Best Practices of Exception Handling in Java

1. Close Resources using the Finally Block

It happens quite often that you use a resource in your try block, like a Scanner or an Input Stream, which you need to close afterwards. A common mistake in these situations is to close the resource at the end of the try block.

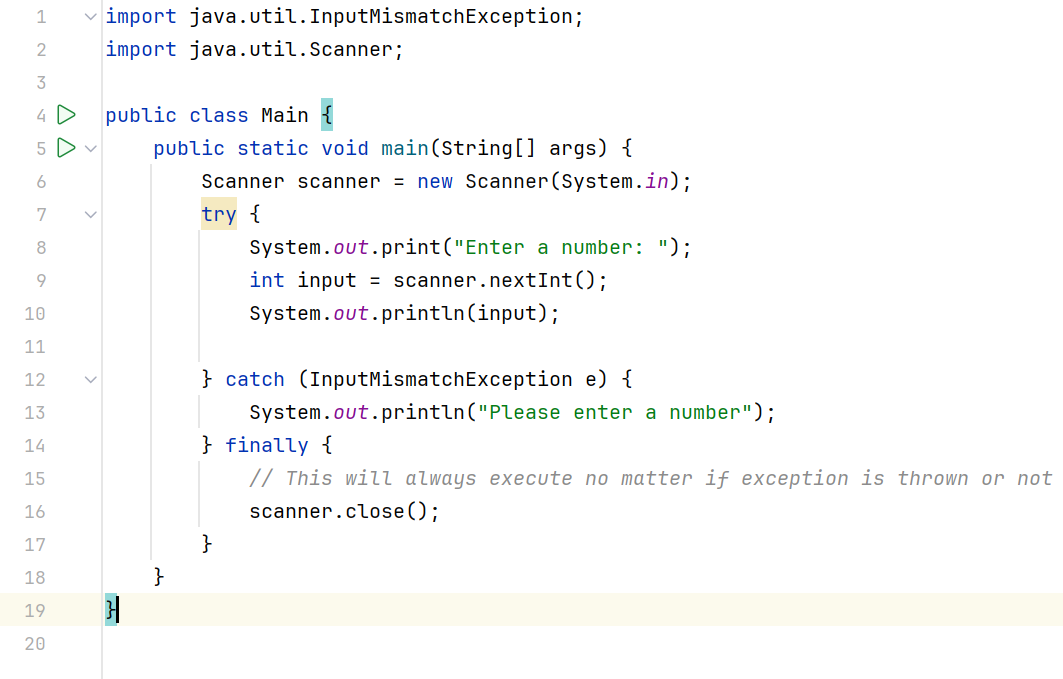
A screenshot of a computer code

Description automatically generated

The problem is that this approach seems to work perfectly if no exception gets thrown. All statements within the try block will get executed, and the resource gets closed.

But when an exception is thrown, it means you are not reaching the end of the try block. As a result, the resource will remain open.

Solution:

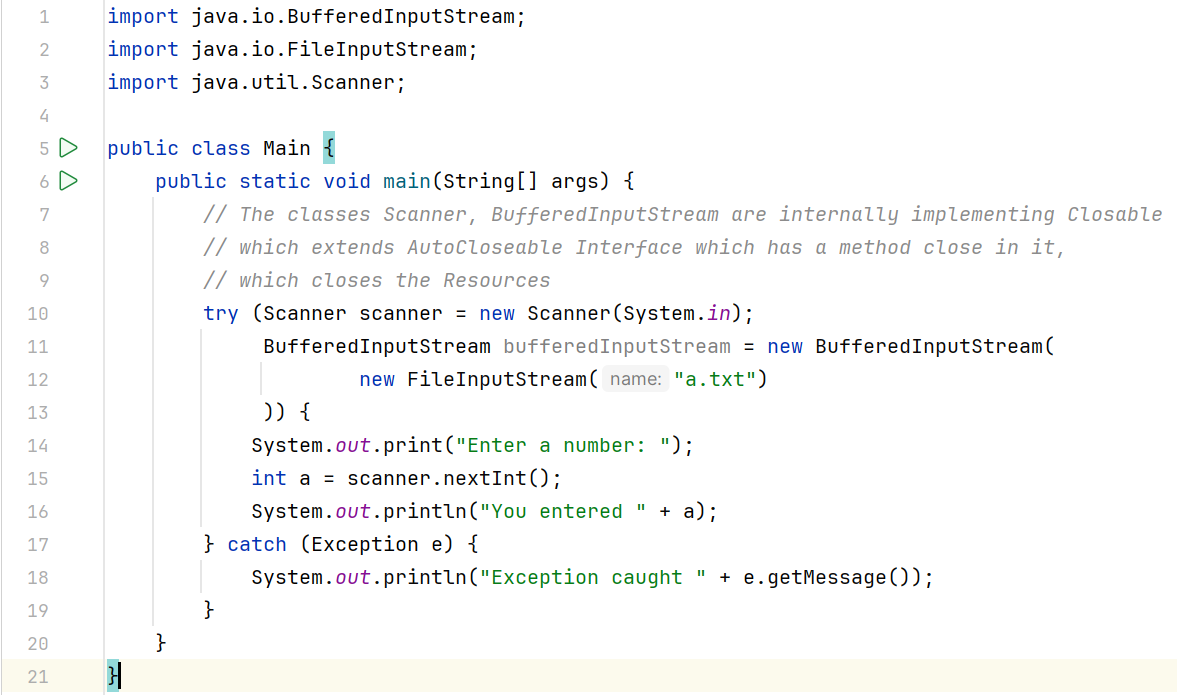
We can use ‘finally’ block to close the resource.

We can also use ‘try-with-resource’ block.

A screen shot of a computer code

Description automatically generated

AutoCloseable In Java

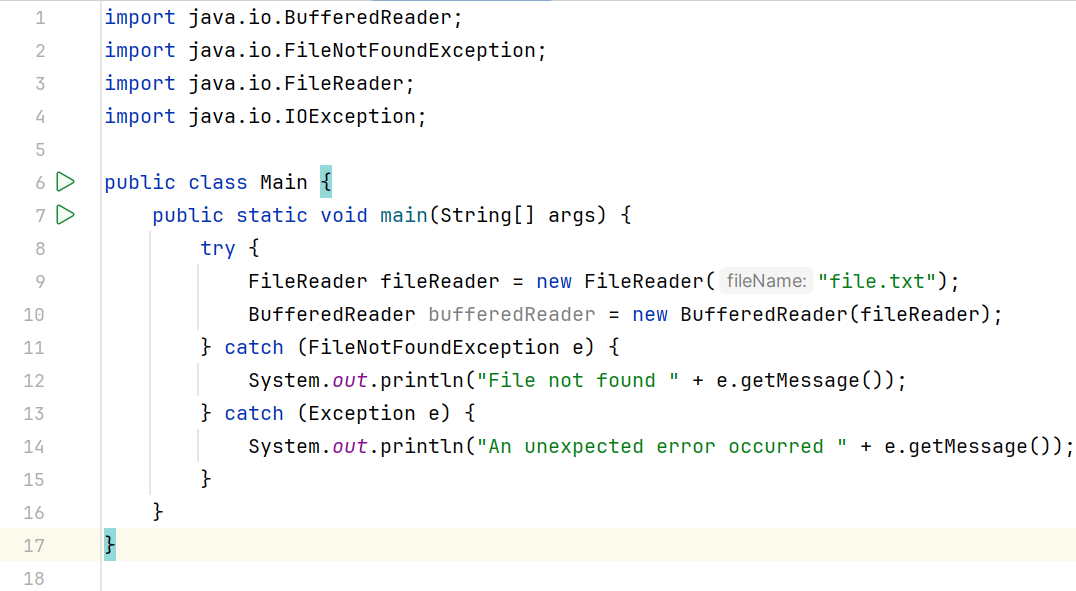


In this example, we will use a BufferedInputReader which internally implements the Closeable interface, which extends the AutoCloseable interface. The close() method of an AutoCloseable object is called automatically when exiting a try-with-resources block for which the object has been declared in the resource specification header hence the BufferedReader resource class is automatically closed by calling close() method of an AutoCloseable object.

