# **Exception Handling**

Before we get into anything new, let's review errors and exceptions together.

In Java, exception handling means taking care of ****runtime errors,**** so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException and IOException, which refer to missing classes and bad input/output respectively****.****

As we've seen, an ****exception**** is an unwanted or unexpected event which occurs during the execution of a program (i.e. at runtime), that disrupts the normal flow of the program’s instructions. In Java, when an exception occurs within a method, Java creates an object with information about what happened. Predictably, this object is called the exception object. It contains information like the name and description of the exception and the state of the program when the exception occurred.

Here's a list of major reasons why ****exceptions**** occur in a program:

* Invalid user input
* Device failure
* Loss of network connection
* Physical memory limitations (e.g., the user's computer has run out of space to complete the process)
* Coding errors
* Attempting to open an unavailable file

## **Errors vs Exceptions**

Unlike exceptions, ****errors**** represent irrecoverable conditions such as the Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. To clarify: exceptions are errors that we've anticipated and can write code to deal with. Errors, on the other hand, are usually beyond the control of the programmer. Therefore, we don't usually try to handle errors directly–we just write code that tries to keep them from happening.

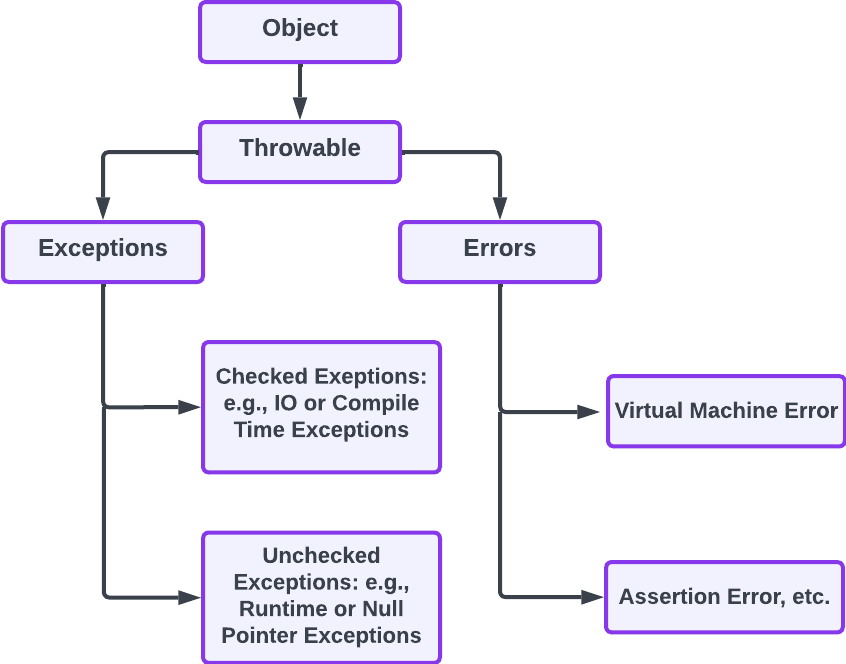
Here are the most important differences between Errors and Exceptions:

* ****Error****: An error indicates a serious problem that a reasonable application should not try to catch.
* ****Exception****: An exception indicates conditions that a reasonable application might try to catch and appropriately handle, so long as resources and access permit them to do so.

## **Exception Hierarchy**

All exception and error types are subclasses of the Throwable class. One branch of this hierarchy is headed by Exception. This class is used for exceptional conditions that our programs should catch. One example of the Exception class is the [NullPointerException](https://rollbar.com/blog/how-to-catch-and-fix-nullpointerexception-in-java/)

