other language (including those that were released after Java) uses checked exceptions seems to reinforce the fact that the introduction of checked exceptions was, in retrospect, a mistake.

But this is not the end of the story.

Checked exceptions are numerous in core Java classes. Multiple libraries and frameworks use them. Even in any random team of developers there may be people that support checked exceptions and tend to use them.

Checked exceptions are already an integral part of the Java programming language. Even if your project architects decide to not use them, core Java and third-party libraries are still there. Checked exceptions are an inevitable part of any past, current, and future Java project.

Even if you are a strong opponent of checked exceptions, the culture of working with them is a must for any professional Java programmer. This is true, at least, for the Java language as we know it. There may be some significant changes in the language or some new language (something like Java Next) that will get rid of them or change the way they are used. But until then, it is important to know them, understand them, and be able to live with them.

7. Real API Examples

In this chapter we will go through several framework examples. Some of them are quite dated and used only in legacy applications, while others are still actively in use.

Java SE

From the very beginning of Java checked und unchecked exceptions were used in core Java classes. This is absolutely in line with the official Sun and Oracle documentation and tutorials' position that promotes active usage of both types of exceptions.

In JDK 1.0 there were 51 classes that extended **java.lang.Exception** class, 19 classes extended **java.lang.Error** and 15 extended **java.lang.RuntimeException**. Altogether it gives us 34 unchecked exceptions and 17 checked exceptions.

In JDK 1.1 there were 115 classes that extended **java.lang.Exception** class, 20 classes extended **java.lang.Error** and 28 extended **java.lang.RuntimeException**. Altogether it gives us 48 unchecked exceptions and 67 checked exceptions.

In JDK 1.2 there were 205 classes that extended **java.lang.Exception** class, 21 classes extended **java.lang.Error** and 89 extended **java.lang.RuntimeException**. Altogether it gives us 110 unchecked exceptions and 95 checked exceptions.

In JDK 1.3 there were 233 classes that extended **java.lang.Exception** class, 21 classes extended **java.lang.Error** and 93 extended **java.lang.RuntimeException**. Altogether it gives us 114 unchecked exceptions and 139 checked exceptions.

In Java J2SE 1.4, there were 332 classes that extended **java.lang.Exception** class, 26 classes extended **java.lang.Error** and 91 extended **java.lang.RuntimeException**. Altogether it gives us 117 unchecked exceptions and 241 checked exceptions.

In J2SE 5.0, there were 432 classes that extended **java.lang.Exception** class, 27 classes extended **java.lang.Error** and 131 extended **java.lang.RuntimeException**. Altogether it gives us 158 unchecked exceptions and 301 checked exceptions.

In Java SE 6, there were 472 classes that extended **java.lang.Exception** class, 31 classes extended **java.lang.Error** and 143 extended **java.lang.RuntimeException**. Altogether it gives us 174 unchecked exceptions and 330 checked exceptions.

In Java SE 7, there were 506 classes that extended **java.lang.Exception** class, 32 classes extended **java.lang.Error** and 162 extended **java.lang.RuntimeException**. Altogether it gives us 194 unchecked exceptions and 344 checked exceptions.

In Java SE 8, there were 514 classes that extended **java.lang.Exception** class, 33 classes extended **java.lang.Error** and 169 extended **java.lang.RuntimeException**. Altogether it gives us 202 unchecked exceptions and 345 checked exceptions.

In Java SE 9, there were 526 classes that extended **java.lang.Exception** class, 33 classes extended **java.lang.Error** and 179 extended **java.lang.RuntimeException**. Altogether it gives us 212 unchecked exceptions and 347 checked exceptions.

In Java SE 10, there were 526 classes that extended **java.lang.Exception** class, 33 classes extended **java.lang.Error** and 179 extended **java.lang.RuntimeException**. Altogether it gives us 212 unchecked exceptions and 347 checked exceptions.

