Chapter 6: Exceptions

Object Oriented Programming with Java

What is an exception | try and catch | Multiple catch | finally | try-with-resources | Nested try | Exception hierarchy | Checked / unchecked | throw and throws | Common exceptions | Error | Custom exceptions | Stack trace

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_06_Exceptions.html

INTRODUCTION

Learning Outcomes

Concept

Define an exception; classify runtime problems as `Exception` or `Error`.

Handling

Handle exceptions with `try`, `catch`, `finally` and try-with-resources; trace the flow step by step.

Hierarchy

Explain how `Throwable`, `Error`, `Exception` and `RuntimeException` relate, and why catch order matters.

Kinds

Tell checked from unchecked exceptions; apply the “catch or declare” rule.

Throwing

Throw exceptions with `throw`, declare them with `throws`; trace how an exception propagates up the call stack.

Practice

Recognize common exception classes, write a custom exception class and read a stack trace.

What Is an Exception?

- An **exception** is an unexpected event that occurs while a program runs and **disrupts its normal flow**.
- Examples: a file that cannot be found, division by zero, an index outside an array, using a `null` reference.
- Exceptions are created **automatically** by the Java runtime (the JVM) or explicitly by the programmer (`throw`).
- In Java an exception is an **object**: it carries its type, a message and where it happened.
- The exception mechanism lets you deal with abnormal situations in an **orderly** way.

When Java hits an exception, it roughly says: **"I give up. I don't know what to do now. Somebody else handle this!"**

Analogy

A waiter learns in the kitchen that a dish has run out. He can't fix it, so he **tells the manager**. The manager either solves it (suggests another dish) or passes it further up.

BASICS

Runtime Errors: Exception and Error

Compile errors stop a program from running at all; logic errors give no signal. A problem that arises **while the program runs** is thrown by the JVM as a `Throwable` object, which has two main branches.

Branch	What it means	What the program can do	Example
Exception (`Exception`)	An unusual condition the program can anticipate	Catch it and recover	`10 / 0`, `arr[5]` (on a 3-element array), a missing file
Error (`Error`)	A serious JVM- or system-level problem	Usually not caught; the program ends	`StackOverflowError`, `OutOfMemoryError`

Exceptions and stack traces are among the most powerful tools for finding and fixing the cause of runtime errors (debugging).

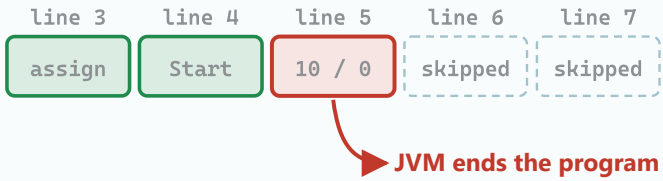
Uncaught Exception: The Program Crashes

```java
public class Crash {
  public static void main(String[] args) {
    int num = 10, den = 0;
    System.out.println("Start");
    int result = num / den;
    System.out.println("Result: " + result);
    System.out.println("Done");
  }
}
```

Output

```
Start
Exception in thread "main"
java.lang.ArithmeticException: / by zero
    at Crash.main(Crash.java:5)
```

- Line 5 divides an integer by zero; the JVM **throws** an `ArithmeticException` object.
- Nobody catches it, so the program **terminates**: “Result” and “Done” are never printed.
- The JVM prints the exception's **type**, **message** and **location** (class, method, line).



BASICS

Two Ways to Deal with an Exception

1) Handle it

The method puts the risky code in `try` and resolves the problem itself in `catch`.
Example: if the file is missing (`FileNotFoundException`), a file-reading method shows a message and goes on with default settings.

2) Make it the caller's problem (declare)

The method does not handle the exception; it declares it with `throws` in its signature. The exception becomes the **caller's** problem.
The caller faces the same choice: handle it or pass it up.

Which one? The code that can **meaningfully resolve** the exception should handle it. Code that got the file name from the user can ask again when the file is missing; a low-level read method cannot know that.