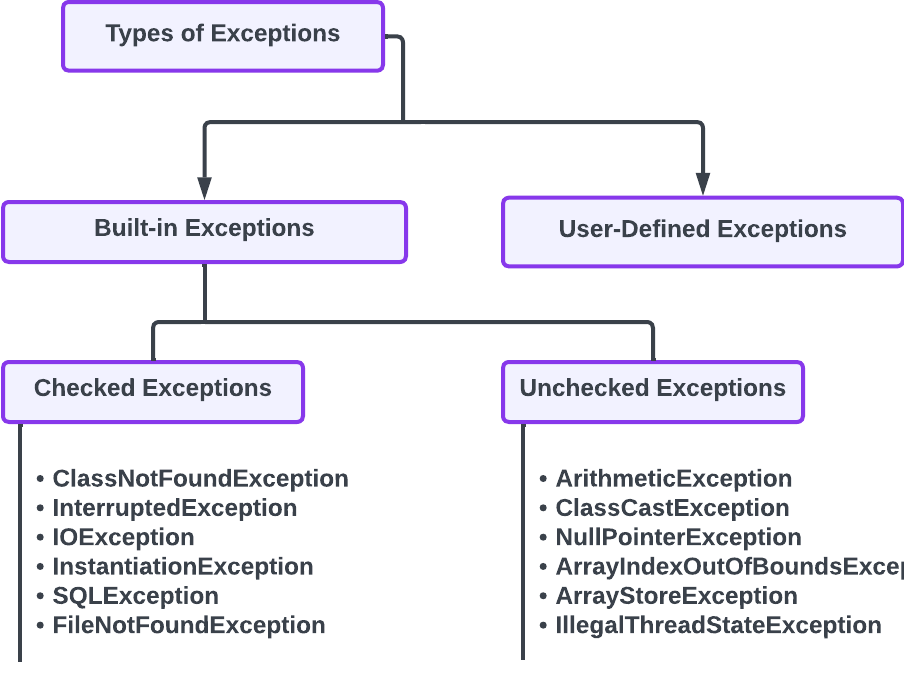
The other major branch of the Throwable class, Error, is used by the Java run-time system (aka, the Java Virtual Machine or JVM) to indicate errors having to do with the run-time environment itself (aka the Java Runtime Environment or JRE). [StackOverflowError](https://rollbar.com/blog/how-to-fix-java-lang-stackoverflowerror-in-java/) is an example of such an error:



*Stemming from the original Object class, these are the branches of the Throwable class*

## **Types of Exceptions**

Java defines several types of exceptions that relate to its various class libraries. As shown in the figure below:



*Java's built-in errors that branch off from the classes defined above*

The branch with no list in the image above is potentially the most expansive: we can define as many or as few of our own exceptions as we need to, and decide the error handling that goes along with them. We'll be going in-depth on defining our own exceptions in a lesson soon.

# Nesting trys

We've learned to use try and catch blocks, and we've learned to catch multiple different kinds of exceptions when a try block fails to achieve its goals–but what if we need to try different things depending on different circumstances? Java has us covered there, too.

## Java Nested try block

In Java, a try block that lives inside of another try block is called a **nested** try **block**. It works like this:

try {

// Code to try before the nested try block goes here.

// This is our nested try:

try {

// Nested code to try

} catch {

// Code to run if the nested try fails

}

} catch {

// Code to run if an exception occurs in the original try, but is not caught by the nested catch try/catch patterns.

}

Why Use Nested try Blocks?

The example above was a bit simplistic. After all, the outer try was fairly extraneous, given that we had nested trys to check the conditions that the outer one would have otherwise covered. However, a situation may arise where a part of a block may cause one exception and the entire block itself may cause another exception. In such cases, exception handlers should be nested so that each individual exception can be handled appropriately.

Annotated example of nested try block

Annotated example of nested try block

In the example image here, there are statements that belong to the initial try block itself. The evaluation of these statements would, therefore, trigger the outer catch exception if they were to fail. Similarly, if our inner try blocks here didn't have appropriate exception handling, the exceptions thrown by their attempts to run would be caught by the outer catch block. In this way, we have a couple of chances to catch these exceptions, but remember: our goal is to catch the exceptions we can anticipate. This is why we call them exceptions, and it's why we have try/catch blocks at all. We want to appropriately handle everything we can, and plan for the contingencies that matter most.