In Java SE 11, there were 422 classes that extended **java.lang.Exception** class, 33 classes extended **java.lang.Error** and 131 extended **java.lang.RuntimeException**. Altogether it gives us 164 unchecked exceptions and 291 checked exceptions.

In Java SE 12, there were 422 classes that extended **java.lang.Exception** class, 33 classes extended **java.lang.Error** and 131 extended **java.lang.RuntimeException**. Altogether it gives us 164 unchecked exceptions and 291 checked exceptions.

In Java SE 13, there were 422 classes that extended **java.lang.Exception** class, 33 classes extended **java.lang.Error** and 131 extended **java.lang.RuntimeException**. Altogether it gives us 164 unchecked exceptions and 291 checked exceptions.

In Java SE 14, there were 419 classes that extended **java.lang.Exception** class, 33 classes extended **java.lang.Error** and 131 extended **java.lang.RuntimeException**. Altogether it gives us 164 unchecked

exceptions and 288 checked exceptions.

In Java SE 15, there were 419 classes that extended **java.lang.Exception** class, 33 classes extended **java.lang.Error** and 131 extended **java.lang.RuntimeException**. Altogether it gives us 164 unchecked exceptions and 288 checked exceptions.

In Java SE 16, there were 419 classes that extended **java.lang.Exception** class, 33 classes extended **java.lang.Error** and 131 extended **java.lang.RuntimeException**. Altogether it gives us 164 unchecked exceptions and 288 checked exceptions.

Sun and then Oracle engineers from time to time refactor their core classes, declare some classes deprecated and later remove them from the Java SE API, as well as introducing some new classes. So, the total number of exceptions used, as well as ration of checked to unchecked exceptions may change over time.

After several original versions, the overall trend remains the same – in core Java classes approximately one third of exceptions are unchecked exception and two thirds are checked.

Java Persistence

In EJB 2.1, there were 11 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 6 extended **java.lang.RuntimeException**. Altogether it gives us 6 unchecked exceptions and 5 checked exceptions.

In EJB 3.0 (without JPA 1.0), there were 16 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 11 extended **java.lang.RuntimeException**. Altogether it gives us 11 unchecked exceptions and 5 checked exceptions.

In JPA 1.0, there were 8 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 11 extended **java.lang.RuntimeException**. Altogether it gives us 8 unchecked exceptions and 0 checked exceptions.

In JPA 2.0, 2.1 and 2.2 there were 11 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 11 extended **java.lang.RuntimeException**. Altogether it gives us 11 unchecked exceptions and 0 checked exceptions.

The overall trend – the Java Presistence developers at some point in time decide to use only unchecked exceptions for the new code and completely discard the idea of using checked ones.

Hibernate

In Hibernate 1.0, there were 16 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 2 extended **java.lang.RuntimeException**. Altogether it gives us 2 unchecked exceptions and 14 checked exceptions.

In Hibernate 2.0, there were 22 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 2 extended **java.lang.RuntimeException**. Altogether it gives us 2 unchecked exceptions and 20 checked exceptions.

In Hibernate 3.0, there were 39 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 33 extended **java.lang.RuntimeException**. Altogether it gives us 33 unchecked exceptions and 6 checked exceptions.

In Hibernate 4.0, there were 90 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 88 extended **java.lang.RuntimeException**. Altogether it gives us 88 unchecked exceptions and 2 checked exceptions.

In Hibernate 5.0, there were 127 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 124 extended **java.lang.RuntimeException**. Altogether it gives us 124 unchecked exceptions and 3 checked exceptions.

In Hibernate 6.0, there were 139 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 138 extended **java.lang.RuntimeException**. Altogether it gives us 138 unchecked exceptions and 1 checked exceptions.

The overall trend is evident – the Hibernate developers at some point in time decide to use only unchecked exceptions for the new code and completely discard the idea of using checked ones, in the latest version there is only one checked exception.

Spring 1.x-5.x

In Spring 1.0, there were 77 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 75 extended **java.lang.RuntimeException**. Altogether it gives us 75 unchecked exceptions and 2 checked exceptions.