“Something Went Wrong”: With a Return Code

```
public static int indexOf(String[] names, String name) {
  for (int i = 0; i < names.length; i++) {
    if (names[i].equals(name)) {
      return i;
    }
  }
  return -1;
}
// in main
String[] names = {"Ali", "Ayşe", "Can"};
System.out.println(indexOf(names, "Can"));
int k = indexOf(names, "Ece");   // k = -1
System.out.println(names[k]);
```

Output

```
2
java.lang.ArrayIndexOutOfBoundsException:
  Index -1 out of bounds for length 3
```

- Exceptions are how a program signals that **“something went wrong”**.
- But “wrong” is **subjective**: is a name that isn't found an error, or a normal result?
- When there is no match, this method returns `-1` instead of throwing an exception.
- If the caller **forgets** to check for `-1`, the error shows up elsewhere, at `names[-1]` (output on the left).

Exceptions Instead of Return Codes

```
public static int find(String[] names, String name) {
  for (int i = 0; i < names.length; i++) {
    if (names[i].equals(name)) {
      return i;
    }
  }
  throw new IllegalArgumentException("Not found: " + name);
}
// in main
try {
  System.out.println(find(names, "Ece"));
} catch (IllegalArgumentException e) {
  System.out.println("Error: " + e.getMessage());
}
```

Output

```
Error: Not found: Ece
```

- As a rule, avoid **special return codes** like `-1`; prefer Java's exception mechanism.
- Exceptions handle error cases **clearly** and **effectively**.

Why exceptions help

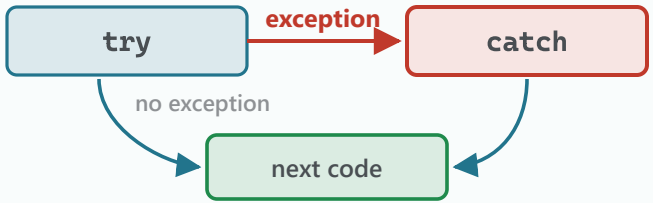
+ Clear, consistent error handling; errors never mix with normal results
+ More readable code, easier debugging
+ More reliable code

try and catch Statements

```java
try {
  // Code that may fail
  int result = 10 / 0; // ArithmeticException
} catch (ArithmeticException e) {
  // Handle the error
  System.out.println("Error: " + e.getMessage());
}
```

Output

```
Error: / by zero
```



- The **basic structure** of error handling.
- `try` block: code that **may fail**.
- `catch` block: runs if an exception of the given type occurs inside `try`, and **handles** it.
- `catch (ArithmeticException e)` : `e` refers to the thrown exception **object** (like a parameter).
- If an exception occurs, the **rest of** `try` **is skipped**; if none occurs, `catch` never runs.
- After catch, the program continues its **normal flow**.

try-catch Flow: Step by Step

```
1  public class DivideDemo {
2    public static void main(String[] args) {
▶      int num = 10, den = 0;
4      System.out.println("Start");
5      try {
6        System.out.println("Before divide");
7        int result = num / den;
8        System.out.println("Result: " + result);
9      } catch (ArithmeticException e) {
10       System.out.println("Error: " +
   e.getMessage());
11     }
12     System.out.println("Done");
13   }
14 }
```

println("Before divide")

result = num / den

println("Result: " + …)

catch (ArithmeticException e)

println("Error: " + …)

println("Done")

Variables are created. `den` is zero: trouble ahead.

Step 1 / 7

OUTPUT

try-catch Flow: Step by Step

```
1  public class DivideDemo {
2    public static void main(String[] args) {
3      int num = 10, den = 0;
▶      System.out.println("Start");
5      try {
6        System.out.println("Before divide");
7        int result = num / den;
8        System.out.println("Result: " + result);
9      } catch (ArithmeticException e) {
10       System.out.println("Error: " +
e.getMessage());
11     }
12     System.out.println("Done");
13   }
14 }
```

println("Before divide")

result = num / den

println("Result: " + …)

catch (ArithmeticException e)

println("Error: " + …)

println("Done")

Normal code outside `try` runs.