#Summary

If there are exceptions that can occur in the subprocesses of a larger try block, we should nest more try/catch blocks inside of the original try and handle their exceptions appropriately.

When a try block does not have a catch block for a particular exception, the outer parent's try block will be checked for that exception. If the exception occurs and the error matches what the catch was looking for, the outer catch's code will be run.

If you want a catch block to handle all general exceptions, remember to simply pass it the parameter Exception e. This will ensure that all exceptions in this block are caught. Note: you can rename e if you like, but it's common practice to just put e.

If none of your catch blocks has been created specifically to handle an exception and you have no default catch, the Java runtime system will handle the exception itself, and display its auto-generated message.

Inner catch blocks don't catch the outer block's exceptions. The outer block needs its own catch block, otherwise, the Java runtime system will have to handle it.

#

Custom Exceptions

As you saw if you clicked the full exception list link in a previous lesson, Java has a ton of built in exceptions that we can take advantage of. Some are automatically available, others need to be imported–but all together, they handle just about everything we need them to. But what happens when they don't?

That's when we write our own custom exceptions. Often, the reasons for writing our own exceptions are pretty straightforward: 1) we want our own exceptions to be more specific than the options Java has provided us, or, 2) the business logic of our application needs more clarity specific to our use cases than Java's built-in exceptions can cover. Whether we create them for these reasons or something else entirely doesn't really matter: we have to make them the same way, so let's get into it.

Making Exceptions

Custom exceptions always start by extending Java's built-in Exception class or any of Exception's subclasses. When we do this, we create what's called a custom exception or a user-defined exception. Our new exception will then have access to the superclasses from which it is extended, and offer us the opportunity to add in the extra information we need.

# **Writing SOLID Code**

Thus far, we've written quite a bit of segregated code. Each file has a simple class that runs its code and the results are typically handed off to another file for further use. However, most code–especially OOP code–doesn't work this way. Generally speaking, the whole point behind OOP is to leverage the power of objects, as we've discussed with the Four Pillars of OOP.

With great power comes great responsibility, though. Keeping our code easy-to-read, reusable, and easily modifiable can represent some challenges. To help us overcome these challenges, Robert C. Martin (aka "Uncle Bob") developed a list of principles that eventually became SOLID: Single responsibility, Open/closed, Liskov integration, Interface segregation, and Dependency inversion. Let's go through them one at a time.

## **#1: Single Responsibility Principle**

The Single Responsibility Principle states that "a class should have one, and only one, reason to change." Essentially, this means that each class should only have one job.

*Don't make these, photo by Patrick on Unsplash*

If you were creating a program to calculate the perimeters of various shapes, you could create a class called shapePerimeterCalculator. This class would likely take in a type of shape and other parameters, like side lengths for rectangles and radii for circles–and that would all be fine. The shapePerimeterCalculator's job, after all, is to calculate the perimeter.

But what if you wanted to output the perimeter after you found it? You may think, "easy! System.out.println at the bottom of the shapePerimeterCalculator!" In a sense, this is true–but in the real world, we rarely put anything straight to the console. In reality, you may need the results of that information in many different places and/or in many different formats.

In order to stick to the Single Responsibility Principle, you'd want to make a printer class to print your calculated perimeter. It may seem silly when you think of just one class using this new printer class, but what if you wanted to expand your calculator? Maybe you'll do some area calculations, or 3D volumes and complete surface areas? Will you add a new print statement to each of these, or would you rather pass the information off to a dedicated printer? You could use your printer for error messages and anything else you needed to output, too. Best of all, if something breaks with your print statements, you'll have exactly one place to go to fix it, rather than three or more.

*Make these, Photo by Elena Rouame on Unsplash*

Consider also that you may need to output your information in other formats, send it to a database, or display it on a screen. By having separate classes handle separate cases, you can keep your logic tightly controlled.