In Spring 2.0, there were 159 classes that extended **java.lang.Exception** class, 0 classes extended

java.lang.Error and 143 extended **java.lang.RuntimeException**. Altogether it gives us 143 unchecked exceptions and 16 checked exceptions.

In Spring 3.0, there were 203 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 181 extended **java.lang.RuntimeException**. Altogether it gives us 181 unchecked exceptions and 22 checked exceptions.

In Spring 4.0, there were 231 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 205 extended **java.lang.RuntimeException**. Altogether it gives us 205 unchecked exceptions and 26 checked exceptions.

In Spring 5.0, there were 244 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 224 extended **java.lang.RuntimeException**. Altogether it gives us 224 unchecked exceptions and 20 checked exceptions.

Though Spring still uses some checked exceptions, the overwhelming majority of exceptions is unchecked.

Apache Hadoop 1.x-3.x

In Apache Hadoop 1.0.4, there were 24 classes that extended **java.lang.Exception** class, 2 classes extended **java.lang.Error** and 4 extended **java.lang.RuntimeException**. Altogether it gives us 6 unchecked exceptions and 20 checked exceptions.

In Apache Hadoop 1.2.1, there were 26 classes that extended **java.lang.Exception** class, 2 classes extended **java.lang.Error** and 4 extended **java.lang.RuntimeException**. Altogether it gives us 6 unchecked exceptions and 22 checked exceptions.

In Apache Hadoop 2.6.0, there were 38 classes that extended **java.lang.Exception** class, 2 classes extended **java.lang.Error** and 7 extended **java.lang.RuntimeException**. Altogether it gives us 9 unchecked exceptions and 31 checked exceptions.

In Apache Hadoop 2.10.1, there were 58 classes that extended **java.lang.Exception** class, 2 classes extended **java.lang.Error** and 12 extended **java.lang.RuntimeException**. Altogether it gives us 12 unchecked exceptions and 48 checked exceptions.

In Apache Hadoop 3.0.0, there were 54 classes that extended **java.lang.Exception** class, 1 classes extended **java.lang.Error** and 13 extended **java.lang.RuntimeException**. Altogether it gives us 14 unchecked exceptions and 41 checked exceptions.

In Apache Hadoop 3.3.0, there were 64 classes that extended **java.lang.Exception** class, 1 classes extended **java.lang.Error** and 13 extended **java.lang.RuntimeException**. Altogether it gives us 14 unchecked exceptions and 51 checked exceptions.

Amazon WS

In AWS Java SDK 1.0.0, there were 60 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 60 extended **java.lang.RuntimeException**. Altogether it gives us 60 unchecked exceptions and 0 checked exceptions.

In AWS Java SDK 1.1.0, there were 104 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 104 extended **java.lang.RuntimeException**. Altogether it gives us 104 unchecked exceptions and 0 checked exceptions.

In AWS Java SDK 1.2.0, there were 123 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 123 extended **java.lang.RuntimeException**. Altogether it gives us 123 unchecked exceptions and 0 checked exceptions.

In AWS Java SDK 1.3.0, there were 198 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 198 extended **java.lang.RuntimeException**. Altogether it gives us 198 unchecked exceptions and 0 checked exceptions.

In AWS Java SDK 1.4.0, there were 343 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 342 extended **java.lang.RuntimeException**. Altogether it gives us 342 unchecked exceptions and 1 checked exceptions.

In AWS Java SDK 1.5.0, there were 362 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 361 extended **java.lang.RuntimeException**. Altogether it gives us 361 unchecked exceptions and 1 checked exceptions.

In AWS Java SDK 1.6.0, there were 373 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 372 extended **java.lang.RuntimeException**. Altogether it gives us 372 unchecked exceptions and 1 checked exceptions.

In AWS Java SDK 1.7.0, there were 422 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 421 extended **java.lang.RuntimeException**. Altogether it gives us 421 unchecked exceptions and 1 checked exceptions.

