**CHECKED(exception) / UNCHECKED(runtimeexception)**

**HANDLED / UNHANDLED(STOPS)**

<https://dzone.com/articles/how-to-deal-with-exceptions>

An old and common guideline when it comes to checked vs unchecked is that runtime exceptions are used to signal situations which the application usually cannot anticipate or recover from, while checked exceptions are situations that a well-written application should anticipate and recover from

Well, I am an advocate of *only using runtime exceptions*. And if I use a library that has a method with a checked exception, I create a wrapper method that turns it into a runtime

So try to *use standard exceptions*, as other developers will understand what happened easier

When seeing a NullPointerException, the reason is clear to anyone.

In the boundaries of my API (let’s say the endpoints of my REST service) I always have generic catch Exception clauses. I do not want any surprises and something that I did not manage to predict or guard against in my code, to potentially reveal things to the outside world.

If an exception is thrown from the lower levels of our program, such as a database related exception, it does not have to provide the details to the caller of our API. Catch the exception and throw a more abstract one, that simply informs callers that their attempted operation failed

In these cases, *make the whole history of the exception available* from throw to throw by passing the original exception to the constructor of the new exception.

Make sure you wait until you have all the information to handle the exception properly. This can be tied to the *throw early-catch late principle*

* Try to avoid exceptions. Use the language features and proper design in order to achieve it.
* Use runtime exceptions, wrap methods with checked exceptions and turn them in at runtime.
* Try to use standard exceptions.
* Make your exceptions specific and descriptive.
* Catch the most specific exception first.
* Do not catch on Exception.
* But catch on Exception on the boundaries of your API. Have complete control over what comes out to the world.
* Create a hierarchy of exceptions that match the layers and functionalities of your application.
* Throw exceptions at the proper abstraction level. Catch an exception and throw a higher level one as you move from layer to layer.
* Pass the complete history of exceptions when rethrowing by providing the exception in the constructor of the new one.
* Think of the try-catch-finally block as a transaction. Make sure you leave your program in a valid state when something goes wrong.
* Catch exceptions when you can handle it.
* Never have empty catch clauses.
* Log an exception when you handle it.
* Have a global exception handling service and have a strategy on how you handle errors.

<https://martinfowler.com/articles/replaceThrowWithNotification.html>

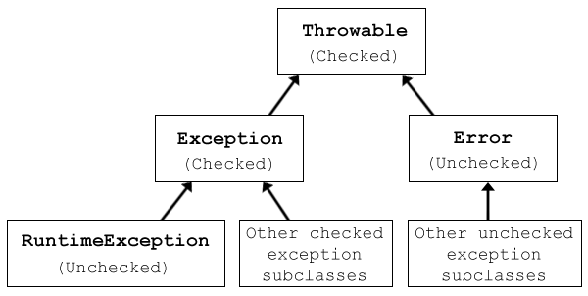
***Martin Fowler: “if a failure is expected behavior, then you shouldn't be using exceptions”***

Exceptions are a very useful technique for handling exceptional behavior and getting it away from the main flow of logic

This refactoring is a good one to use only when the outcome signaled by the exception isn't really exceptional, and thus should be handled through the main logic of the program

<https://howtodoinjava.com/best-practices/java-exception-handling-best-practices/>

**Type of exceptions**

[](https://howtodoinjava.files.wordpress.com/2013/04/exceptionhierarchy3.png)Exception

**1) Never swallow the exception in catch block**

**Checked exceptions** are exceptions that must be declared in the throws clause of a method. They extend Exception and are intended to be an “in your face” type of exceptions. Java wants you to handle them because they somehow are dependent on external factors outside your program. A checked exception indicates an expected problem that can occur during normal system operation. Mostly these exception happen when you try to use external systems over network or in file system. Mostly, the correct response to a checked exception should be to try again later, or to prompt the user to modify his input.