1. Catch the most specific exception first.

The problem is that only the first catch block that matches the exception gets executed. So, if you catch an IllegalArgumentException first, you will never reach the catch block that should handle the more specific NumberFormatException because it’s a subclass of the IllegalArgumentException.

Always catch the most specific class first and add the less specific catch blocks to the end of your list.



We caught the most specific exception class FileNotFoundException first, and then we caught the Exception class to handle any unexpected errors that weren’t caught by the specific exceptions.

1. Don’t catch Throwable.

Throwable is the superclass of all exceptions and errors. If you use Throwable in a catch clause, it will not only catch all exceptions, but also catch all errors.

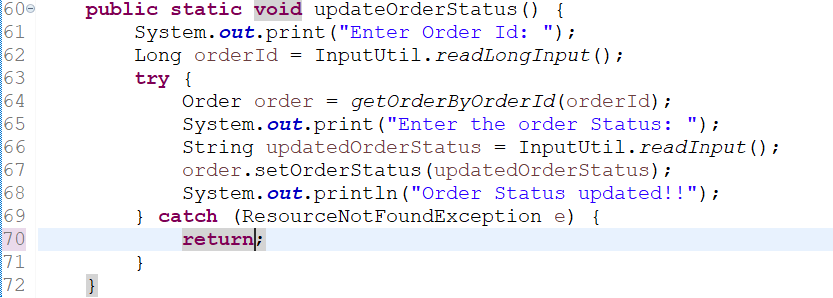
Errors are thrown by JVM to indicate serious problems that should not be handled by the application.

Examples: OutOfMemoryError and StackOverFlowError.

1. Don’t swallow Exceptions.

The practice of catching an error or exception, and then continuing without logging, processing, or reporting the error to other parts of the software.

Example:



Here we are catching exception but doing nothing. We are just returning from the function. This is NOT GOOD.

Solution:

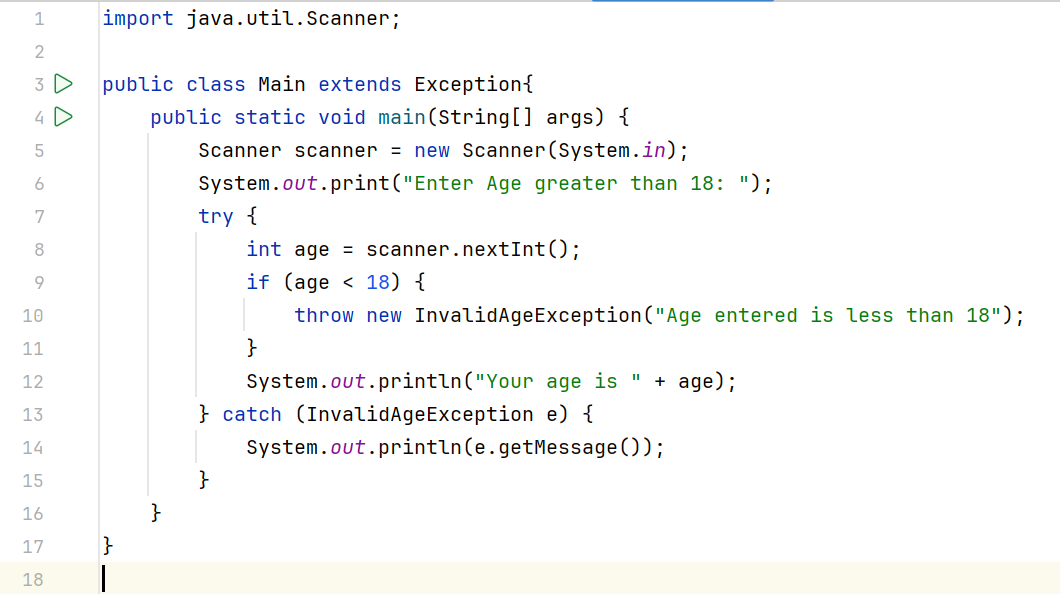
We do not have to use try-catch block unnecessarily. Rather we can make a check using the if statement and return the employee accordingly.

