# Java Exceptions

## Table of contents


What is an exception? . . . . . 1
Exception hierarchy . . . . . 2
Java exception handling . . . . . 3
Custom exceptions . . . . . 7


Introduced in 2026-03-19-exceptions[1] (Lecture, Week 4, Part 1). Traced in depth in week5-tutorial-exceptions[2].

### What is an exception?

An **unexpected event that occurs during program execution**, e.g.:

```
System.out.println(5 / 0); // Infinity??
```

No — integer division by zero causes a `java.lang.ArithmeticException: / by zero`. Infinity is the tempting wrong answer: it is what floating-point division gives, not integer division.

Common causes:

- invalid user input
- loss of network connection
- physical limitations (out of disk memory)
- code errors
- opening an unavailable file

[1] courses/csse2002/2026-03-19-exceptions.pdf
[2] courses/csse2002/week5-tutorial-exceptions.pdf

When an exception occurs within a method **it creates an object** — the **exception object** — which contains information about the exception. *Effective Java* (2017), Item 75, says to *include failure-capture information in detail messages*: that object is where the diagnostic information has to live.

We can catch the exception to handle it, and the catch block is where the program is put back into a usable state:

```
try {
    System.out.println(5 / 0);
}  catch (ArithmeticException e) {
    System.out.println(e); // print out error message
    //do something to recover
}
```

## Exception hierarchy

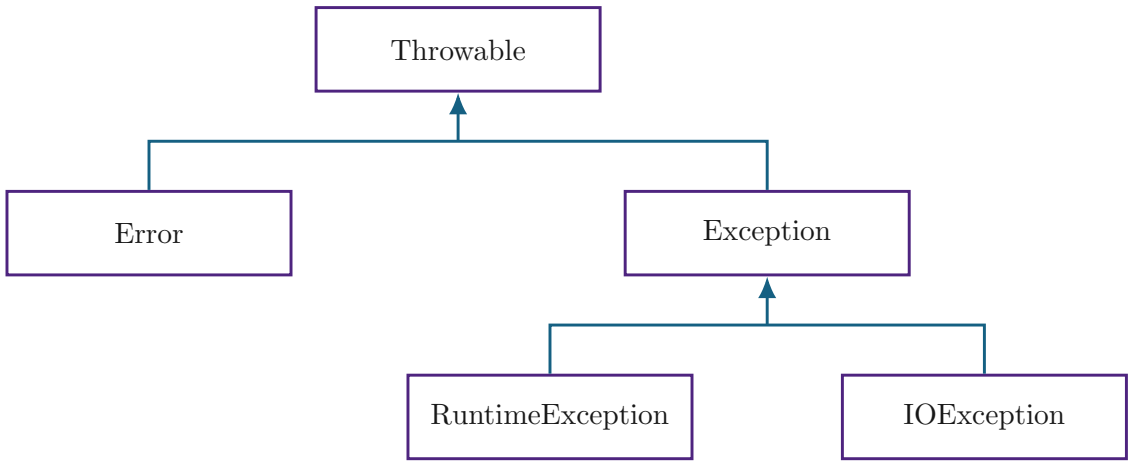


Figure 1: The exception hierarchy. Throwable is the root; Error and Exception sit beneath it; RuntimeException (unchecked) and IOException (checked) sit beneath Exception. Each connector joins the subclasses and points up to their parent. Which branch a class is on is what decides whether the compiler forces you to handle it, so this one picture settles the checked/unchecked question.

### `RuntimeException` — unchecked

A runtime exception happens **due to a programming error**. Also known as **unchecked** exceptions: the compiler does not force you to handle them.

| Cause | Exception |
|---|---|
| Improper use of an API | `IllegalArgumentException` |
| Null pointer access | `NullPointerException` |
| Out-of-bounds array access | `ArrayIndexOutOfBoundsException` |
| Dividing a number by 0 | `ArithmeticException` |

#### `IOException` — checked

Also known as a **checked** exception: it is checked by the compiler at compile time, and the programmer is prompted to handle it.

| Cause | Exception |
|---|---|
| Trying to open a file that doesn't exist | `FileNotFoundException` |
| Trying to read past the end of a file | `EOFException` |
| A connection attempt to a remote host fails | `ConnectException` |

### Java exception handling

There are three mechanisms: the `try…catch` block, the `throw` and `throws` keywords, and the `finally` block.