**Unchecked exceptions** are exceptions that do not need to be declared in a throws clause. JVM simply doesn’t force you to handle them as they are mostly generated at runtime due to programmatic errors. They extend RuntimeException. The most common example is a NullPointerException [Quite scary.. Isn’t it?]. An unchecked exception probably shouldn’t be retried, and the correct action should be usually to do nothing, and let it come out of your method and through the execution stack. At a high level of execution, this type of exceptions should be logged.

**Errors** are serious runtime environment problems that are almost certainly not recoverable. Some examples are OutOfMemoryError, LinkageError, and StackOverflowError. They generally crash you program or part of program. Only a good logging practice will help you in determining the exact causes of errors.

**User defined custom exceptions**

Anytime when user feels that he wants to use its own application specific exception for some reasons, he can create a new class extending appropriate super class (mostly its Exception.java) and start using it in appropriate places. These user defined exceptions can be used in two ways:

**1)** Either directly throw the custom exception when something goes wrong in application

throw new DaoObjectNotFoundException("Couldn't find dao with id " + id);

**2)** Or wrap the original exception inside custom exception and throw it

catch (NoSuchMethodException e) {

throw new DaoObjectNotFoundException("Couldn't find dao with id " + id, e);

}

Wrapping an exception can provide extra information to the user by adding your own message/ context information, while still preserving the stack trace and message of the original exception. It also allows you to hide the implementation details of your code, which is the most important reason to wrap exceptions.

Now lets start exploring the best practices followed for exception handling industry wise.

**Best practices you must consider and follow**

**1) Never swallow the exception in catch block**

|  |
| --- |
| catch (NoSuchMethodException e) {     return null;  } |

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

**2) Declare the specific checked exceptions that your method can throw**

|  |
| --- |
| public void foo() throws Exception { //Incorrect way  } |

Always avoid doing this as in above code sample. It simply defeats the whole purpose of having checked exception. Declare the specific checked exceptions that your method can throw. If there are just too many such checked exceptions, you should probably wrap them in your own exception and add information to in exception message. You can also consider code refactoring also if possible.

|  |
| --- |
| public void foo() throws SpecificException1, SpecificException2 { //Correct way  } |

**3) Do not catch the Exception class rather catch specific sub classes**

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

The problem with catching Exception is that if the method you are calling later adds a new checked exception to its method signature, the developer’s intent is that you should handle the specific new exception. If your code just catches Exception (or Throwable), you’ll never know about the change and the fact that your code is now wrong and might break at any point of time in runtime.

**4) Never catch Throwable class**

Well, its one step more serious trouble. Because java errors are also subclasses of the Throwable. Errors are irreversible conditions that can not be handled by JVM itself. And for some JVM implementations, JVM might not actually even invoke your catch clause on an Error.

**5) Always correctly wrap the exceptions in custom exceptions so that stack trace is not lost**

|  |
| --- |
| catch (NoSuchMethodException e) {     throw new MyServiceException("Some information: " + e.getMessage());  //Incorrect way  } |

This destroys the stack trace of the original exception, and is always wrong. The correct way of doing this is:

|  |
| --- |
| catch (NoSuchMethodException e) {     throw new MyServiceException("Some information: " , e);  //Correct way  } |

**6) Either log the exception or throw it but never do the both**

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

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

**7) Never throw any exception from finally block**

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

This is fine, as long as cleanUp() can never throw any exception. In the above example, if someMethod() throws an exception, and in the finally block also, cleanUp() throws an exception, that second exception will come out of method and the original first exception (correct reason) will be lost forever. If the code that you call in a finally block can possibly throw an exception, make sure that you either handle it, or log it. Never let it come out of the finally block.

**8) Always catch only those exceptions that you can actually handle**

|  |
| --- |
| catch (NoSuchMethodException e) {     throw e; //Avoid this as it doesn't help anything  } |

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

**9) Don’t use printStackTrace() statement or similar methods**

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

**10) Use finally blocks instead of catch blocks if you are not going to handle exception**

|  |
| --- |
| try {    someMethod();  //Method 2  } finally {    cleanUp();    //do cleanup here  } |

This is also a good practice. If inside your method you are accessing some method 2, and method 2 throw some exception which you do not want to handle in method 1, but still want some cleanup in case exception occur, then do this cleanup in finally block. Do not use catch block.