Step 2 / 7

OUTPUT

```
Start
```

try-catch Flow: Step by Step

```
1  public class DivideDemo {
2    public static void main(String[] args) {
3      int num = 10, den = 0;
4      System.out.println("Start");
5      try {
6        System.out.println("Before divide");
7        int result = num / den;
8        System.out.println("Result: " + result);
9      } catch (ArithmeticException e) {
10       System.out.println("Error: " +
   e.getMessage());
11     }
12     System.out.println("Done");
13   }
14 }
```

println("Before divide")

result = num / den

println("Result: " + …)

catch (ArithmeticException e)

println("Error: " + …)

println("Done")

We enter the `try` block; the first line runs fine.

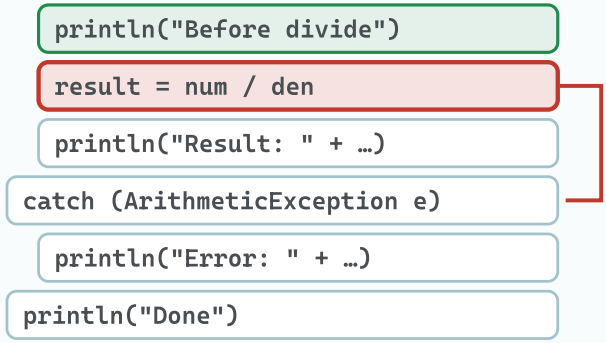
Step 3 / 7

OUTPUT

```
Start
Before divide
```

try-catch Flow: Step by Step

```
1  public class DivideDemo {
2    public static void main(String[] args) {
3      int num = 10, den = 0;
4      System.out.println("Start");
5      try {
6        System.out.println("Before divide");
▶        int result = num / den;
8        System.out.println("Result: " + result);
9      } catch (ArithmeticException e) {
10       System.out.println("Error: " +
   e.getMessage());
11     }
12     System.out.println("Done");
13   }
14 }
```



10 / 0: the JVM creates an `ArithmeticException` object and **throws** it. No assignment happens.

Step 4 / 7

OUTPUT

```
Start
Before divide
```

try-catch Flow: Step by Step

```
1  public class DivideDemo {
2    public static void main(String[] args) {
3      int num = 10, den = 0;
4      System.out.println("Start");
5      try {
6        System.out.println("Before divide");
7        int result = num / den;
8        System.out.println("Result: " + result);
▶      } catch (ArithmeticException e) {
10       System.out.println("Error: " +
   e.getMessage());
11     }
12     System.out.println("Done");
13   }
14 }
```

println("Before divide")

result = num / den

println("Result: " + …)

catch (ArithmeticException e) ✓

println("Error: " + …)

println("Done")

Type matches: `catch` runs and `e` refers to the exception object. The “Result” line was **skipped**.

Step 5 / 7

OUTPUT

```
Start
Before divide
```

try-catch Flow: Step by Step

```
1  public class DivideDemo {
2    public static void main(String[] args) {
3      int num = 10, den = 0;
4      System.out.println("Start");
5      try {
6        System.out.println("Before divide");
7        int result = num / den;
8        System.out.println("Result: " + result);
9      } catch (ArithmeticException e) {
▶        System.out.println("Error: " +
   e.getMessage());
11     }
12     System.out.println("Done");
13   }
14 }
```

println("Before divide")

result = num / den

println("Result: " + …)

catch (ArithmeticException e)

println("Error: " + …)

println("Done")

e.getMessage() → "/ by zero".

Step 6 / 7

OUTPUT

```
Start
Before divide
Error: / by zero
```

try-catch Flow: Step by Step

```
1  public class DivideDemo {
2    public static void main(String[] args) {
3      int num = 10, den = 0;
4      System.out.println("Start");
5      try {
6        System.out.println("Before divide");
7        int result = num / den;
8        System.out.println("Result: " + result);
9      } catch (ArithmeticException e) {
10       System.out.println("Error: " +
   e.getMessage());
11     }
▶      System.out.println("Done");
13   }
14 }
```

println("Before divide")

result = num / den

println("Result: " + …)

catch (ArithmeticException e)

println("Error: " + …)

println("Done")

No **crash**: after catch, normal flow continues.