In AWS Java SDK 1.8.0, there were 441 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 439 extended **java.lang.RuntimeException**. Altogether it gives us 439 unchecked exceptions and 2 checked exceptions.

In AWS Java SDK 2.16.47, there were 3829 classes that extended **java.lang.Exception** class, 0 classes extended **java.lang.Error** and 3825 extended **java.lang.RuntimeException**. Altogether it gives us 3825 unchecked exceptions and 4 checked exceptions.

As one can see, the AWS Java SDK architects from the very beginning almost did not use checked exception and they keep following this pattern.

Research Study of UOW

The research study [12] by the University of Waterloo covers the source code of more than 550,000 Java projects in the GitHub repository and more than 50,000 Java project in the SourceForge repository, as of September 2015. It is a static analysis of exception handling practices.

Detailed information can be found in the original paper, which analyzes the information given in the research study from different perspectives.

We will outline only aggregated, extrapolated and the most relevant facts from this study.

The subject of the study is catch blocks, since (according to the authors) they provide the most accurate static picture of exception handling practices.

The overwhelming majority (95%-99%) of catch blocks intercept standard exceptions, which is in line with the recommendation to favor their use.

Around of 45-47% catch blocks intercepts java.lang.Throwable or java.lang.Exception. This fact clearly violates the design pattern that a catch block normally should not intercept base exception classes.

Around 20% of catch blocks are empty. This is in clear violation of design patterns and common sense.

Around 15% of catch blocks intercept exceptions that indicated programming bugs (from java.lang) IllegalArgumentException, NumberFormatException, IllegalArgumentException, etc. This is another violation of sound coding practices.

Poor alternative to logging (printStackTrace() and System.out.println()) takes place in 20% of catch blocks, whereas logging with one of popular frameworks takes place in 18%.

Around 23% of the caught exceptions are wrapped and re-thrown, most of the time (around 80%) as unchecked exceptions.

Around 90% of time when checked exceptions are re-thrown, they are re-thrown as unchecked exceptions.

Analysis by OverOps

Another form of analysis [25] was done by OverOps software analytics company. Their agent software is installed in the client Java environment and collects the runtime statistics.

In the study they monitored more than 1,000 applications.

More detailed information can be found on their website, we just aggregate and extrapolate some more prominent facts from it and provide our own analysis.

97% of logged exceptions are caused by 3% of unique exception types. The 3% corresponds to 1o types.

An average Java application throws 9.2 million exceptions a month.

An average Java application generates about 2.7 TB of storage a month.

An average Java application throws 53 unique exceptions a month.

The research team selected top 10 exceptions thrown in all projects. Next, they analyzed those exceptions from most to least frequent.

Let us revisit those results.

In **70%** of analyzed environments **NullPointerException** (a subclass of java.lang.RuntimeException) was in the top 10 exceptions thrown. This exception indicates a typical programming bug where the code that throws it does not initialize the corresponding reference to any valid object or looses the object somewhere during the process by explicitly assigning reference to null.

In **55%** of analyzed environments **NumberFormatException**(a subclass of java.lang.RuntimeException) was in the top 10 exceptions thrown. This is another programming bug exception. An incorrect String (the one that cannot be converted to a number) is fed to Integer.parseInt() or some similar method. It does not always happen because a user made such call explicitly. It may also happen “in the background”, when for example a processing servlet receives a string parameter and tries to convert it into a number. But this fact does not change the nature of the problem.

In **50%** of analyzed environments **IllegalArgumentException**(a subclass of java.lang.RuntimeException)was between top 10 exceptions thrown. This is another programming bug that is the result of the caller not checking parameters for validity, and the called method has not choice but to throw this exception. It is less specific than the previous two exceptions and funding out what causes it may take more efforts.

In **23%** of analyzed environments **RuntimeException** was in the top 10 of exceptions thrown. This case is very interesting. The fact is that the Java SE classes never throw an instance of these classes, only subclasses. This means that all cases when an exception of this type is thrown, the custom code explicitly called one of RuntimeException constructors and created an instance of it. This is very poor programming practice.

