[**10 Exception handling Best Practices in Java Programming**](http://javarevisited.blogspot.com/2013/03/0-exception-handling-best-practices-in-Java-Programming.html)

Exception handling is an important part of writing robust Java application. It’s a non functional requirement for any application, to gracefully handle any erroneous condition like resource not available, invalid input, null input and so on. Java provides several exception handling features, in built in language itself in form of try, catch and [finally keyword](http://javarevisited.blogspot.com/2012/11/difference-between-final-finally-and-finalize-java.html).  Java  programming language also allows you to create new exceptions and throw them using  [throw and throws](http://javarevisited.blogspot.com/2012/02/difference-between-throw-and-throws-in.html) keyword. In reality, Exception handling is more than knowing syntax. Writing a robust code is an art more than science, and here we will discuss few Java best practices related to Exception handling. These [Java best practices](http://javarevisited.blogspot.com/2013/01/java-best-practices-method-overloading-constructor.html) are followed even in standard JDK libraries, and several open source code to better deal with Errors and Exceptions. This also comes as handy guide of writing robust code for Java programmers.

Exception Handling Java Best Practices

[Java best practices for Exception handling code](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)Here is my collection of 10 Java best practices to write Exception handling code in Java. There have been both applause and criticism of checked Exception in Java, which is a language feature to force dealing with Exceptions. In this article, we will look to minimize use of checked Exception and learn when to use checked vs unchecked exceptions in Java as well.

**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.  See [checked vs unchecked exceptions](http://javarevisited.blogspot.com/2011/12/checked-vs-unchecked-exception-in-java.html) for more information on choosing between checked and RuntimeException in Java.

**2) Close or release resource in finally block**

This is a well known best practice in Java and quite a standard, while dealing with networking and IO classes. Closing resources in finally block guarantees that precious and scarce resource released properly in case of normal and aborted execution, guaranteed by finally block. From Java 7, language has a more interesting [automatic resource management or ARM blocks](http://javarevisited.blogspot.com/2011/09/arm-automatic-resource-management-in.html), which can do this for you. Nevertheless, always remember to close resources in finally block, which is important to release limited resources like FileDescriptors, used in case of both socket and files.

**3) Including cause of Exception in stack-trace**

Many times Java library and open source code wraps one Exception into another, when one exception is thrown due to result of another exception. Its become extremely important to log or print cause of root exception. Java Exception class provides getCause() method to retrieve cause which can be used to provide more information about root cause of Exception. This Java best practice helps a lot while [debugging](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) or troubleshooting an issue. Always remember to pass original Exception, into constructor of new Exception, if you are wrapping one exception into another.

**4) Always provide meaning full message on Exception**

message of Exception is the most important place, where you can point out cause of problem because this is the first place every programmer looks upon. Always try to provide precise and factual information here. For example, compare these two Exception messages for IllegalArgumentException :

message 1: "Incorrect argument for method"

message 2: "Illegal value for ${argument}: ${value}

first one just says that argument is illegal or incorrect, but second one include both name of argument and its illegal value which is important to point out cause of error. Always follow this *Java best practice*, when writing code for handling exceptions and errors in Java.

**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**

One thing which is worth remembering is that Exceptions are costly, and can slow down your code. Suppose you have method which is reading from ResultSet and often throws SQLException than move to next element, will be much slower than normal code which doesn't throw that Exception. So minimizing catching unnecessary Exception and moving on, without fixing there root cause. 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. Avoid unnecessary Exception handling by fixing root cause.

**8) Avoid empty catch blocks**

Nothing is more worse than empty catch block, because it not just hides the Errors and Exception, but also may leave your [object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html) in unusable or corrupt state. Empty catch block only make sense, if you absolutely sure that Exception is not going to affect object state on any ways, but still its better to [log](http://javarevisited.blogspot.com/2011/05/top-10-tips-on-logging-in-java.html) any error comes during program execution. This is not a Java best practice, but a most common practice, while writing Exception handling code in Java.