#### `try … catch`

Place the code that might generate an exception inside the `try` block. **Every `try` block is followed by a `catch` block.**

```
int result;
try {
    result = 5 / 0;
    …
} catch (ArithmeticException e) {
    System.out.println(e);      // print out error message
    result = Integer.MAX_VALUE; // Assign a default or fallback value
}
```

1. When an exception occurs, **the rest of the code inside the `try` block is skipped**.
2. The `catch` block catches the exception and the statements inside the `catch` block are executed.
3. If none of the statements in the `try` block generates an exception, **the `catch` block is skipped**.

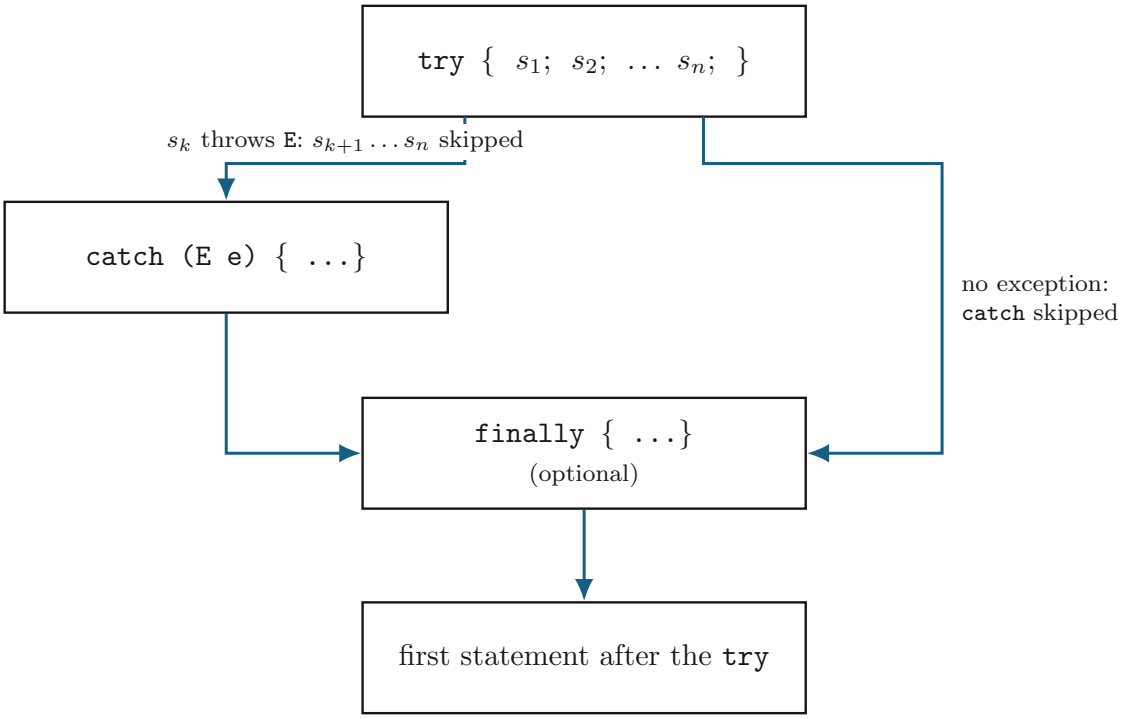


Figure 2: The two paths through a try/catch/finally, and the fact that both of them end up in finally. This is the three numbered rules together with the finally rule, drawn as one picture: the exceptional path abandons the remaining statements of the try and enters catch, the normal path skips catch entirely, and neither can skip finally.