In **22%** of analyzed environments **IllegalStateException**(a subclass of java.lang.RuntimeException)was in the top 10 exceptions thrown. This is yet another programming bug that results from an object being in a condition where calling some of its methods was inappropriate. Untangling the problem may take some time. But is does not change the fact that this exception again indicates a programming bug.

In **16%** of analyzed environments **NoSuchMethodException** (a subclass of java.lang.Exception) was in the top 10 exceptions thrown. Reflection is widely used by multiple Java technologies, when the method Java reflection API is trying to invoke does not exist, this exception is thrown. It appears this situation is very common.

In **15%** of analyzed environments **ClassCastException** was in the top 10 exceptions thrown.This is another programming bug. Casting a class is something that normally should be avoided, but sometimes it is unavoidable.

In **15%** of analyzed environments **Exception** was in the top 10 exceptions thrown.This is another programming bug. Casting a class is something that normally should be avoided, but sometimes it is unavoidable. This case is similar to the case of RuntimeException. Again, the Java SE classes never throw an instance of this classes. Only subclasses. Which means that all cases when an exception of this type is thrown, the custom code explicitly called one of Exception constructors and created an instance of it. This is very poor programming practice.

In **13%** of analyzed environments **ParserException** was in the top 10 exceptions thrown. This exception is similar to NumberFormatException, but contrary to it is a checked exception. The fact that it is a checked exception in a list of unchecked (we do not consider Exception as a checked one) demonstrates the strange decision made by Sun/Oracle architects. This exception in its nature corresponds to a programming bug, but somehow was declared as a direct subclass of Exception and is considered checked. It would not be a mistake to say, that it indicates a programming bug.

In **13%** of analyzed environments **InvocationTargetException**was in the top 10 exceptions thrown. Though declared as a checked exception, it is yet another programming bug. Another reflection associated exception, the wrapper for an exception thrown by the reflected method.

In summary, it seems like the majority of the exceptions thrown in an average Java application are the result of logical and medium level programming bugs (that are undetected and unprocessed) turn into lower level, more generic exceptions.

Observations

From the real API examples, we can observe the following facts.

The trend toward reducing the use of checked exceptions is evident. Old frameworks almost stopped using them in the early 2000s, and newer frameworks almost do not use them at all.

The culture of exception handling in production projects is quite low. Even shallow code analysis shows that, at a minimum, half of the catch blocks are written in clear violation of good design principals and design patterns. They either try to catch everything relevant and irrelevant, try to catch programming bugs, or do nothing when they catch something.

Analysis of actual execution in production shows that most of the exceptions actually thrown are mainly those that correspond to low level programming mistakes. It does not mean that most of programming bugs take place at low level, it means that they are detected at low level. It is most likely (though this fact requires additional analysis) that most originate at a high logical level, or medium processing level. Though the dynamic analysis by OverOps showed different aspects of how exception handling takes place in real world enterprise applications, it nevertheless confirmed the sad fact that the culture of handling exceptions is low. The fact that only 23% of applications explicitly create instances of java.lang.RuntimeException class (and it is one of the most common exception) is telling.

8. Modern Challenges

Exception handling is often viewed as concern exclusive to programmers. The reality is that this is far from the truth.

Definitely there are purely programming aspects associated with exception handling: design principles, good/bad coding practices, etc. But focusing on only these aspects is not enough.

Non-“True Path” Cases

System requirements are normally written from the perspective of a success-oriented mindset, they concentrate on the correct outcome of the program (as we already said, Bill Venners and James Gosling called it “the one true path”). Rarely is it discussed what happens if there is an unexpected situation. This challenge is implicitly given to the programmers. The programmers, in turn, try to make the product work; for it to do its required job. The question of unexpected situations is swept under the rug.

We are not saying that requirements do not cover major cases when the program receives an incorrect input or data such as incorrect password, wrong account number, not enough funds on the users account, etc. These cases are normally covered. Instead, we are referring to less visible issues.