Step 7 / 7

OUTPUT

```
Start
Before divide
Error: / by zero
Done
```

Getting Information from the Exception Object

```java
try {
  int[] arr = new int[3];
  arr[5] = 7;
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println(e.getMessage());
  System.out.println(e);
  System.out.println(e.getClass().getSimpleName());
  e.printStackTrace();
}
```

Output

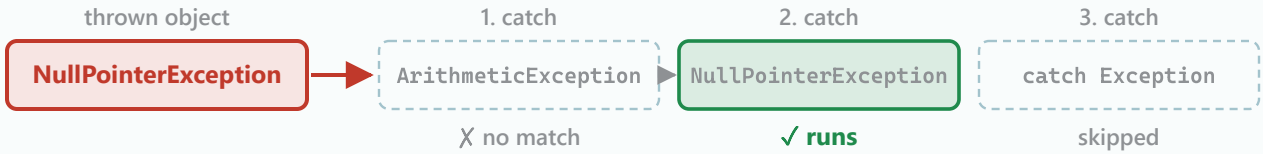
```
Index 5 out of bounds for length 3
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
ArrayIndexOutOfBoundsException
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
    at ExceptionDemo.main(ExceptionDemo.java:5)
```

Method	Returns
`getMessage()`	Just the message (or `null`)
`toString()`	Full class name + ": " + message; `println(e)` uses it
`getClass()`	The exception's actual class
`printStackTrace()`	`toString()` + stack trace; writes to **System.err**
`getCause()`	The exception that caused this one (if any)

Multiple catch Blocks

```
try {
  // Code that may fail
} catch (ArithmeticException e) {
  // Handle ArithmeticException
} catch (NullPointerException e) {
  // Handle NullPointerException
} catch (Exception e) {
  // All other exceptions
}
```

- A block of code can throw **more than one** type of exception.
- Add a separate `catch` block per type; each holds the response **specific** to that type.
- When an exception is thrown, the catch blocks are tried **top to bottom**; the **first match** runs and the rest are skipped.
- "Match" means the thrown object is the catch type **itself or a subclass** (the `instanceof` rule).
- The final `catch (Exception e)` is the **general** case that catches everything else.



Chained catch Blocks: Order Matters

✗ GENERAL TYPE FIRST

```
try {
  int result = 10 / 0;
} catch (Exception e) {
  System.out.println("General error");
} catch (ArithmeticException e) {
  System.out.println("Arithmetic error");
}
```

```
OrderError.java:7: error: exception
ArithmeticException
    has already been caught
```

- If the catch types are related by **inheritance**, order matters: **subclasses must come before superclasses**.
- `ArithmeticException` is an `Exception`; the first block already catches it, so the second **can never run**. The compiler rejects this.
- For unrelated types (`ArithmeticException`, `NullPointerException`) order does not matter.

✓ SPECIFIC TO GENERAL

```
} catch (ArithmeticException e) {
  System.out.println("Arithmetic error");
} catch (Exception e) {
  System.out.println("General error");
}
```

Which catch Runs?

```
1  static void divide(String[] input) {
2    try {
3      int x = Integer.parseInt(input[0]);
▶      int y = Integer.parseInt(input[1]);
5      System.out.println("Result: " + (x / y));
6    } catch (NumberFormatException e) {
7      System.out.println("Not a number: " +
   e.getMessage());
8    } catch (ArithmeticException e) {
9      System.out.println("Zero: " + e.getMessage());
10   } catch (ArrayIndexOutOfBoundsException e) {
11     System.out.println("Missing: " +
   e.getMessage());
12   } finally {
13     System.out.println("finally ran");
14   }
15 }
16 // in main
17 divide(new String[] {"10", "abc"});
18 divide(new String[] {"10", "0"});
19 divide(new String[] {"10"});
```

`NumberFormatException`

1. `catch: NumberFormat`
2. `catch: Arithmetic`
3. `catch: ArrayIndexOutOfBounds`
4. `finally`

1st call: `parseInt("abc")` → `NumberFormatException`. The division line is skipped.