### `throw`

The `throw` keyword is used to **explicitly throw a single exception**. Two examples:

```
public class Main {
    public static void main(String[] args) {
        divideByZero();
    }
    public static void divideByZero() {
        // throw an exception
        throw new ArithmeticException("Trying to divide by 0");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        checkEntry(17);
    }

    public static void checkEntry(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Access denied ");
        }
        else {
            System.out.println("Access granted ");
        }
    }
}
```

`throw` is used to manually trigger an error when your program detects something invalid, so it doesn't continue in a bad state. That is the point of the whole keyword, and it is also the bridge to java-specification[3]: throwing is how a method refuses input that breaks its precondition instead of limping on.

### `throws`

The `throws` keyword **declares the type of exceptions that might occur within the method**, and it is used **in the method declaration**. The caller is then the one prompted to handle it:

[3] courses/csse2002/java-specification.pdf

```java
public class ThrowsEx {
    public static void main(String[] args) {
        try {
            findFile();
        } catch (FileNotFoundException e) {
            System.out.println(e);
        }
    }
    public static void findFile() throws FileNotFoundException {
        // code that may generate FileNotFoundException
        File newFile = new File("file.txt");
        FileInputStream stream = new FileInputStream(newFile);
    }
}
```

`throw` and `throws` are not two spellings of one thing: `throw` (statement, inside a body) *raises* an exception; `throws` (clause, in a signature) *declares* that one can escape. Only checked exceptions have to be declared — that is exactly what "checked by the compiler" buys you.

### `finally`

- The `finally` block is executed **regardless of whether an exception is thrown**.
- The `finally` block is **optional**.

```java
try {
    //code
}
catch (ExceptionType1 e1) {
    // catch block
}
finally {
    // finally block always executes
}
```

### Exercise — what will the following print?

```java
try{
    System.out.println("Good Morning");
    throw  new FileNotFoundException();
    System.out.println("the earth says");
}catch (Exception e){
    System.out.println("hello");
```

```
}finally {
    System.out.println("world");
}
```

Answer:

```
Good Morning
hello
world
```

Working through it against the rules above: `"Good Morning"` prints; the `throw` immediately abandons the rest of the `try`, so `"the earth says"` never prints; the `catch` parameter is `Exception`, and `FileNotFoundException` is an `Exception`, so this catch matches and `"hello"` prints; `finally` always runs, so `"world"` prints last.

As written this snippet would not actually compile. `System.out.println("the earth says");` sits directly after an unconditional `throw` in the same block, and Java rejects that at compile time as an unreachable statement. Read it as a tracing exercise about the *order* of the three rules, not as something to paste into a file.

## Custom exceptions

Define your own by extending `Exception` (or a subclass of it), then declare it with `throws` and raise it with `throw`:

```
public class MyCustomException extends Exception {
...

}

int myFunction() throws MyCustomException {
throw new MyCustomException();
}
```

### Worked example — `ItemNotFoundException`

The exception class gives a no-argument constructor with a fixed message and a second constructor taking a custom one, both delegating to `super(...)` — the `Throwable` message is the failure-capture information from Item 75:

```java
public class ItemNotFoundException extends Exception{
    public ItemNotFoundException(){
        super("Item not found");
    }
    // Constructor that accepts a custom message
    public ItemNotFoundException(String message) {
        super(message);
    }
}
```

The class that uses it declares `throws` on the searching method and throws when the loop falls through without a match:

```java
public class Store {
    private List<String> items;
    public Store() {
        items = new ArrayList<>();
        items.add("apple");
        items.add("banana");
        items.add("orange");
    }
    public String findItem(String itemName) throws ItemNotFoundException {
        for (String item : items) {
            if (item.equals(itemName)) {
                return item;
            }
        }
        throw new ItemNotFoundException("Item '" + itemName + "' not found in the
        ↪  list.");
    }

    public static void main(String[] args) {
        Store store = new Store();
        try {
            String item = store.findItem("grape");
            System.out.println("Found item: " + item);
        } catch (ItemNotFoundException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

`findItem("grape")` misses, so the `try` is abandoned at the call — `"Found item: …"` never prints — and the catch reports `Item 'grape' not found in the list.` on `System.err`.