For example, say you are creating an online flight booking website. A user selects the tickets, enters his information and proceeds to payment. If there are not enough funds to pay for the tickets, the system will immediately notify the user and require selecting another method of payment. This scenario is taken care of, and the programmers coded it correctly. After the payment was successfully made, the system made changes to its own storage of information, makes calls to some external systems (for example to PayPal), etc., then it is supposed to send an email to the user with the ticket information. Normally it is done in a separate thread. It may be a scheduler task that scans changes from time to time and generates emails or it may be an asynchronous message that notifies a corresponding listener to generate the email. Whatever it is, sending the email notification in this case is a secondary activity, detached from the main process.

What happens if it fails? For example, an email template needed to generate and send the email was deleted, and a new one was not uploaded. As we said earlier, this activity is detached from the main process, so the original transaction(s) and changes will not be rolled back, but the email is not sent, and the user does not know the details of his flight. Moreover, the developers may solemnly reply on the idea that the template is always at place. The value object that corresponds to the template is always valid. But it turns out to be null. An attempt to do something with this object results in a nasty NullPointerException.

What should be done?

The developer may perform a check for not found template and make a corresponding debug record. While this will not solve the problem, it will at least make finding its root cause easier. The developers may create a scheduler that rescans the changes and sends emails to the users that were supposed to but failed to receive corresponding notifications. There are many things that can be done to eliminate, or at least minimize the outcome of the fact that the template is missing. But, as we said earlier, in this scenario it is not clearly articulated in the requirements, all that will be given to the discretion of the developers, which in turn are generally inclined to do nothing. Until something bad happens.

In the scenarios similar to the one above most of the time there will be a requirement to upload the template, possibly the requirement to provide a convenient interface how to delete, upload and/or update the template. Nevertheless, the explanation of what the system is supposed to do if the template is missing will not be there.

The moral of the story is that secondary failure scenarios are often overlooked, not planned, and not thought ahead of time. There is no complex and exhaustive approach to failure recovery. The approach is rather selective and optimistic. As a result, the resolution of these situations is often manual, time consuming and frustrating. Often it results in the appropriate changes in the system, but at much greater price.

Lack of Systemic Approach

In addition to simply ignoring possible exceptional situations, they may be handed improperly. And this is exactly what happens on a very large scale.

We already mentioned the studies on real Java projects and how they handle exceptions in the

“Observation” paragraph.

Programmers on a large scale either do not know the basics of proper exceptions handling, or simply ignore them. All that clearly indicates that there is lack of systemic approach to exception handling in actual Java projects.

Increased Complexity

Old interviews given by experts, as well as articles and books on Java, often describe exception handling in terms of “message pump”, “message loop”, “UI”. This means that back then most applications were standalone, and UI based. The same is true of their Web-based successors. There was a main single thread that handled the application logic, if there was some failure during the execution, this failure in some form was immediately reported to the user in the form of a message.

Modern applications, compared to their 20+ year-old predecessors, are much more distributed and asynchronous. As a result, they are much more undetermined in terms of the final outcome. The result is not immediately known.

For example, you make an online order, the system checks availability of the product, finds that everything is fine and successfully registers you order. But ten minutes later, after it receives the updated data, it finds out that the product is no longer available. These situations are normally expected, but there may be more than one action like this in the chain which renders the entire process more complicated. In addition, there may be some technical issues in the chain, resulting in even more increased complexity.

