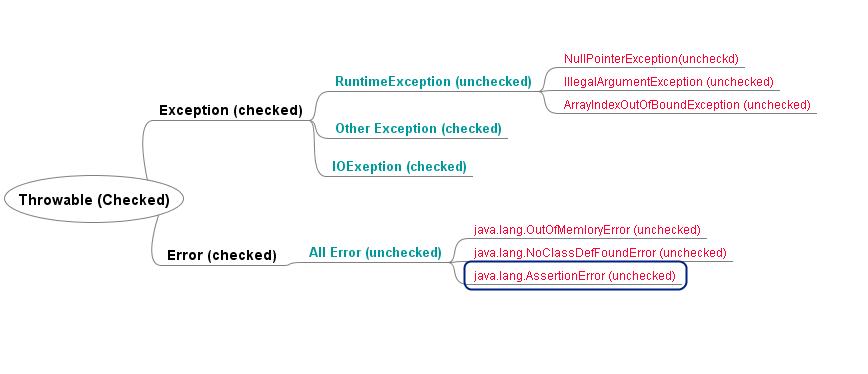
JAVA EXCEPTION HANDLING



3) **Checked Exception** is a reminder by compiler to programmer to handle failure scenario.

### Example of checked Exception in Java API Following are some Examples of Checked Exception in Java library:

IOException

[SQLException](http://javarevisited.blogspot.com/2012/01/javasqlsqlexception-invalid-column.html)

DataAccessException

[ClassNotFoundException](http://javarevisited.blogspot.com/2011/08/classnotfoundexception-in-java-example.html)

InvocationTargetException

FileNotFoundException

<https://wikis.oracle.com/pages/viewpage.action?pageId=30146595>

[**Why do we need checked exception in Java?**](https://wikis.oracle.com/pages/viewpage.action?pageId=30146595)

[**Blog Posts: October 2010**](https://wikis.oracle.com/display/sunidmdev/2010/10/)

* Why do we need checked exception in Java?

[<< June 2010](https://wikis.oracle.com/display/sunidmdev/2010/06/)

* Added by [shivjay](https://wikis.oracle.com/display/~shivjay), last edited by [shivjay](https://wikis.oracle.com/display/~shivjay) on Oct 27, 2010

Checked exceptions are as much part of your method signature as the name of the method and its parameters so the exception is part of the contract that you have with the client code. They serve as a marker to the client code that you have come across a error situation that you do not know how to handle or that you want the client code to know about so that it can handle the problem for you.  
As an example, your method provides a generic file write operation (think java.io.Writer or java.io.FileWriter) and it turns out that while you are writing to the file, the disk is full. Two options, you can either gracefully ignore that and pretend that you have written all the bytes to the file or you can report the problem to the client code and throw e.g. java.io.IOException. As per Johan, I can tell you that no client code is going to like option one, especially if the client code is a text processor and the user was saving this graduation thesis ;-)

Developers are lazy and this is why too often exceptions that are thrown by third party components are simply added to the throws clause of the calling method. The developer that does this, most likely did not think hard enough about the consequences as it changes the contract of his method. Take Johan's example where he had to implement a network implementation on top of a serial communication layer. All the errors that are thrown by the serial communication layer are relevant and are situation where the network layer will most likely not be able to recover from and hence will need to be reported to the client code but... adding them as such to the method signature reveals to the client code that are using serial comms as an implementation technique. **In other words, you have exposed parts of your implementation and when later on, you want to change that implementation you will not be able to remove that serial communication exception from your method signature if you want to stay 100% backwards compatible.**What you want to do in this situation is catch **the serial communication exceptions and convert them (wrap them so to speak) into your own exception type**

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

* **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
* **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
* **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

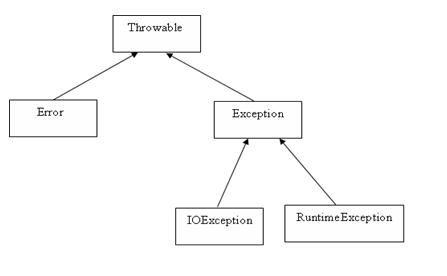
## Exception Hierarchy:

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses: **IOException** class and **RuntimeException** Class.