**9) Use Standard Exceptions**

Our ninth Java best practice advise on using standard and inbuilt Java Exceptions. Using standard Exception instead of creating own Exception every now and then is much better in terms of maintenance and consistency. Reusing standard exception makes code more readable, because most of  Java developers are familiar with standard RuntimeException from JDK like, IllegalStateException, IllegalArgumentException or [NullPointerException](http://javarevisited.blogspot.com/2012/06/common-cause-of-javalangnullpointerexce.html), and they will immediately be able to know purpose of Exception, instead of looking out another place on code or docs to find out purpose of user defined Exceptions.

**10) Document Exception thrown by any method**

Java provides [throw and throws keyword](http://java67.blogspot.com/2012/10/difference-between-throw-vs-throws-in.html) to throw exception and in javadoc you have @throw to document possible Exception thrown by any method. This becomes increasingly important if you are writing API or public interface. With proper documentation of Exception thrown by any method you can potentially alert anyone who is using it.

That's all on Java best practices to follow while handing Exceptions in Java. Do let us know what practices you follow while  writing Exception handling code in Java.

Read more: <http://javarevisited.blogspot.com/2013/03/0-exception-handling-best-practices-in-Java-Programming.html#ixzz3KRPrnO00>

# [Checked vs Unchecked Exception in Java Example](http://javarevisited.blogspot.com/2011/12/checked-vs-unchecked-exception-in-java.html)

**Checked and Unchecked Exception** is two types of Exception exist in Java. Though there is no difference in functionality and you can very achieve same thing with either *checked Exception or Unchecked Exception*, there is some difference on exception handling part. In this Java tutorial we will see **what is checked and Unchecked Exception in Java**, Examples of Checked and Unchecked Exception and most importantly we will learn when to use Checked Exception and when to use Unchecked Exception in Java and lastly we will see **difference between checked and unchecked exception** to understand things better. By the way this article is second in my post on Exception along with d[ifference between throw and throws in Java](http://javarevisited.blogspot.com/2012/02/difference-between-throw-and-throws-in.html) and improved Exception handling in Java 7 with [Automatic resource management ARM](http://javarevisited.blogspot.sg/2011/09/arm-automatic-resource-management-in.html)and multi-catch block in JDK 7.

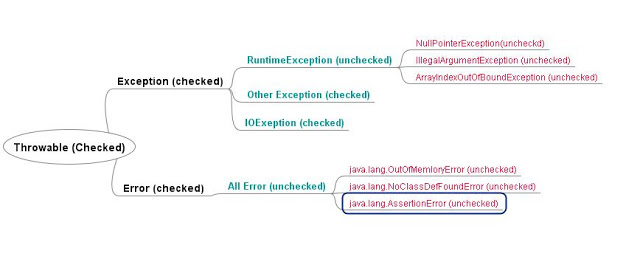
## Difference between Checked vs Unchecked Exception in Java

### What is Checked Exception in Java?

[Checked Exception vs Unchecked Exception in Java example](http://javarevisited.blogspot.com/2011/09/step-by-step-guide-to-convert-string-to.html)**Checked Exception in Java** is all those Exception which requires being catches and handled during [compile time](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html). If Compiler doesn’t see try or catch block handling a Checked Exception, it throws Compilation error. Now Which Exception is checked Exception and Why Checked Exception are introduced in first place? All the Exception which are direct sub Class of Exception but not inherit RuntimeException are Checked Exception.

While doing File Programming in C++ I found that most of the time programmer forgets to close file descriptors , which often result in locking of file on OS level. Since Java is introduced after C++, designers of Java thought to ensure such mistakes are not allowed and resources opened are closed properly. To ensure this they introduced Checked Exception. If you see most of [File IO related operation](http://javarevisited.blogspot.com/2011/12/read-and-write-text-file-java.html) comes under IOException which is checked one. Though is a special scenario related to Checked Exception but you can generalize this as, where Java sees an opportunity of failure more, they ensure that programmer provide recovery strategy or at least handle those scenario gracefully.

Since a picture is worth 1000 words I have put together Exception hierarchy in mind map which clearly says which Exceptions are checked and which Exceptions are unchecked.

