**Checked Exceptions in Java**

In broad terms, a checked exception (also called a logical exception) in Java is something that has gone wrong in your code and is potentially recoverable. For example, if there’s a client error when calling another API, we could retry from that exception and see if the API is back up and running the second time. A checked exception is caught at compile time so if something throws a checked exception the compiler will enforce that you handle it.

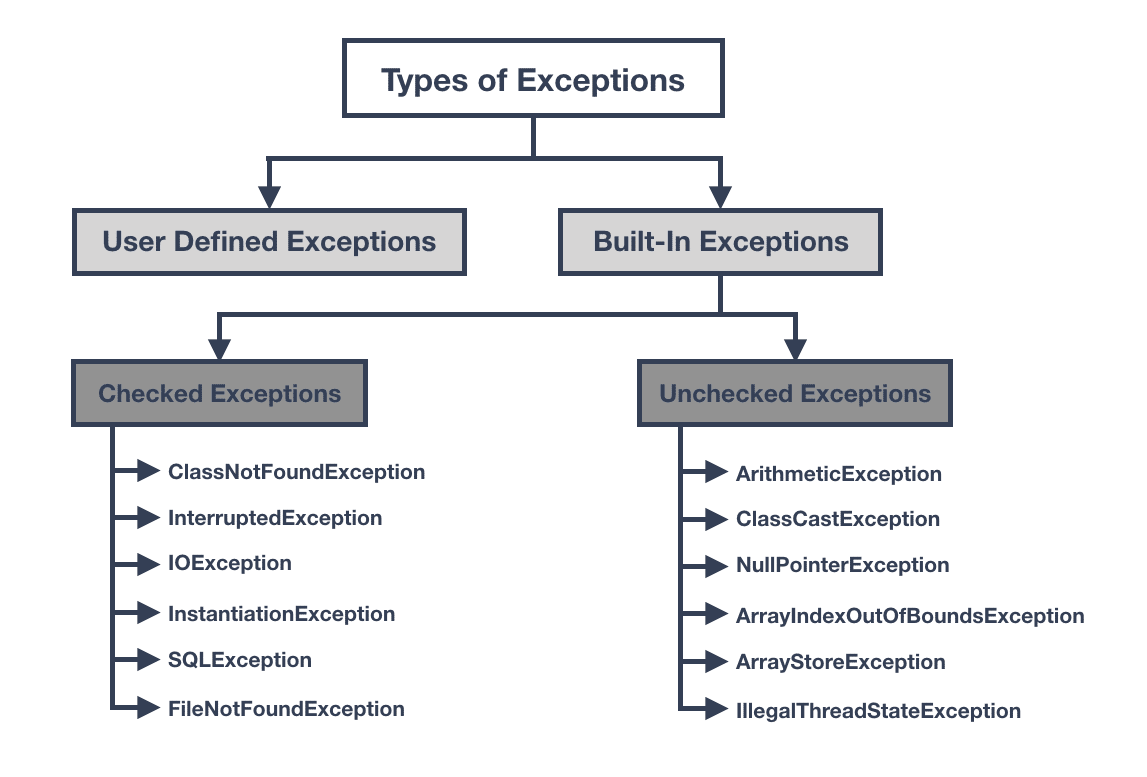


Fig1: Types of Exceptions in Java, Checked vs Unchecked

* ***Checked exception***
  + Must be handled by either the try-catch mechanism or the throws-declaration mechanism.
* **Runtime exception**
  + The right time to deal with runtime exceptions is when you’re designing, developing, and debugging your code. Since runtime exceptions should never be thrown in finished code.

**Checked Exception Examples**

The code below shows the FileInputStream method from the java.io package with a red line underneath. The red line is because this method throws a checked exception and the compiler is forcing us to handle it. You can do this in one of two ways.

import java.io.File;

import java.io.FileInputStream;

public class CheckedException {

public void readFile() {

String fileName = "file does not exist";

File file = new File(fileName);

FileInputStream stream = new FileInputStream(file);

}

}

**Try Catch**

You simply wrap the Java code which throws the checked exception within a try catch block. This now allows you to process and deal with the exception. With this approach it's very easy to swallow the exception and then carry on like nothing happened. Later in the code when what the method was doing is required you may find yourself with our good friend the NullPointerException.