Step 1 / 7

OUTPUT

Which catch Runs?

```
1  static void divide(String[] input) {
2    try {
3      int x = Integer.parseInt(input[0]);
4      int y = Integer.parseInt(input[1]);
5      System.out.println("Result: " + (x / y));
▶    } catch (NumberFormatException e) {
▶      System.out.println("Not a number: " +
   e.getMessage());
8    } catch (ArithmeticException e) {
9      System.out.println("Zero: " + e.getMessage());
10   } catch (ArrayIndexOutOfBoundsException e) {
11     System.out.println("Missing: " +
   e.getMessage());
12   } finally {
13     System.out.println("finally ran");
14   }
15 }
16 // in main
17 divide(new String[] {"10", "abc"});
18 divide(new String[] {"10", "0"});
19 divide(new String[] {"10"});
```

NumberFormatException

1. catch: NumberFormat ✓
2. catch: Arithmetic
3. catch: ArrayIndexOutOfBounds
finally

The first catch matches and runs; the other catches are **not checked**.

OUTPUT

```
Not a number: For input string: "abc"
```

Which catch Runs?

```
1  static void divide(String[] input) {
2    try {
3      int x = Integer.parseInt(input[0]);
4      int y = Integer.parseInt(input[1]);
5      System.out.println("Result: " + (x / y));
6    } catch (NumberFormatException e) {
7      System.out.println("Not a number: " +
   e.getMessage());
8    } catch (ArithmeticException e) {
9      System.out.println("Zero: " + e.getMessage());
10   } catch (ArrayIndexOutOfBoundsException e) {
11     System.out.println("Missing: " +
   e.getMessage());
▶    } finally {
13     System.out.println("finally ran");
14   }
15 }
16 // in main
17 divide(new String[] {"10", "abc"});
18 divide(new String[] {"10", "0"});
19 divide(new String[] {"10"});
```

NumberFormatException

1. catch: NumberFormat ✓
2. catch: Arithmetic
3. catch: ArrayIndexOutOfBounds

finally

Then `finally`.

Step 3 / 7

OUTPUT

```
Not a number: For input string: "abc"
finally ran
```

Which catch Runs?

```
1  static void divide(String[] input) {
2    try {
3      int x = Integer.parseInt(input[0]);
4      int y = Integer.parseInt(input[1]);
▶      System.out.println("Result: " + (x / y));
6    } catch (NumberFormatException e) {
7      System.out.println("Not a number: " +
   e.getMessage());
8    } catch (ArithmeticException e) {
9      System.out.println("Zero: " + e.getMessage());
10   } catch (ArrayIndexOutOfBoundsException e) {
11     System.out.println("Missing: " +
   e.getMessage());
12   } finally {
13     System.out.println("finally ran");
14   }
15 }
16 // in main
17 divide(new String[] {"10", "abc"});
18 divide(new String[] {"10", "0"});
19 divide(new String[] {"10"});
```

`ArithmeticException`

1. `catch: NumberFormat`
2. `catch: Arithmetic`
3. `catch: ArrayIndexOutOfBounds`
4. `finally`

2nd call: `10 / 0` → `ArithmeticException`.