**11) Remember “Throw early catch late” principle**

This is probably the most famous principle about Exception handling. It basically says that you should throw an exception as soon as you can, and catch it late as much as possible. You should wait until you have all the information to handle it properly.

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

**12) Always clean up after handling the exception**

If you are using resources like database connections or network connections, make sure you clean them up. If the API you are invoking uses only unchecked exceptions, you should still clean up resources after use, with try – finally blocks. Inside try block access the resource and inside finally close the resource. Even if any exception occur in accessing the resource, then also resource will be closed gracefully.

**13) Throw only relevant exception from a method**

Relevancy is important to keep application clean. A method which tries to read a file; if throws NullPointerException then it will not give any relevant information to user. Instead it will be better if such exception is wrapped inside custom exception e.g. NoSuchFileFoundException then it will be more useful for users of that method.

**14) Never use exceptions for flow control in your program**

We have read it many times but sometimes we keep seeing code in our project where developer tries to use exceptions for application logic. Never do that. It makes code hard to read, understand and ugly.

**15) Validate user input to catch adverse conditions very early in request processing**

Always validate user input in very early stage, even before it reached to actual controller. It will help you to minimize the exception handling code in your core application logic. It also helps you in making application consistent if there is some error in user input.

For example: If in user registration application, you are following below logic:

1) Validate User  
2) Insert User  
3) Validate address  
4) Insert address  
5) If problem the Rollback everything

This is very incorrect approach. It can leave you database in inconsistent state in various scenarios. Rather validate everything in first place and then take the user data in dao layer and make DB updates. Correct approach is:

1) Validate User  
2) Validate address  
3) Insert User  
4) Insert address  
5) If problem the Rollback everything

**16) Always include all information about an exception in single log message**

|  |
| --- |
| LOGGER.debug("Using cache sector A");  LOGGER.debug("Using retry sector B"); |

Don’t do this.

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

Do it like this:

|  |
| --- |
| LOGGER.debug("Using cache sector A, using retry sector B"); |

**17) Pass all relevant information to exceptions to make them informative as much as possible**

This is also very important to make exception messages and stack traces useful and informative. What is the use of a log, if you are not able to determine anything out of it. These type of logs just exist in your code for decoration purpose.

**18) Always terminate the thread which it is interrupted**

|  |
| --- |
| while (true) {    try {      Thread.sleep(100000);    } catch (InterruptedException e) {} //Don't do this    doSomethingCool();  } |

InterruptedException is a clue to your code that it should stop whatever it’s doing. Some common use cases for a thread getting interrupted are the active transaction timing out, or a thread pool getting shut down. Instead of ignoring the InterruptedException, your code should do its best to finish up what it’s doing, and finish the current thread of execution. So to correct the example above:

|  |
| --- |
| while (true) {    try {      Thread.sleep(100000);    } catch (InterruptedException e) {      break;    }  }  doSomethingCool(); |

**19) Use template methods for repeated try-catch**

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

For example below code tries to close a database connection.

|  |
| --- |
| class DBUtil{      public static void closeConnection(Connection conn){          try{              conn.close();          } catch(Exception ex){              //Log Exception - Cannot close connection          }      }  } |

This type of method will be used in thousands of places in your application. Don’t put whole code in every place rather define above method and use it everywhere like below:

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

**20) Document all exceptions in your application in javadoc**

Make it a practice to javadoc all exceptions which a piece of code may throw at runtime. Also try to include possible course of action, user should follow in case these exception occur.