Exceptions Methods:

Following is the list of important medthods available in the Throwable class.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor. |
| 2 | **public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object. |
| 3 | **public String toString()** Returns the name of the class concatenated with the result of getMessage() |
| 4 | **public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream. |
| 5 | **public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack. |
| 6 | **public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace. |

Catching Exceptions:

## The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

## The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

Note the following:

* A catch clause cannot exist without a try statement.
* It is not compulsory to have finally clauses when ever a try/catch block is present.
* The try block cannot be present without either catch clause or finally clause.
* Any code cannot be present in between the try, catch, finally blocks.

## Declaring you own Exception:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

* **All exceptions must be a child of Throwable.**
* **If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.**
* **If you want to write a runtime exception, you need to extend the RuntimeException class.**

# [When to choose checked and unchecked exceptions](http://stackoverflow.com/questions/27578/when-to-choose-checked-and-unchecked-exceptions) (IMP)

Checked Exceptions are great, so long as you understand when they should be used. The Java core API fails to follow these rules for IOException and SQLException which is why they are so terrible.

**Checked Exceptions** should be used to declare for **expected**, but **unpreventable** errors that are **reasonable to recover** from.

**Unchecked Exceptions** should be used for everything else.

I'll break this down for you, because most people misunderstand what this means.

1. **Expected but unpreventable**: The caller did everything within their power to validate the input parameters, but some condition outside their control has caused the operation to fail. For example, you try reading a file but someone deletes it between the time you check if it exists and the time the read operation begins. By declaring a checked exception, you are telling the caller to anticipate this failure.
2. **Reasonable to handle**: There is no point telling callers to anticipate exceptions that they cannot recover from. If a user attempts to read from an non-existing file, the caller can prompt him for a new filename. On the other hand, if the method fails due to a programming bug (invalid input parameters or buggy method implementation) there is nothing the application can do to fix the problem in mid-execution. The best it can do is log the problem and wait for the developer to fix it at a later time.

Unless the exception you are throwing meets **all** of the above conditions it should use an Unchecked Exception.

**Reevaluate at every level**: Sometimes the method catching the checked exception isn't the right place to handle the error. In that case, consider what is reasonable for your own callers. If the exception is expected, unpreventable and reasonable for them to recover then you should throw a checked exception yourself. If not, you should wrap the exception in an unchecked exception. If you follow this rule you will find yourself converting checked exceptions to unchecked exceptions and vice versa depending on what layer you are in.

For both checked and unchecked exceptions, **use the right abstraction level**. For example, a code repository with two different implementations (database and filesystem) should avoid exposing implementation-specific details by throwing SQLException or IOException. Instead, it should wrap the exception in an abstraction that spans all implementations (e.g. RepositoryException).

FEW CHECKED AND UNCHECKED EXCEPTIONS   
  
**Unchecked** (RUN TIME)  
[ArrayIndexOutOfBoundsException](http://docs.oracle.com/javase/7/docs/api/java/lang/ArrayIndexOutOfBoundsException.html)   
[ClassCastException](http://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html)   
[IllegalArgumentException](http://docs.oracle.com/javase/7/docs/api/java/lang/IllegalArgumentException.html)   
[IllegalStateException](http://www.coderanch.com/how-to/java/IllegalStateException)   
[**NullPointerException**](http://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html)  
[NumberFormatException](http://docs.oracle.com/javase/7/docs/api/java/lang/NumberFormatException.html)   
[AssertionError](http://docs.oracle.com/javase/7/docs/api/java/lang/AssertionError.html)   
[ExceptionInInitializerError](http://docs.oracle.com/javase/7/docs/api/java/lang/ExceptionInInitializerError.html)   
[StackOverflowError](http://docs.oracle.com/javase/7/docs/api/java/lang/StackOverflowError.html)   
[**NoClassDefFoundError**](http://docs.oracle.com/javase/7/docs/api/java/lang/NoClassDefFoundError.html)  
  