Step 4 / 7

OUTPUT

```
Not a number: For input string: "abc"
finally ran
```

Which catch Runs?

```
1  static void divide(String[] input) {
2    try {
3      int x = Integer.parseInt(input[0]);
4      int y = Integer.parseInt(input[1]);
5      System.out.println("Result: " + (x / y));
6    } catch (NumberFormatException e) {
7      System.out.println("Not a number: " +
   e.getMessage());
▶    } catch (ArithmeticException e) {
▶      System.out.println("Zero: " + e.getMessage());
10   } catch (ArrayIndexOutOfBoundsException e) {
11     System.out.println("Missing: " +
   e.getMessage());
12   } finally {
13     System.out.println("finally ran");
14   }
15 }
16 // in main
17 divide(new String[] {"10", "abc"});
18 divide(new String[] {"10", "0"});
19 divide(new String[] {"10"});
```

ArithmeticException

1. catch: NumberFormat ✗
2. catch: Arithmetic ✓
3. catch: ArrayIndexOutOfBounds
finally

The 1st catch doesn't match (✗), the 2nd does (✓).

Step 5 / 7

OUTPUT

```
Not a number: For input string: "abc"
finally ran
Zero: / by zero
```

Which catch Runs?

```
1  static void divide(String[] input) {
2    try {
3      int x = Integer.parseInt(input[0]);
▶      int y = Integer.parseInt(input[1]);
5      System.out.println("Result: " + (x / y));
6    } catch (NumberFormatException e) {
7      System.out.println("Not a number: " +
   e.getMessage());
8    } catch (ArithmeticException e) {
9      System.out.println("Zero: " + e.getMessage());
10   } catch (ArrayIndexOutOfBoundsException e) {
11     System.out.println("Missing: " +
   e.getMessage());
12   } finally {
13     System.out.println("finally ran");
14   }
15 }
16 // in main
17 divide(new String[] {"10", "abc"});
18 divide(new String[] {"10", "0"});
19 divide(new String[] {"10"});
```

`ArrayIndexOutOfBoundsException`

1. `catch: NumberFormat`
2. `catch: Arithmetic`
3. `catch: ArrayIndexOutOfBounds`
4. `finally`

3rd call: the array has one element; `input[1]` → `ArrayIndexOutOfBoundsException`.

Step 6 / 7

OUTPUT

```
Not a number: For input string: "abc"
finally ran
Zero: / by zero
finally ran
```

Which catch Runs?

```
1  static void divide(String[] input) {
2    try {
3      int x = Integer.parseInt(input[0]);
4      int y = Integer.parseInt(input[1]);
5      System.out.println("Result: " + (x / y));
6    } catch (NumberFormatException e) {
7      System.out.println("Not a number: " +
   e.getMessage());
8    } catch (ArithmeticException e) {
9      System.out.println("Zero: " + e.getMessage());
▶    } catch (ArrayIndexOutOfBoundsException e) {
▶      System.out.println("Missing: " +
   e.getMessage());
▶    } finally {
13     System.out.println("finally ran");
14   }
15 }
16 // in main
17 divide(new String[] {"10", "abc"});
18 divide(new String[] {"10", "0"});
19 divide(new String[] {"10"});
```

ArrayIndexOutOfBoundsException

1. catch: NumberFormat ✗

2. catch: Arithmetic ✗

3. catch: ArrayIndexOutOfBounds ✓

finally

The first two catches don't match, the third does; then `finally`.

Step 7 / 7

OUTPUT

```
Not a number: For input string: "abc"
finally ran
Zero: / by zero
finally ran
Missing: Index 1 out of bounds for
length 1
finally ran
```

One catch, Several Types (Java 7+)

```
try {
  Object o = "Hello";
  Integer i = (Integer) o;
} catch (ClassCastException
    | NullPointerException e) {
  System.out.println("Type or null error: "
    + e.getClass().getSimpleName());
}
```

Output

```
Type or null error: ClassCastException
```

- Same response? Join the types with `|` to avoid duplicate code.
- In this block `e` is implicitly **final**; it cannot be reassigned.

✗ CAN'T COMBINE A SUBCLASS WITH ITS SUPERCLASS

```
} catch (FileNotFoundException
    | IOException e) {
```

```
error: Alternatives in a multi-catch
statement
  cannot be related by subclassing
  Alternative FileNotFoundException is a
  subclass of alternative IOException
```

Fix: name only the supertype: `catch (IOException e)`. It already covers the subtype.