// Make this class abstract so that developers are forced to create

// suitable exception types only

public abstract class BaseException extends Exception{

    //Each exception message will be hold here

    private String message;

    public BaseException(String msg)

    {

        this.message = msg;

    }

    //Message can be retrieved using this accessor method

    public String getMessage() {

        return message;

    }

}

public class TestExceptions {

    public static void main(String[] args)

    {

        try

        {

            throw new DBExeption.NoData(&quot;No row found for id : x&quot;);

        }

        catch(Exception e)

        {

            e.printStackTrace();

        }

    }

}

Multicatche:

try {

// Dodgy database code here

catch (IOException|SQLException ex) {

<https://javarevisited.blogspot.com/2013/03/0-exception-handling-best-practices-in-Java-Programming.html>

**1) Use Checked Exception for Recoverable error and Unchecked Exception for programming error.**

Choosing between checked and unchecked exception is always been confusing for Java programmers. Checked exceptions ensures that you provide exception handling code for error conditions, which is a way from language to enforcing you for writing robust code, but same time it also add lots of clutter into code and makes it unreadable. Also, it seems reasonable to catch exception and do something if you have alternatives or recovery strategies.

**5) Avoid overusing Checked Exception**

Checked Exception has there advantage in terms of enforcement, but at same time it also litters the code and makes it unreadable by obscuring business logic. You can minimize this by not overusing checked Exception which result in much cleaner code. You can also use newer Java 7 features like [one catch block for multiple exceptions](http://javarevisited.blogspot.com/2011/07/jdk7-multi-cache-block-example-tutorial.html) and [automatic resource management](http://java67.blogspot.com/2012/09/what-is-new-in-java-7-top-5-jdk-7.html), to remove some duplication.

**6) Converting Checked Exception into RuntimeException**

This is one of the technique used to limit use of checked Exception in many of frameworks like Spring ,where most of checked Exception, which stem from JDBC is wrapped into DataAccessException, an unchecked Exception. This Java best practice provides benefits, in terms of restricting specific exception into specific modules, like SQLException into [DAO layer](http://javarevisited.blogspot.com/2013/01/data-access-object-dao-design-pattern-java-tutorial-example.html) and throwing meaningful RuntimeException to client layer.

**7) Remember Exceptions are costly in terms of performance**

Don’t just throw and catch exceptions, if you can use boolean variable to indicate result of operation, which may result in cleaner and performance solution.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<http://literatejava.com/exceptions/checked-exceptions-javas-biggest-mistake/>

**1.  Prefer ‘throw’ to ‘catch’**

Principle here is that, in general, most exceptions are unrecoverable from your code. Your code does not (should not) have the ability, or specific knowledge, to repair the network or restart the database. That would violate encapsulation.

Given that, the responsibility of your code is to **fail immediately & safely, and report an error.** The critical danger to avoid is allowing code to continue with incorrect/ erroneous data, and save or output false results to the business level.

**2.  Categorize exceptions by cause**

Since exceptions come with a stacktrace, it is more useful to categorize them by *subsystem causing failure,* rather than by which application module the failure occurred in.

For example, categorizing exceptions into SQL, file access, or configuration types is generally far more useful than separate types for Customer, Account, and Order modules.

Keep the heirarchy simple. For the vast majority of applications, the application module which failure occurred in is irrelevant — no practical difference in handling or reporting would be made.

It is useful, however, to distinguish major underlying causes of failure/ and different degrees of recoverability.

**3.  Keep your exception heirarchy simple**

A simple hierarchy is easy for developers to use & throw, making it obvious to find “the right exception”. It should offer basic broad categories for diagnosis & handling.

Overly complicated hierarchies, non-obvious naming, or per-module exceptions leave developers scratching their head looking round for what to use. *(This is why people end up using RuntimeException).* It shouldn’t be that hard.

Library code which is genuinely separate from the application body may deserve it’s own exceptions, but don’t go overboard.