[](http://4.bp.blogspot.com/-ocSXMYac3Eo/T9IGCdsbHQI/AAAAAAAAAYM/EjDyZa9ykik/s1600/checked+exception+vs+unchecked+exception+in+java.JPG)

### When to use Checked Exception in Java

Knowing **Checked Exception** is not that useful until you know how to use Checked Exception in Java. Java has often been criticized for its Checked Exception strategy, arguments given are that checked Exception adds lot of boiler plate code and makes whole class or function unreadable. Somewhat I agree with this and java also recognize this by introducing [improved Exception handling mechanism in Java7](http://javarevisited.blogspot.com/2011/07/jdk7-multi-cache-block-example-tutorial.html) but Checked Exception does have its real purpose. Following are some scenarios where I would prefer to use Checked Exception to ensure that [Code is Robust and stable](http://javarevisited.blogspot.com/2011/09/how-to-write-production-quality-code.html):

1) All Operation where chances of failure is more e.g. IO Operation, Database Access or Networking operation can be handled with Checked Exception.

2) When you know what to do (i.e. you have alternative) when an Exception occurs, may be as part of Business Process.

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

### What is Unchecked Exception in Java?

**Unchecked Exception in Java** is those Exceptions whose handling is not verified during Compile time. Unchecked Exceptions mostly arise due to programming errors like accessing method of a null object, accessing element outside an array bonding or invoking method with illegal arguments. In Java, Unchecked Exception is direct sub Class of RuntimeException. *What is major benefit of Unchecked Exception* is that it doesn't reduce code readability and keeps the client code clean.

### When to use UnCheckedException in Java

A good strategy of **Exception handling in Java** is wrapping a checked Exception into **UnCheckedException**. Since most of [Database operation](http://javarevisited.blogspot.com/2011/11/database-transaction-tutorial-example.html) throws SQLException but it’s not good to let SQLException propagate from your DAO layer to up higher on business layer and client code provide exception handling you can handle SQLException in DAO layer and you can wrap the cause in a RuntimeException to propagate through client code. Also as I said earlier unchecked exceptions are mostly programming errors and to catch them is real hard until you do a load test with all possible input and scenario.

### Difference between Checked and Unchecked Exception in Java

Now we have enough information to differentiate Checked Exception with Unchecked Exception:

1) Checked Exception is required to be handled by compile time while Unchecked Exception doesn't.

2) Checked Exception is direct sub-Class of Exception while Unchecked Exception are of RuntimeException.

3) CheckedException represent scenario with higher failure rate while UnCheckedException are mostly programming mistakes.

### Example of unchecked Exception in Java API

Here are few **examples of Unchecked Exception in Java** library:

NullPointerException

ArrayIndexOutOfBound

IllegalArgumentException

IllegalStateException

**Summary:**

1. Both Checked and Unchecked Exception are handled using keyword try, catch and finally.

2. In terms of Functionality Checked and Unchecked Exception are same.

3. Checked Exception handling verified during compile time.

4. Unchecked Exception are mostly programming errors

5. JDK7 provides improved Exception handling code with c[atching multiple Exception in one catch block](http://javarevisited.blogspot.sg/2011/07/jdk7-multi-cache-block-example-tutorial.html) and reduce amount of boiler plate code required for exception handling in Java.

Read more: <http://javarevisited.blogspot.com/2011/12/checked-vs-unchecked-exception-in-java.html#ixzz3KRQ8T3K9>

# [ARM- Automatic resource management in Java7 an example tutorial](http://javarevisited.blogspot.com/2011/09/arm-automatic-resource-management-in.html)

ARM **automatic resource management**is  another attractive features of Java 7 and project coin. Asname itself implies that now JVM is going to be handling all the external resource and make programmer free to bother about resource management.  If my java programmers use any external resources like file, printer or any devices to close after my program execution complete. Normally we close the resources which we have open in beginning of our program or we decide that if program finish normally how to manage the resource or if our program finish abnormally how to close the resource.