We have now caught the exception and processed the error in a meaningful way by adding our code to the catch block, the code sequence carries on crisis averted.

import java.io.File;

import java.io.FileInputStream; import java.io.FileNotFoundException;

public class CheckedException {

public void readFile() {

String fileName = "file does not exist";

File file = new File(fileName);

try {

FileInputStream stream = new FileInputStream(file);

} catch (FileNotFoundException e) {

e.printStackTrace();

}

}

}

**Throws**

We use the keyword throws to throw the checked exception up the stack to the calling method to handle. This is what FileInputStream has just done to you. This looks and feels great - no messy exception code we are writing and we no longer need to handle this exception as someone else can deal with it. The calling method then needs to do something with it ... maybe throw again.

As with try catch be wary of always throwing as you need to think who SHOULD be handling the error and what piece of code is best placed to handle it correctly.

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

public class CheckedException {

public void readFile() throws FileNotFoundException {

String fileName = "file does not exist";

File file = new File(fileName);

FileInputStream stream = new FileInputStream(file);

}

}

**Unchecked Exceptions in Java**

An unchecked exception (also known as an runtime exception) in Java is something that has gone wrong with the program and is unrecoverable. Just because this is not a compile time exception, meaning you do not need to handle it, that does not mean you don’t need to be concerned about it.

The most common Java unchecked exception is the good old [**NullPointerException**](https://rollbar.com/blog/how-to-catch-and-fix-nullpointerexception-in-java/) which is when you are trying to access a variable or object that doesn’t exist.

So to summarize; the difference between a checked and unchecked exception is that a checked exception is caught at compile time whereas a runtime or unchecked exception is, as it states, at runtime. A checked exception must be handled either by re-throwing or with a try catch block, a runtime isn’t required to be handled. An unchecked exception is a programming error and are fatal, whereas a checked exception is an exception condition within your codes logic and can be recovered or retried from.

**Unchecked Exception Examples**

**BindException**

Because we live in a world where systems are built from lots of small micro services doing their own thing all talking to each other, generally over HTTP, this exception is popping up more and more. There isn’t a lot you can do about it other than find a free port. Only one system can use a single port at any one time and it’s on a first come, first serve basis. Most web applications default to port 8080 so the easiest option is to pick another one.

**IndexOutOfBoundsException**

This is a very common Java unchecked exception when dealing with arrays. This is telling you; you have tried to access an index in an array that does not exist. If an array has 10 items and you ask for item 11 you will get this exception for your efforts.

import java.util.ArrayList;

import java.util.List;

public class IndexOutOfBounds {

public static void main(String[] args) {

List<String> lst = new ArrayList<>();

lst.add("item-1");

lst.add("item-2");

lst.add("item-3");

var result = lst.get(lst.size());

}

}

The above piece of Java code is a common way to get an IndexOutOfBoundsException. The reason this trips people up is because the size of the array is 3 - makes sense; there are 3 items - but arrays are 0-based so the last item in the array is at index 2. To access the last item, it is always the size -1.

var result = lst.get(lst.size()-1);

**Checked Exceptions During Runtime**

Below is an example that is very commonly used in micro service architecture. If we received a request and we cannot, say, read data from our database needed for this request, the database will throw us a checked exception, maybe an SQLException or something similar. Because this data is important, we cannot fulfil this request without it.

This means there is nothing we can actually do with this exception that can fix the problem, but if we do nothing the code will carry on its execution regardless.

We could throw the exception to the calling code until we get to the top of the chain and return the exception to the user. By doing that we are then littering all the layers above with an exception that they really do not care about, nor should they. What we really want is an unchecked exception to terminate this request gracefully.

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

public class CheckedException {

public void readFile() {

String fileName = "file does not exist";

File file = new File(fileName);

try {

FileInputStream stream = new FileInputStream(file);

} catch (FileNotFoundException e) {

throw new ProcessingException("Error opening file"); }

}

}

}

Above we have our same piece of Java code for handling the checked exception thrown from the FileInputStream method but this time we are throwing our own RuntimeException and because this exception isn’t checked at compile time, we don’t need to declare it.

public class ProcessingException extends RuntimeException {

public ProcessingException(String message) {

super(message);

}

}

Declaring your own exception type is as simple as extending the runtime exception class because as we have seen from the diagram at the top, RuntimeException is a subtype of Exception.

**Difference Between Checked and Unchecked Exceptions in Java**

To summarize, the difference between a checked and unchecked exception is:

* A checked exception is caught at compile time whereas a runtime or unchecked exception is, as it states, at runtime.
* A checked exception must be handled either by re-throwing or with a try catch block, whereas an unchecked isn’t required to be handled.
* A runtime exception is a programming error and is fatal whereas a checked exception is an exception condition within your code’s logic and can be recovered or re-tried from.