A clean & effective hierarchy may look like this:

* FailureException extends RuntimeException
  + AppSQLException
  + AppFileException
  + AppConfigException
  + AppDataException
  + AppInternalException
* RecoverableException extends RuntimeException
  + UserException
  + ValidationException

In this hierarchy, exceptions a developer needs to rethrow will be under an “App\*” exception classname. This makes it easy for developers to find & throw the correct type.

**4.  Catch at the outermost level**

Exceptions must be reported to the business/ external world — and this is done by returning a 500 “error response” or displaying an error in the UI.

Since such an ‘catch’ block or exception-handler requires access to the outside to respond, it means exceptions must be caught & handled at the outermost level.

Simultaneously, the final handler should log the exception & full stacktrace for investigation.

**‘Catch’ blocks in internal code should be avoided and their use minimized** as they interfere with the reliable propagation of exceptions to the outermost handler.

**5.  “Recovering from exceptions” should be done rarely, if at all**

Your system has suffered a runtime exception.

Most code shouldn’t try and recover from exceptions. If you can’t prove that business outputs will be 100% correct, it’s generally best just to fail.

There are a couple of situations where exceptions can correctly be caught and handled.

**1.  Parsing** – when a missing/ invalid input can be ignored.  
**2.  “Business contingency”** — exceptions indicating a business outcome other than success; such as InsufficientFundsException. These should result in a different process/ execution flow.

High-reliability software may also attempt to *skip* or *retry* failed subsystems, in some select situations:

**3.  Optional subsystems** — eg. a printer, which might not be required right now to complete the business process.  
**4.  Retry unreliable subsystems** — retry may be used for unreliable systems/ connections. It is also common when obtaining locks against concurrent (multi-user) databases.

Retry & fault-recovery code is complex, and creates the possibility of pathological interactions (flooding, loops, log flooding, overloading a failing subsystem). This creates it’s own potential for error — I have seen Oracle crash due to faults in “recovery” code.

Generally, recovery or retry code should be used only where there is a definite business need.

**6.  Containers, but not threads, provide the outermost exception handler for you**

Because the requirement to report errors & log stacktraces is so common and basic, most EJB/ web containers & frameworks (Spring, Struts, Swing/ AWT) provide a standard error-handler for you.

These typically return a basic error response & log the exception. Some allow customization, to show a nicer looking error-page.

However, Threads and threaded code (SwingWorker, ExecutorPool etc) do not have nice exception-handlers builtin. Uncaught exceptions thrown in these will typically result in silent failure, which is time-consuming to debug.

For standard web applications & Swing, you can rely on the standard exception-handlers without needing to write any code. **If you’re using threads, however, make sure your exceptions will be handled.**

**7.  Prefer runtime to checked exceptions**

Modern Java use and other languages (C#) have moved away from the “checked exception” concept, to strongly prefer runtime (unchecked) exceptions.

Checked exceptions were originally intended for “contingency outcomes” — predictable business results, other than success. The classical example is InsufficientFundsException when attempting to pay or transfer money.

These are useful for specific individual situations, for which an alternate business process & response may exist.

However, they were never conceived to include unpredictable low-level & infrastructure failures — which can occur anywhere in code, at any time, and for which no effective business alternates exist.

“Potential to fail anytime, anywhere” due to underlying causes is simply not meaningful to deal with in a specific way. Runtime exceptions were designed to be unchecked, to allow unpredictable failure to be thrown from anywhere & handled further out. This is much better suited & allows *generalized exception handling strategies* such as “fail & report”, “retry” or “skip optional step”.

The best approach to dealing with checked exceptions (not representing genuine alternates/ or business contingencies) is to **wrap and rethrow** using a suitable unchecked exception.

Often, it is most convenient to wrap the entire method body in a catch-block to rethrow checked exceptions.

**8.  Rethrow with a cause**

In most projects, the most common “exception handling” code is actually just to rethrow the exception. While the guidelines above should help you drastically reduce the number of catch-blocks you code, it’s crucial to code the remaining ones correctly.

First and most important, **always include the cause.** Constructing and throwing a wrapper exception must always provide full details of the underlying failure.

