IOException

The IOException is a special kind of exception, called a **Checked Exception**.

It's the parent class of many common exceptions you'll encounter, when working with external resources.

Checked Exception

A checked exception represents an anticipated or common problem, that might occur.

You can imagine that a typo in the file name might be a common occurrence, and the system wouldn't be able to locate a file.

It's so common, that Java usually has a named exception just for that situation, the FileNotFoundException, which is a subclass of IOException.

A Checked Exception must be caught, or specified in the containing method's throws clause

How do you handle a checked exception?

You have two options.

You can wrap the statement that throws a checked exception, in a try catch block, and then handle the situation in the catch block.

Or, alternately, you can change the method signature, declaring a throws clause, and specifying this exception type.

LBYL or EAFP?

LBYL stands for, "Look Before You Leap". This style of coding involves checking for errors, before you perform an operation.

EAFP stands for, "Easier to Ask Forgiveness than Permission". This assumes an operation will usually succeed, and then handles any errors that occur, if they do occur.

Which is Better, LBYL or EAFP?

Like all questions about software, the answer is, that depends.

Feature	LBYL	EAFP
Approach	Check for errors before performing an operation.	Assume that the operation will succeed and handle any errors that occur.
Advantages	Can be more efficient if errors are rare.	Can be more concise and easier to read.
Disadvantages	Can be more verbose if errors are common.	Can be more difficult to debug if errors are unexpected.

How do you recognize a Checked Exception?

When you're coding with an IDE, it's pretty easy to recognize one, because your code won't compile, if one is thrown from code you're calling.

A checked exception means it's NOT an **unchecked exception**.

An **Unchecked Exception** is an instance of a **RuntimeException**, or one if its subclasses.

Let me just pull RuntimeException up, in the API documentation.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/RuntimeException.html

The finally clause

A `finally` clause is used in conjunction with a `try` statement.

A traditional `try` statement requires either a `catch` or a `finally` clause, or can include both.

The `finally` clause is always declared after the `catch` block if one is declared.

The `finally` block's code is always executed, regardless of what happens in the `try` or `catch` blocks.

The `finally` block does not have access to either the `try` or `catch` block's local variables.

What is the purpose of the finally clause?

The original purpose for the finally clause, was to have a single block of code to perform cleanup operations

This included closing open connections, releasing locks, or freeing up resources.

This clause ensured that this code was executed, both during normal completion as well as in the event of an exception.

The **try with resources** syntax, introduced in JDK7, **is a better approach** than using the finally clause for closing resources.

The finally clause can be used to execute other important tasks, such as logging or updating the user interface.

Disadvantages of using the finally clause

- It can be difficult to read and understand code, that uses this clause.
- It can be used to hide errors, which can make debugging more difficult.
- If you execute code that's not related to cleanup tasks, this will make it harder to maintain your code.