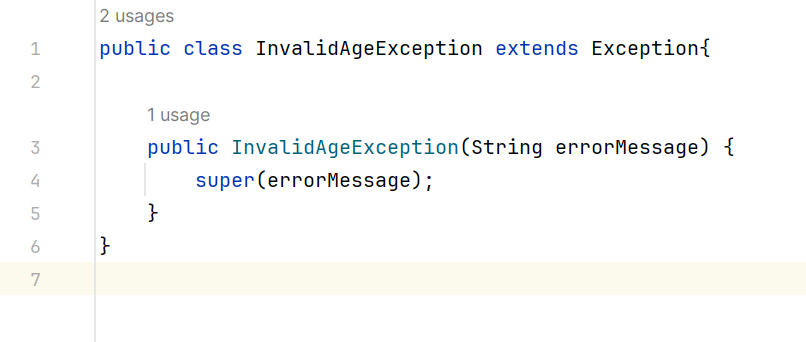
1. Use Custom Exceptions

Define custom exception classes when the standard Java exceptions do not adequately represent the error conditions in your application. This helps to provide more meaningful error messages and simplify exception handling logic.

To create custom exception, we need to extend Exception class that belongs to java.lang package.

Example: Creating custom exception of Invalid Age





This helps other developers to figure out what is the specific meaning of the exception thrown.

1. Keep Exception Handling Concise

Keep exception handling code concise and focused on error recovery or reporting. Avoid cluttering your code with excessive try-catch blocks or complex error handling logic if not necessary.

1. Throw Early, Catch Late

Throwing exceptions early and catching them late is a principle of software development that promotes better error handling and program stability.

By throwing exceptions as soon as an error is detected, we can get the right context for that exception. This allows developers to understand and handle the error at the right level of abstraction, so that the code’s flow remains clearer and focused.

Example: If we throw an exception at layer 1 (closest to a specific resource, that can raise exception) of our application, we can propagate exception at layer 2, adding more context and information to it and re-throwing it to layer 3, and finally handling the exception at layer 3.

A screenshot of a diagram

Description automatically generated

1. Avoid Empty Catch Blocks

Never leave catch blocks empty as they can obscure the root cause of an error. Always handle the exception or propagate it appropriately.

1. Throw Checked Exceptions When Appropriate

If a method encounters an error that it cannot handle, consider throwing a checked exception to notify the caller about the error. Document the checked exceptions thrown by your methods to provide clear guidance to callers.

1. Log and Handle Exceptions Consistently

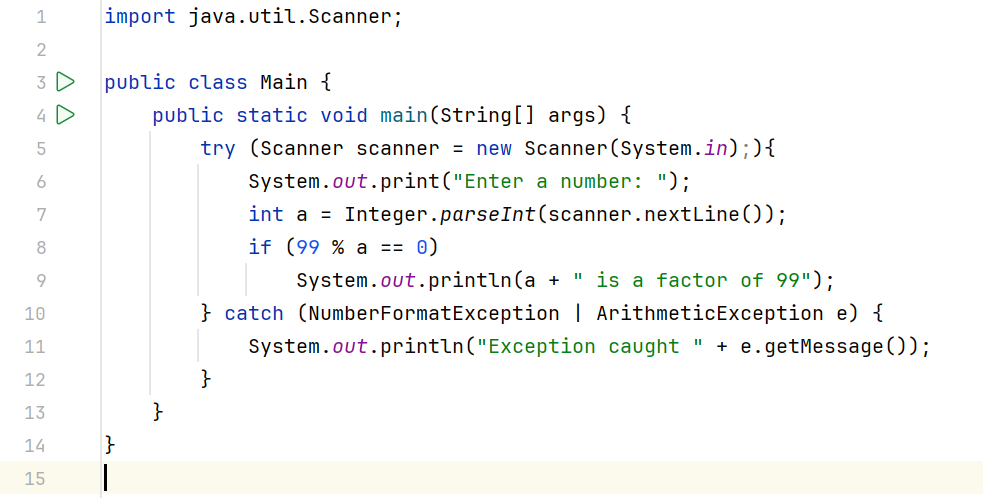
Provide detailed logs for exceptions to facilitate quick diagnosis and resolution. Ensure that the logs are consistent in format and detail. Choose an appropriate strategy for handling exceptions based on the error and your application requirements. This may include displaying an error message to the user, retrying the operation, or gracefully shutting down the application.