Logging should generally be left to the outermost handler. This may seem counter-intuitive — shouldn’t all exceptions be logged? — but if your outermost handlers are comprehensive & correct, all exceptions will be logged there. Avoid duplication.

In some methods, we may want to “catch and rethrow” exceptions to provide a more-informative message/ or additional diagnostic information.

**9.  Add informative messages**

While throwing runtime exceptions outward requires the minimal amount of code, we may sometimes want better information as to what action & data failures the program failed on.

The ideal, is for exception messages (and the program itself) to be self-diagnosing. Any developer should be able to read the exception message & immediately understand what failed.

Great informative reporting requires only a few steps:

1. Throw a type, appropriate to the underlying cause;
2. Message stating what failed;  in business English.
3. Include the primary keys for data being accessed.

Here’s an example.

|  |
| --- |
| public Customer loadCustomer (int id) {      try {          //... load the Customer        } catch (SQLException x) {          throw new AppSQLException("error loading customer: id="+id, x);      }  } |

**10. Use logging**

Last but not least, logging is your friend. Exceptions should be logged at the outermost handler, to capture a comprehensive record of failures.

Methods throwing exceptions can also log error/ or warning lines to enrich the log output & increase the available information. Since major facts (eg primary keys) should be in the exception message, this is normally used to provide more minor & contextual detail.

As well as recording exceptions, logging should also record your program’s activity & decisions to understand *why* an exception occurred. Good practice is to log major business requests, decisions, outcomes and actions — these provide the context to understand what your program is doing & why.

Several logging frameworks are available in Java:

* Log4J — long-standing & most popular
* Logback — successor to Log4J
* Slf4J — universal API; plugs into an underlying logging framework
* Apache Commons logging — older common API; less favored these days
* java.util.logging (JULI) — unpopular Java built-in logging

Any of the first three are great. Java util logging (JULI) is not recommended, due to it’s awkward API and configuration. Log4J, Logback and Slf4J all provide excellent APIs to log your program’s activity & exceptions.

**Conclusion**

The foundations of clean exception-handling, are to avoid redundant catch blocks & make throwing easy.

As we all know, catch blocks are prone to “swallowing exceptions”, silently continuing despite failure, rethrowing without the cause, and many other common or garden coding errors. Minimizing the need for these increases developer productivity & avoids whole classes of errors.

Best practice is to concentrate on **throwing & informative reporting,** while concentrating actual handling in just a few well-defined catch blocks at the outermost level. (In many cases the container provides these.)

In between, there may sometimes be a need to wrap & rethrow — especially with checked exceptions.

Declaring a simple exception hierarchy, makes it easy for developers to find & throw the *right* exception. Categorizing by *cause* & using systematic naming will help developers to quickly find these.

Exceptions in general have two crucial & overarching goals:

1. prevent incorrect/ erroneous business actions being taken,  and
2. diagnose errors for subsequent resolution.

Exception handling code should first and foremost ensure that  code fails fast, and does not “run on” to produce uninitialized or erroneous results after a failure.

Recovery code, such as default values, should be used thoughtfully; reliability code, such as retrying, brings its own potential failure modes. Such added complexity should only be introduced where a well-defined need exists.

Following these practices will help minimize the amount of code you need to write, eliminate boilerplate, and deliver more reliable, self-diagnosing software.

The goal is for exceptions & logging to identify where the application was, what it was attempting to do, and exactly what error occurred.

Achieving this will reduce the number of errors you need to see & help you diagnose them in seconds — bringing you to true exception-handling nirvana.

**Do you need cleaner, better exception-handling? Share your story here.**

**Checked exceptions: Java’s biggest mistake**

[May 25, 2014](http://literatejava.com/exceptions/checked-exceptions-javas-biggest-mistake/) [Tom](http://literatejava.com/author/tom/) [10 Comments](http://literatejava.com/exceptions/checked-exceptions-javas-biggest-mistake/#comments)

Checked exceptions have always been a controversial feature of the Java language.