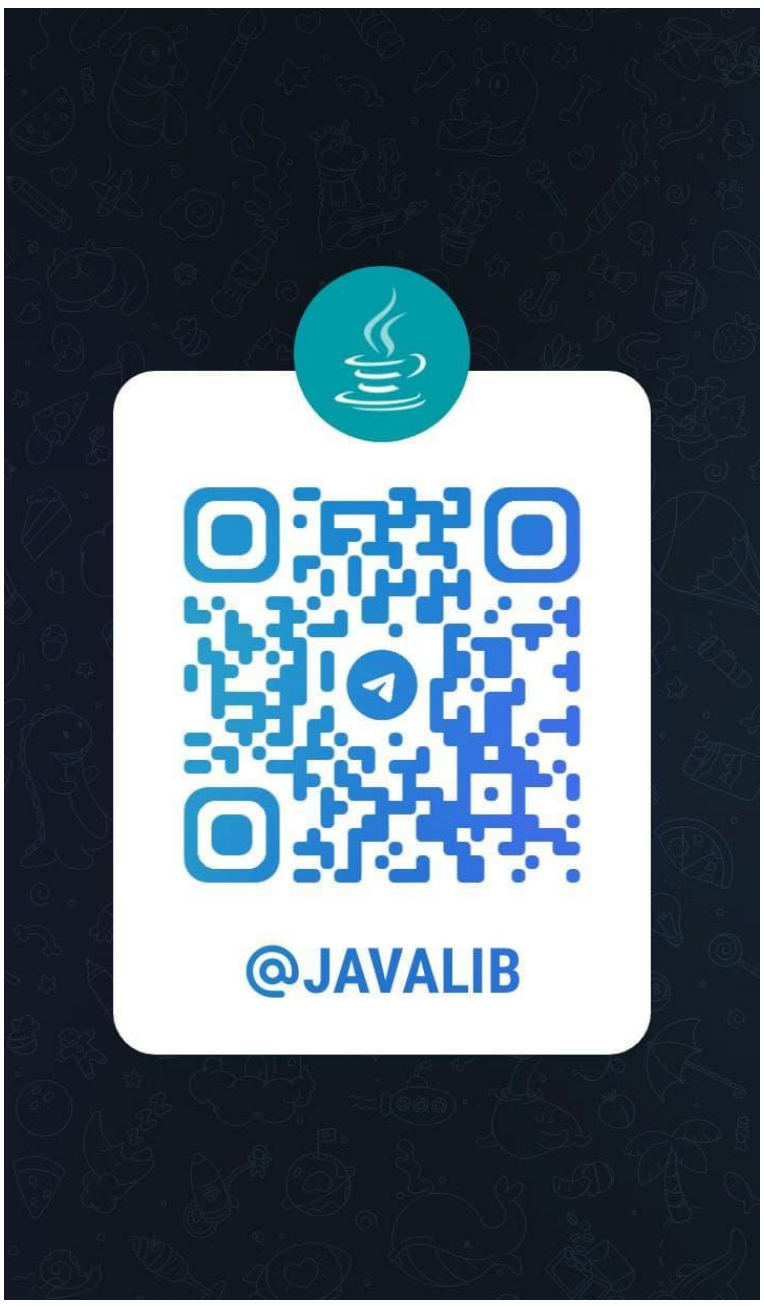
In summary, today, communicating and handling failures is a nontrivial task.

Summary

The problem of not carefully considering possible failures is a longstanding problem, articulated long ago. It is universal and timeless in nature.

The problem of increased complexity of failures in modern applications is relatively new.

It is likely that they both will continue to be overseen. This will, as a result, demand more complex and careful consideration of exception handling in the future.



9. References

[1] B. Ruzek, "Effective Java Exceptions," 10 January 2007. [Online]. Available: https://www.oracle.cc resources/articles/enterprise-architecture/effective-exceptions-part1.html.

[2] Oracle, "The Java Virtual Machine Specification, Java SE 16 Edition," 12 February 2021. [Online] https://docs.oracle.com/javase/specs/jvms/se16/jvms16.pdf.

[3] Oracle (Java Platform, Standard Edition Tools Reference), "Main Tools to Create and Build Applicatior Command," 12 February 2021. [Online]. Available: https://docs.oracle.com/en/java/javase/16/docs/specs/man/javac

[4] A. Shipilëv, "The Exceptional Performance of Lil' Exception," 11 January 2014. [Online]. https://shipilev.net/blog/2014/exceptional-performance/.