1. Use Multi-catch Statements

Before Java 7, we had to catch only one exception type in each catch block. So, whenever we needed to handle more than one specific exception but take some action for all exceptions, we had to have more than one catch block containing the same code.

Starting from Java 7.0, it is possible for a single catch block to catch multiple exceptions by separating each with | (pipe symbol) in the catch block. Catching multiple exceptions in a single catch block reduces code duplication and increases efficiency.

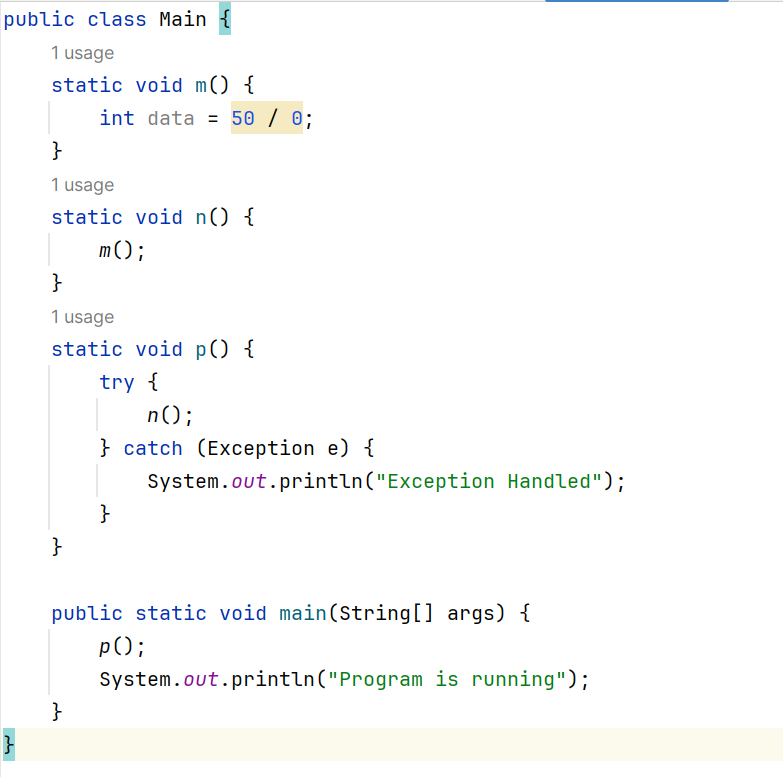
Example:



More about Exception Handling

1. Exception Propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

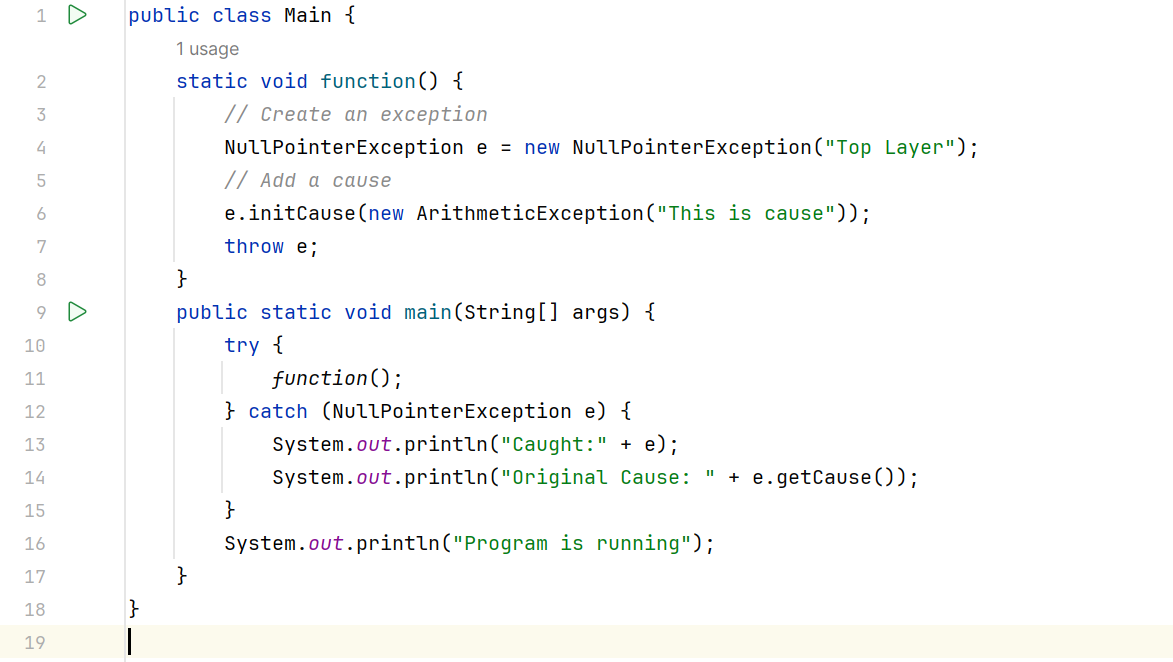


In the above example exception occurs in the m() method where it is not handled, so it is propagated to the previous n() method where it is not handled, again it is propagated to the p() method where exception is handled.

1. Chained Exceptions

Chained Exception helps to identify a situation in which one exception causes another Exception in an application.

For instance, consider a method which throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw the ArithmeticException to the caller. The caller would not know about the actual cause of an Exception. Chained Exception is used in such situations.

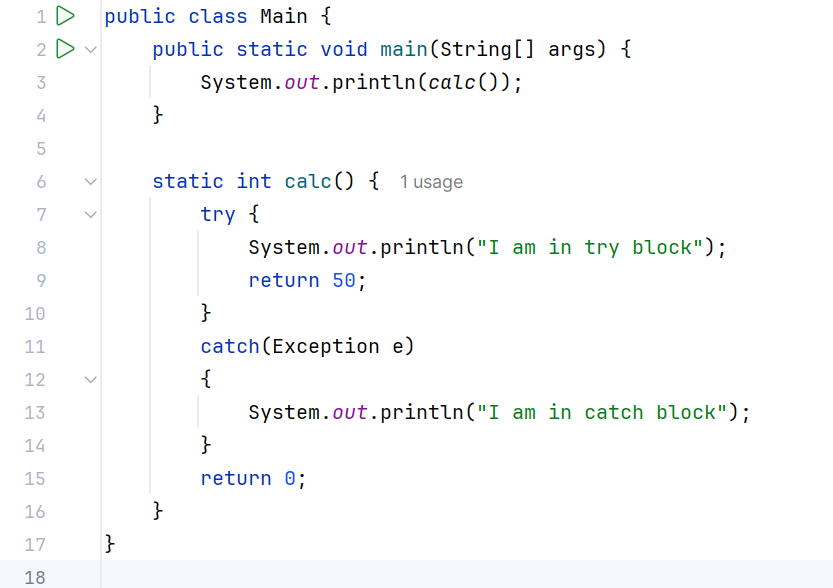


1. **Can we have a return statement in the catch or, finally blocks in Java?**

Yes, we can write a return statement of the method in catch and finally block.

* There is a situation where a method will have a return type and we can return some value at any part of the method based on the conditions.
* If we return a value in the catch block and we can return a value at the end of the method, the code will execute successfully.
* If we return a value in the catch block and we can write a statement at the end of the method after return a value, the code will not execute so it became unreachable code as we know Java does not support unreachable codes.
* If we return a value in the final block and no need of keeping a return value at the end of the method.

Example without Exception: This will print “I am in try block 50”.



Example with Exception: This will print “I am in try block 20” 