Advocates claim they ensure checking & recovery from failures. Detractors say “catch” blocks can almost never recover from an exception, and are a frequent source of mistakes.

Meanwhile, Java 8 and lambdas are here. Are checked exceptions becoming obsolete in the Java world?

**The Intent of Checked Exceptions**

In the mid 90′s, James Gosling at Sun came up with a new language.

At the time, C++ programming required every single function return to be checked for error. He decided there had to be a better way, and built the concept of “exceptions” into Java.

The intent of **checked exceptions** was to locally flag, and force developers to handle, possible exceptions. Checked exceptions have to be declared on a method signature, or handled.

This was intended to encourage software reliability & resilience. There was an intent to “recover” from contingencies – predictable outcomes other than success, such as InsufficientFundsException on attempting a payment. There was less clarity, as to what “recovery” actually entailed.

**Runtime exceptions** were also included in Java. Since null pointers, data errors, and illegal states/ accesses could occur anywhere in code, these were made subtypes of RuntimeException.

Runtime exceptions can be thrown anywhere, without requiring to be declared, and are much more convenient. But would it be correct to use them instead?

**The Drawbacks**

The crucial point here, is that runtime & checked exceptions are functionally equivalent. There is no handling or recovery which checked exceptions can do, that runtime exceptions can’t.

The biggest argument against “checked” exceptions is that most exceptions can’t be fixed. The simple fact is, **we don’t own the code/ subsystem that broke.**We can’t see the implementation, we’re not responsible for it, and can’t fix it.

Particularly problematic were the areas of JDBC (SQLException) and RMI for EJB (RemoteException). Rather than identifying fixable contingencies as per the original “checked exception” concept, these forced pervasive systemic reliability issues, not actually fixable, to be widely declared.

For any method, the possibility of failure includes all sub-methods called by it. Potential failures accumulate up the call tree. Declaring these on method signatures no longer offers a specific & local highlight for the developer to watch for – declared exceptions spread throughout the call tree.

Most EJB developers have experienced this – declared exceptions become required on methods through the tier, or entire codebase. Calling a method with different exceptions requires dozens of methods to be adjusted.

Many developers were told to catch low-level exceptions, and rethrow them again as higher (application-level) checked exceptions. This required vast numbers – 2000 per project, upwards – of non-functional “catch-throw” blocks.

Swallowing exceptions, concealing the cause, double logging, and returning ‘null’/ uninitialized data all became common. Most projects could count 600+ mis-coded or outright errors.

Eventually, developers rebelled against the vast numbers of “catch” blocks, and the source of error these had become.

**Checked Exceptions – incompatible with Functional Coding**

And then we get to Java 8, with its new ***functional programming***features – such as lambdas, Streams, and function composition.

These features are built on generics – parameter & return types are genericized, so that iteration & stream operations ( forEach, map, flatMap) can be written which perform a common operation, regardless of item type.

Unlike data types, however, declared exceptions can’t be genericized.

There is no possibility in Java to provide a stream operation (like, for example,  Stream.map) which takes a lambda declaring some checked exception, & transparently passes that same checked exception to surrounding code.

This has always been a major points against checked exceptions – all intervening code, between a throw and the receiving “catch” block, is forced to be aware of exceptions.

The workaround, of “wrapping” it in a RuntimeException, conceals the original type of the exception – rendering the exception-specific “catch” blocks envisaged in the original concept useless.

Finally we can capture Java’s new philosophy in a nutshell, by noting that none of the new “functional interfaces” in Java 8 declare checked exceptions.

**Conclusion**

Exceptions in Java provided major benefits in reliability & error-handling over earlier languages. Java enabled reliable server & business software, in a way C/ C++ never could.

Checked exceptions were, in their original form, an attempt to handle *contingencies* rather than *failures*. The laudable goal was to highlight specific predictable points (unable to connect, file not found, etc) & ensure developers handled these.