## ARM- Automatic resource management in Java

[automatic resource management (ARM) in java jdk7](http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html)Earlier this things or **resource management** are achieved **by try {} catch {} finally block** we need to declare resource variable outside of try block and in catch or finally we use this variables for achieving normal execution of program. This JDK7 tutorial is in continuation of my earlier tutorial [How to code with multi-cache exception in JDK7](http://javarevisited.blogspot.com/2011/07/jdk7-multi-cache-block-example-tutorial.html) and [How to use String in Switch case on JDK7](http://javarevisited.blogspot.com/2011/08/string-switch-case-jdk7-example.html)

### Example of resource management in java before JDK7

Here is an example of how we used to do handle *resource management*before [automatic resource management (ARM) feature](http://openjdk.java.net/projects/jdk7/features/) was made available.

FileInputStream stockQuoteReader= **null**;

      FileOutputStream stockQuoteWriter = **null**;

**try** {

        stockQuoteReader = **new** FileInputStream("StockQuotes.txt");

        stockQuoteWriter = **new** FileOutputStream("StockQuotes.txt");

**int** var;

**while** (var = stockQuoteReader.read()) != -1)

          stockQuoteWriter.write(var);

      } **finally** {

**if** (stockQuoteReader!= **null**)

          stockQuoteReader.close();

**if** (stockQuoteWriter!= **null**)

          stockQuoteWriter.close();

      }

But with java 1.7 we manage this thing very easily by**try with resource block**where inside try we mange this external resources.

### Signature of Automatic Resource Management (ARM)

**Signature** is **try(resource1;resource2){}**after final resource **;semicolon is not allowed**and the resource should be like **var=expression type** and bydefault all the resources are final  type.

### What has been added in API for Automatic Resource Management

**java.lang.AutoCloseable**, interface has been added in API which contains single method **close() throws Exception**this interface is a parent of **java.io.closeable** interface so all the input and output devices inherit this property.

### Example of Automatic Resource Management (ARM) in JDK7

Here is **example of automatic resource management with JDK 1.7** source base. Please make sure you run this with java source 1.7 otherwise you will get compilation error.

**try** (

FileInputStream stockQuoteReader = **new** FileInputStream("StockQuotes.txt");

FileOutputStream stockQuoteWriter = **new** FileOutputStream("StockQuotes.txt")

) {

**int** var;

**while**((var= stockQuoteReader.read()) != -1 )

            stockQuoteWriter.write();

  }  
  
  
In this code inside try we have declare two file stream one is input file we are reading from one file and writing to another file. After the whole process both streams will be closed automatically either the code has been executed normally or not that means **stockQuoteReader.close()** and stockQuoteWriter.close() called automatically which is the best part of ARM.  
If we compare this with earlier example   then if any exception occur during input file closing i.e. **stockQuoteReader.close()** , stockQuoteWriter.close() will never get executed so our code terminated abnormally.

### Some important points which needs to be keep in mind when use ARM

          Whatever resource we are using should be subtypes ofAutoCloseable other wise will get compile time error.

          The resources which we are using are closed in reverse order means stockQuoteWriter.close() will be called first then stockQuoteReader.close().

That’s all on **new automatic resource management (ARM) feature on JDK7,** some how it address the cluttering of code due to checked exception handling and code duplication on several exception cache block.

Read more: <http://javarevisited.blogspot.com/2011/09/arm-automatic-resource-management-in.html#ixzz3KRQCSvSK>

# [How to use Java 1. 7 Multiple Catch Block with example - JDK 7 tutorial](http://javarevisited.blogspot.com/2011/07/jdk7-multi-cache-block-example-tutorial.html)

As release of JDK 7 approaching General Availability (GA) on 2011/07/28, I thought to have a look on language enhancement as part of **project coin,** also called as **Small language enhancements or JSR 334**. Though there is not any major changes like [Enum](http://javarevisited.blogspot.com/2011/08/enum-in-java-example-tutorial.html) or [Generics](http://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html) of Java 1.5,  but they are still very useful, in terms of simplifying your day to day programming task. Some of the interesting changes are allowing [String in Switch cases](http://javarevisited.blogspot.com/2011/08/string-switch-case-jdk7-example.html),  , inclusion of fork join framework in JDK itself , , type inference using diamond operator,automatic resource management using  [try with resource](http://javarevisited.blogspot.sg/2011/09/arm-automatic-resource-management-in.html) feature, and ability to catch multiple Exception in single catch block . In this Java 7 tutorial, we will learn how **multi catch block of JDK 1.7** makes Exception handling code simpler and elegant. Multiple catch block will allow you to catch multiple exceptions in one block but it’s only available in JDK7 and you need to compiler your code with source 1.7. This article also shows you how to use JDK 7 multiple catch block with example. I also recommend book Java 7 Recipes: A Problem-Solution Approach to learn more about all the changes made in JDK 1.7 and how to make effective use of them. 