****The Single Responsibility Principle ensures that your code remains reusable and maintainable by keeping your classes focused on achieving one goal.****

## **#2: Open/Closed Principle**

Probably the most confusingly named of the SOLID Principles is the Open/Closed Principle. Let's take it in two parts:

1. Open: the open part of the Open/Closed Principle refers to the fact that a class should be open for extension. As you've seen throughout this week, extension means adding something on to a class, such as an attribute or a method. For instance, we may have a Food class that has an isEdible attribute, a calorieCount, etc., but we could add onto that class–extend it–with something like an isArt attribute. Most food won't be art–it'll just be food–but a class that's open to extension can be made into something more whenever we need it to be.

*An artistic wedding cake, photo by Deva Williamson on Unsplash*

1. Closed: the closed part of the Open/Closed Principle refers to modifying the original class. In the example above, we extended the class to make it something more than it was originally, but we didn't have to dig into the class itself and assign new attributes to the Food class. It's that simple.

To follow the Open/Closed Principle, you can use classes and interfaces to hand down the attributes and methods that all of your instances will need–and then extend those basics to take care of edge cases, like edible art.

****The Open/Closed Principle means keeping your classes/interfaces open for extension, but making them foolproof enough that you don't have to go in and edit the underlying class/interface itself.****

## **#3: Liskov Substitution Principle**

Named after [Barbara Liskov](https://en.wikipedia.org/wiki/Barbara_Liskov), this principle can be a bit hard to understand, especially when you read her original 1987 explanation:

*What is wanted here is something like the following substitution property: if for each object O1 of type S there is an object O2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T.*

What it means in practice is fairly simple: any class derived from another should fulfill the base class's behaviour. Essentially, if you're going to define a Bird class that has a method called fly(), later defining an emu, an ostrich, or a penguin, would cause you to break the Liskov Substitution Principle. The base class said that instances of this class would have a fly() method, but these birds can't fly. It really is that simple: the instances of a class must represent all of the features of the parent class.

****To follow the Liskov Substitution Principle, return to the Open/Closed Principle and think: what does the base class/interface need, and what should be an extension built upon that foundation?**** Keep the extensions separated, and you'll be fine.

## **#4: Interface Segregation Principle**

The Interface Segregation Principle builds upon all of the above: interfaces should not pass on any behaviour to the classes that extend them beyond what those classes will use. If, instead of a Bird class, we had a IBird interface, we shouldn't have it feature an abstract method called fly() any more than it should have an abstract method called swim(). While most birds do fly, and many can swim, it doesn't make much sense to ensure that all birds inherit these methods because some of them won't be using them.

In practice, this means writing more granular interfaces, rather than trying to capture everything in one big one. As the name suggests, segregate your needed functionality into separate interfaces, then have your classes implement as many of them as they need.

****The Interface Segregation Principle is exactly how it sounds: keep your interfaces separated, and ensure that all instances that inherit from them need the functionality that they provide.****

## **#5: Dependency Inversion Principle**

The Dependency Inversion Principle basically states that the code we write should depend on abstractions. In the words of Uncle Bob, "high level modules should not depend upon low level modules. Both should depend on abstractions," and "abstractions should not depend on details. Details should depend upon abstractions."

While it may be a little early in your dev career to be getting so abstract, this really should be your end goal: wherever possible, your code should rely on abstraction. Abstraction helps to keep your code loosely coupled, which basically means that no two parts of your code rely directly upon each other any more than they have to.

Suppose you had a class A that depended upon a class B, and that class B depended upon class C. If class C changes, say in the way that it outputs its response, class B would have to change. If class B has to change, there's a very high likelihood that class A would have to change, too. In companies that scale their products (like banks), this can be a major cause for concern.

However, if classes A, B, and C, all relied upon abstract methods from an interface or two, we could keep their interdependency to a minimum. If C changed, it wouldn't necessarily require that A or B be changed–and that's exactly what we want.

****The Dependency Inversion Principle tells us to rely on abstractions, rather than concrete examples. Where possible, we should leverage the power of interfaces to ensure that changes we make in one part of our code don't cause cascading troubles for the rest of our program.****