What was never included in the original concept, was to force a vast range of systemic & unrecoverable failures to be declared. These *failures* were never correct to be declared as checked exceptions.

Failures are generally possible in code, and EJB, web & Swing/AWT containers already cater for this by providing an outermost “failed request” exception-handler. The most basic correct strategy is to rollback the transaction & return an error.

Runtime exceptions allow any exception-handling possible with checked exceptions, but avoid restrictive coding restraints. This simplifies coding & makes it easier to follow best practice of [throw early, catch late](http://wikijava.org/wiki/10_best_practices_with_Exceptions#Throw_early_catch_late) where exceptions are handled at the outermost/ highest possible level.

Leading Java frameworks and influences have now definitively moved away from checked exceptions. Spring, Hibernate and modern Java frameworks/ vendors use only runtime exceptions, and this convenience is a major factor in their popularity.

Personalities such Josh Bloch (Java Collections framework), Rod Johnson, Anders Hejlsberg (father of C#), Gavin King and Stephen Colebourn (JodaTime) have all come out against checked exceptions.

Now, in Java 8, lambdas are the fundamental step forward. These language features abstract the “flow of control” from functional operations within. As we’ve seen, this makes checked exceptions & the requirement to “declare or handle immediately” obsolete.

For developers, it is always important to pay attention to reliability & diagnose likely points of failure (contingencies) such as file open, database connection, etc. If we provide good error messages at this points, we will have created self-diagnosing software – a pinnacle of engineering achievement.

But we should do this with unchecked exceptions, and if we have to rethrow, should always use RuntimeException or an app-specific subclass.

As Stephen Colebourn says, if your projects are still using or advocating checked exceptions, your skills are 5-10 years out date. Java has moved on.

**How are you dealing with exceptions & reliability? Add your thoughts now.**

* Where it makes sense, use a pre-defined Java exception. For example, if your code has some sort of I/O Error, it is fine to throw an IOException.
* Only use exception hierarchies if you need to differentiate between the two exceptions in a try/catch block. A lot of times it is perfectly fine to have a single component throw a single exception type with different messages for different errors. If the user cannot really do anything to handle the error specially, use the same generic exception class. If the user is able to handle them differently, that is when you should use a hierarchy.
* For hierarchies, do not make all exceptions from different components inherit from a base exception. There is no real reason to do this. If the consumer wants to catch anything, they can simply catch Exception.
* https://d.adroll.com/cm/aol/out?advertisable=JLRSHR4HC5DRDIELOVVJDOhttps://d.adroll.com/cm/n/out?advertisable=JLRSHR4HC5DRDIELOVVJDO

<https://dzone.com/articles/implementing-custom-exceptions-in-java>

You should use checked exceptions for all exceptional events that you can anticipate and that a well-written application should be able to handle. A checked exception extends the Exception class. A method that throws a checked exception or that calls a method that specifies a checked exception needs to either specify or handle it.

Unchecked exceptions extend the RuntimeException. You should use them for internal errors that you can’t anticipate and that, most often, the application can’t recover from. Methods can but don’t need to handle or specify an unchecked exception

<https://examples.javacodegeeks.com/java-basics/exceptions/java-custom-exception-example/>

<https://stackify.com/common-mistakes-handling-java-exception/>

Super z kodem bledu.

Mistake 7: Add unnecessary exception transformations

As I explained earlier, it can be useful to wrap exceptions into custom ones as long as you set the original exception as its cause. But some architects overdo it and introduce a custom exception class for each architectural layer. So, they catch an exception in the persistence layer and wrap it into a MyPersistenceException. The business layer catches and wraps it in a MyBusinessException, and this continues until it reaches the API layer or gets handled.

It’s easy to see that these additional exception classes don’t provide any benefits. They just introduce additional layers that wrap the exception. And while it might be fun to wrap a present in a lot of colorful paper, it’s not a good approach in software development.

You should always ask yourself if the new exception class provides any additional information or other benefits.