**Checked**  (Compile time)  
Exception   
[IOException](http://docs.oracle.com/javase/7/docs/api/java/io/IOException.html)   
[FileNotFoundException](http://docs.oracle.com/javase/7/docs/api/java/io/FileNotFoundException.html)   
[ParseException](http://docs.oracle.com/javase/7/docs/api/java/text/ParseException.html)   
[ClassNotFoundException](http://docs.oracle.com/javase/7/docs/api/java/lang/ClassNotFoundException.html)   
[CloneNotSupportedException](http://docs.oracle.com/javase/7/docs/api/java/lang/CloneNotSupportedException.html)   
[InstantiationException](http://docs.oracle.com/javase/7/docs/api/java/lang/InstantiationException.html)   
[InterruptedException](http://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html)   
[NoSuchMethodException](http://docs.oracle.com/javase/7/docs/api/java/lang/NoSuchMethodException.html)   
[NoSuchFieldException](http://docs.oracle.com/javase/7/docs/api/java/lang/NoSuchFieldException.html)

SQLException

**If a method throws NullPointerException in super class, can we override it with a method which throws RuntimeException?**

One more tricky Java questions from overloading and overriding concept. Answer is you can very well throw super class of RuntimeException in overridden method but you cannot do same if its checked Exception.

<http://javahungry.blogspot.com/2013/06/top-25-most-frequently-asked-core-java.html>

### Can subclass overriding method declare an exception if parent class method doesn't throw an exception ?

Yes but only unchecked exception not checked.

Even better, if you think the client code is not going to take any action other than logging if the username is already taken, throw a unchecked exception:

throw new RuntimeException("Username already taken");

Alternatively, you can even provide a method that checks if the username is already taken.

It is worth repeating that checked exceptions are to be used in situations where the client API can take some productive action based on the information in the exception. *Prefer unchecked exceptions for all programmatic errors*. They make your code more readable

# [What is the difference between NoClassDefFoundError and ClassNotFoundException?](http://stackoverflow.com/questions/1457863/what-is-the-difference-between-noclassdeffounderror-and-classnotfoundexception)

|  |  |
| --- | --- |
| down voteaccepted | The difference from the Java API Specifications is as follows.  For [ClassNotFoundException](http://java.sun.com/javase/6/docs/api/java/lang/ClassNotFoundException.html):  ClassNotFoundException is Thrown when an application tries to load in a class through its string name using:   * The forName method in class Class. * The findSystemClass method in class ClassLoader. * The loadClass method in class ClassLoader.   but no definition for the class with the specified name could be found.  For [NoClassDefFoundError](http://java.sun.com/javase/6/docs/api/java/lang/NoClassDefFoundError.html):  Thrown if the Java Virtual Machine or a ClassLoader instance tries to load in the definition of a class (as part of a normal method call or as part of creating a new instance using the new expression) and no definition of the class could be found.  The searched-for class definition existed when the currently executing class was compiled, but the definition can no longer be found.  **So, it appears that the NoClassDefFoundError occurs when the source was successfully compiled, but at runtime, the required class files were not found. This may be something that can happen in the distribution or production of JAR files, where not all the required class files were included.**  As for ClassNotFoundException, it appears that it may stem from trying to make reflective calls to classes at runtime, but the classes the program is trying to call is does not exist.  The difference between the two is that one is an Error and the other is an Exception. With NoClassDefFoundError is an Error and it arises from the Java Virtual Machine having problems finding a class it expected to find. A program that was expected to work at compile-time can't run because of class files not being found, or is not the same as was produced or encountered at compile-time. This is a pretty critical error, as the program cannot be initiated by the JVM.  On the other hand, the ClassNotFoundException is an Exception, so it is somewhat expected, and is something that is recoverable. Using reflection is can be error-prone (as there is some expectations that things may not go as expected. There is no compile-time check to see that all the required classes exist, so any problems with finding the desired classes will appear at runtime. |

**java.lang.NoClassDefFoundError is thrown when a particular class referenced by your program is not available in the classpath.**