# Java: Handling exceptions

## Quick summary

- Try-catch blocks are ways of safeguarding your code from blowing up if there are errors.
- Java runs the code in the try block; if there's a problem, it runs the code in the catch block and keeps going through the routine. without catch blocks, the entire program would stop if it encountered an error. The catch block handles the error so your program can continue despite errors.
- `finally` is a catch block that runs regardless of the error
- Use catch blocks specific to the type of error you anticipate. There are 50+ type of errors that can be thrown.
- Using catch blocks focused on a specific error will help you identify problems with the code
- Run your more generic catch blocks last; this way your more specific catch blocks will identify the error more accurately.

```
Eclipse example: handling_exceptions

Eclipse example: handling_exceptions_2

Eclipse example: handling_exceptions3_compile_time

Eclipse example: handling_exceptions4
```

## Detailed description

Error handling is one of the most important concepts to consider in your code because it helps identify errors that are happening. When you set up your method, if you add a `throws` to your method, it will launch a specific type of error if that code fails. The code will stop running at that point and be handled by that type of exception.

Here I've identified that the `run` method will throw a `FileNotFoundException`.

```
public void run() throws FileNotFoundException {

File file = new File("test2.txt");

FileReader fr = new FileReader(file);



System.out.println("This won't run due to the error, because it wasn't enclosed in
a try-catch block.");

}
```

Now when I define the method this way, then I need to add the try catch block that handles this exception. This is a very important piece of information because it's a way of letting the user know that if the code fails, it will throw those particular types of exceptions.

You can also list multiple types of exceptions that a method throws:

```
public void run() throws FileNotFoundException, IOException {

File file = new File("test2.txt");

FileReader fr = new FileReader(file);



System.out.println("This won't run due to the error, because it wasn't enclosed in
a try-catch block.");
```

```
}
```

But if you surround the code with a try-catch block, then the error will be handled through the catch block, and the code will keep running after that block. In other words, Java tries to run the code in the `try` statement. If an error occurs, Java goes to the `catch` block and runs that.

```
run() {

File file = new File("test2.txt");

try {

FileReader fr = new FileReader(file);

System.out.println("This won't be run if an exception is found.");

} catch (Exception e) {

e.printStackTrace();

}

System.out.println("This code still runs because it's outside the Try block.");



}
```

In Eclipse, just surround the code you want to enclose with the try-catch block and right-click, then select **Surround with > Try Catch **block.

Although you can only have one try statement per try-catch block, you can have multiple catch blocks. This is useful if you want to handle different types of errors in different ways.

You can also add a `finally` block that will be processed regardless of whether an error takes place.

There's also a try-multicatch block that you can add. You add this in the same way, by selecting the statement, right-clicking, and choosing **Surround with > Multicatch**.

Here's what the multi catch might look like:

```
try {

test.run();

} catch (IOException | ParseException e) {



e.printStackTrace();

}
```

As you can see, two different catch blocks are combined here. The error is assigned to the `e` variable, so by running the `printStackTrace` method on e, you can see the error.

You could also manually separate the two catch blocks as follows:

```
try {

test.run();

} catch (IOException e) {



System.out.println("an IO exception happened here")

}



catch (ParseException f) {
```

```
System.out.println("a Parse exception happened here")

}
```

`Exception` is the most generic type of exception. It will catch any type of exception, and therefore it must appear last, since you want to use specific error catches to better identify the type of error that is taking place.

```
try {

// run some code here

}


catch (ParseException e)

{    // handle the ParseException here}


catch (Exception e)

{    // handle other exceptions}
```

If you look at this Exception page in the Javadoc, you can see that there are probably 50 different types of exceptions. You want to be really specific about the types of exceptions you catch so that you can more appropriately identify what might be wrong with your code.

You have to handle the exceptions in the right order. If you put the wrong exception first, it might catch the error before a more specific catch block handles the error.

The exceptions are automated when you use your IDE to add the try-catch blocks. The IDE will populate the exceptions based on the methods you're using.

## Finally

There's also a `finally` keyword that you can leverage. Here's an example:

```
try {

averageImpl.setInts(ints2);

} catch(IllegalArgumentException iae) {

System.out.println("Oops! can't use an empty array");

} catch(ArithmeticException ae) {

throw ae;

} finally {

System.out.println("Made it past the exception!");

}
```

– Java 7 for Absolute Beginners

What does `finally` do here?

> The finally keyword lets you create a block of code that gets run no matter what. Even if an exception happens, the code in the finally block gets run.
>
> Java 7 for Absolute Beginners

## The logic of looking for exceptions

In *Beginning Java 8 Fundamentals*, the author explains an important concept about why Java uses try-catch blocks. Without them, your code would look like a bunch of nested if-else blocks:

```
// Connect to the database
if (connected to the database successfully) {
// Fetch the employee record
if (employee record fetched) {
// Update the employee salary
if (update is successful) {
// Commit the changes
if (commit was successful ) {
// Employee salary was saved successfully
}
else {
// An error. Save failed
}
}
else {
//An error. Salary could not be updated
}
}
else {
// An error. Employee record does not exist
}
}
```

```
else {

// An error. Could not connect to the database

}
```

I think PHP often looks like this. With them, however, your code is much cleaner:

```
try {

// Connect to the database

// Fetch employee record

// Update employee salary

// Commit the changes

}

catch(DbConnectionException e1){

// Handle DB Connection exception here

}

catch(EmpNotFoundException e2){

// Handle employee not found exception here

}

catch(UpdateFailedException e3){

// Handle update failed exception here

}

catch(CommitFailedException e4){

// Handle commit failed exception here

}
```

The catch blocks will handle the errors if they occur. There is no spaghetti-looking code of nested if-else statements. This is part of the genius of Java.

## Checked vs unchecked exceptions

If an exception is checked, it means that it is caught in a try-catch block, like this:

```
public void ioOperation(boolean isResourceAvailable) {

try {

if (!isResourceAvailable) {

throw new IOException();

}

} catch(IOException e) {

// Handle caught exceptions.

}

}
```

This sample comes from Unchecked Exceptions, Java.

Note that only methods throw exceptions.