### JDK 1. 7 feature: Improved exception handling using multi-catch block

[jdk7 multi-cache block example tutorial](http://javarevisited.blogspot.com/2011/07/javalangunsupportedclassversionerror.html)Java has always been criticized for having [checked exception](http://javarevisited.blogspot.sg/2011/12/checked-vs-unchecked-exception-in-java.html) and polluting code with cluttered exception handling code, multi-catch block in Java 1.7  certainly assuage those wounds. With **multi catch block,** you can catch multiple exceptions in one catch block, which will eventually result in more readable code. Prior to **JDK 7** if you want to catch two exceptions, you need to provide two catch blocks and if you have same code to run on these two blocks, then either you need to use [finally block](http://javarevisited.blogspot.sg/2012/11/difference-between-final-finally-and-finalize-java.html) or just duplicate the code on two catch blocks. finally block is  not an ideal solution, because it will execute even if Exception is not thrown so ultimately lot of duplicate code which sometime makes code unreadable and clumsy. Now with JDK7 multi catch block we can catch multiple exception in one catch block separated by pipe (|) and reduce the code duplication. Let’s see an example of multiple exceptions catching in Java 7.

**public** **static** **void** main(String args[]) {

    Scanner scnr = **new** Scanner(System.in);

    String number = scnr.next();

**try** {

**if** (number.length() > 5) {

**throw** **new** IllegalArgumentException();

        }

        Integer.parseInt(number);

    } **catch** (**NumberFormatException | IllegalArgumentException e**) {

        e.printStackTrace();

    }

}

In above code example or ***JDK7 multi-catch block*** we have used multiple catch block of JDK 1.7 and control will come on this block whenever code throws either NumberFormatException or IllegalArgumentException.

### Java 7  multiple catch block example tutorial

We have seen code making use of this new Java 7 feature of catching more than one Exception in one catch block. In our example, we are catching NumberFormatException and IllegalArgumentException together and her we will verify that by entering input which will result in both type of Exception one by one. If we are able to catch both Exception than it's proven.

### Testing of JDK 1.7 multi-cache block

### If we will enter any number with alphabets, than it will throw NumberFormatException as shown below :

Input: 23ff

**java.lang.NumberFormatException: For input string: "23ff"**  
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)  
        at java.lang.Integer.parseInt(Integer.java:492)  
        at java.lang.Integer.parseInt(Integer.java:527)  
        **at jdk7demo.JDK7Demo.main(JDK7Demo.java:25)**

Now let's enter number with more than 5 digit this will result in IllegalArgumentException as per our code.

Input :123333  
**java.lang.IllegalArgumentException**  
        **at jdk7demo.JDK7Demo.main(JDK7Demo.java:23)**

I used Netbeans 7 to compile and run this project. **Setting up JDK 7 in Netbeans** is very easy just *download JDK7* and then click on **Tool-->Java Platform**and then click **"Add Platforms"** it will open a file browser just point out JDK7 installation directory and it will import JDK 1.7  binaries , source and docs and set it up for your use. One more thing you need to remember is that setting **source as 1.7** because this new language feature is only available in *JDK7*. In next series of this *JDK7 feature* article we will see [how to use String in Switch statement](http://javarevisited.blogspot.com/2011/08/string-switch-case-jdk7-example.html).  
  
  
**Recommended Book for further Reading**  
Java 7 Recipes: A Problem-Solution Approach By Josh Juneau, Carl Dea, Freddy Guime, John O'Conner

Read more: <http://javarevisited.blogspot.com/2011/07/jdk7-multi-cache-block-example-tutorial.html#ixzz3KRTAusnv>