[5] ResearchGate, "A Study of Exception Handling and Its Dynamic Optimization in Java," December 20(Available:
https://www.researchgate.net/publication/2573860_A_Study_of_Exception_Handling_and_Its_Dynamic_Optimizat

[6] ResearchGate, "Efficient Java Exception Handling in Just-in-Time Compilation," November 2000. [Online https://www.researchgate.net/publication/2433780_Efficient_Java_Exception_Handling_in_Just-in-Time_Compilat

[7] Oracle, "The Java® Language Specification Java SE 16 Edition," 12 February 2021. [Online] https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf.

[8] Open JDK website, Mailing List, "Inferring Exception type of a lambda body," 5 February 2010. [Online http://mail.openjdk.java.net/pipermail/lambda-dev/2013-February/007981.html.

[9] Oracle, "Java® Platform, Standard Edition & Java Development Kit, Version 16 API Specification," 2 Fe [Online]. Available: https://docs.oracle.com/en/java/javase/16/docs/api/index.html.

[10] Oracle, "Programming With Assertions," [Online]. https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html.

[11] Oracle, "Assertion Facility," [Online]. https://web.archive.org/web/20140512115520/http://www.oracle.com/technetwork/articles/javase/javapch06.pdf.

[12] Suman Nakshatri, Maithri Hegde, Sahithi Thandra, David R. Cheriton, "Analysis of Exception Handling Pa Projects: An Empirical Study," School of Computer Science University of Waterloo Ontario, Canada, 14 May 20 Available: https://ieeexplore.ieee.org/abstract/document/7832935. [Accessed 23 October 2021].

[13] Artima, "Failure and Exceptions, A Conversation with James Gosling, Part II," 23 September 2003. [Online https://www.artima.com/articles/failure-and-exceptions. [Accessed 23 October 2012].

[14] B. Eckel, "Does Java need Checked Exceptions?," 11 June 2002. [Online]. https://web.archive.org/web/20020611051228/http://www.mindview.net/Etc/Discussions/CheckedExceptions. [Acc 2020].

[15] Oracle (The Java™ Tutorials), "Exceptions, Unchecked Exceptions — The Controversy," [Online] https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html.

[16] Artima, "The Trouble with Checked Exceptions, A Conversation with Anders Hejlsberg, Part II by Bill V Bruce Eckel," 18 August 2003. [Online]. Available: https://www.artima.com/intv/handcuffs.html. [Accessed 2 2020].

[17] IBM: developersWork, "Java theory and practice; The exceptions debate: To check, or not to check? E [Online]. Available: https://web.archive.org/web/20150407111821/http://www.ibm.com/developerworks/library/ jtp05254-pdf.pdf.

[18] Blog of Gavin King, "Object construction and validation," 5 April 2016. [Online]. Available: h lang.org/blog/2016/04/05/object-validation/.

[19] Stephen Colebourne's blog, "Checked Exceptions (#bijava)," 23 September 2010. [Online]. https://blog.joda.org/2010/09/checked-exceptions-bijava_9688.html.

[20] Artima, "Interface Design by Bill Venners: Use checked exceptions for conditions that client code may r expected to handle," [Online]. Available: https://www.artima.com/interfacedesign/CheckedExceptions.html.

[21] Stack Overflow, "Discussions on checked exceptions," 23 October 2021. [Online]. https://stackoverflow.com/search?q=checked+exceptions.

[22] StackOverflow, "The case against checked exceptions," 5 March 2009. [Online]. https://stackoverflow.com/questions/613954/the-case-against-checked-exceptions.

[23] Reddit, "Discussions on checked exceptions," 23 October 2021. [Online]. Available: https://www.reddit. q=checked%20exception.

[24] Hacker News, "Discussions on checked exceptions," [Online]. Available: https://hn.a
q=checked+exceptions.

[25] A. Zhitnitsky, "The complete guide to Solving Java Application Errors in Production," OverOps, 2 June 20
Available: https://land.overops.com/java-applic
utm_source=blog&utm_medium=post&utm_campaign=postchapterCTA.