The finally Block

```
try {
  // Code that may fail
} catch (Exception e) {
  // All other exceptions
} finally {
  // Always runs
}
```

- Comes after the `try` and `catch` blocks.
- Runs **in every case**: whether or not an exception occurs, and whether or not it is caught.
- Even if `try` or `catch` contains a `return`, it runs **before** the method exits.
- **Guarantees** that certain steps run: **resource management** and cleanup, such as closing files, network and database connections.
- `try` must be followed by at least one `catch` **or** a `finally`; `try { } finally { }` is valid.
- The one exception: `System.exit()` stops the JVM at once; finally does not run.

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
▶    System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

`try { … }`

`catch (ArithmeticException e)`

`finally { … }`

no return value yet

Before `println` runs, its argument must be evaluated: `divide(10, 2)` is called.

Step 1 / 13

CALL STACK

`main()`

OUTPUT

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
▶      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

try { … }

catch (ArithmeticException e)

finally { … }

no return value yet

The try block starts.

Step 2 / 13

CALL STACK

divide(10, 2)

main()

OUTPUT

try started

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
▶      int s = a / b;
▶      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

`try { … }`

`catch (ArithmeticException e)`

`finally { … }`

no return value yet

The division succeeds: `s = 5`.

Step 3 / 13

CALL STACK

`divide(10, 2)`

`main()`

OUTPUT

```
try started
try ended
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
▶      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

`try { … }`

`catch (ArithmeticException e)`

`finally { … }`

pending return value: 5

`return s` : the return value **5** is ready, but the method doesn't end yet...

Step 4 / 13

CALL STACK

`divide(10, 2)`

`main()`

OUTPUT

```
try started
try ended
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
▶      System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

try { … }

catch (ArithmeticException e)

finally { … }

pending return value: 5

…first `finally` runs. catch was skipped (no exception).

Step 5 / 13

CALL STACK

- divide(10, 2)
- main()

OUTPUT

```
try started
try ended
finally
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
▶    System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

try { … }

catch (ArithmeticException e)

finally { … }

pending return value: 5

Now 5 is returned and printed.

Step 6 / 13

CALL STACK

main()

OUTPUT

```
try started
try ended
finally
Result = 5
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
▶    System.out.println("Result = " + divide(10, 0));
17 }
```

`try { … }`

`catch (ArithmeticException e)`

`finally { … }`

no return value yet

Second call: `divide(10, 0)`.

Step 7 / 13

CALL STACK

`main()`

OUTPUT

```
try started
try ended
finally
Result = 5
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
▶      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

try { … }

catch (ArithmeticException e)

finally { … }

no return value yet

try starts.

Step 8 / 13

CALL STACK

divide(10, 0)

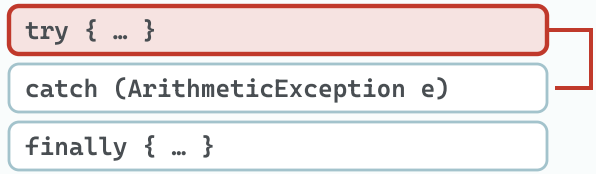
main()

OUTPUT

```
try started
try ended
finally
Result = 5
try started
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
▶      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```



no return value yet

`10 / 0` → exception! The rest of try (“try ended”, `return s`) is skipped.

Step 9 / 13

CALL STACK

- `divide(10, 0)`
- `main()`

OUTPUT

```
try started
try ended
finally
Result = 5
try started
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
▶      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

try { … }

catch (ArithmeticException e) ✓

finally { … }

no return value yet

catch matches and prints the message.

Step 10 / 13

CALL STACK

- divide(10, 0)
- main()

OUTPUT

```
try started
try ended
finally
Result = 5
try started
catch: / by zero
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
▶      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

try { … }

catch (ArithmeticException e) ✓

finally { … }

pending return value: -1

`return -1` : the value is ready, but again finally runs first.

Step 11 / 13

CALL STACK

- `divide(10, 0)`
- `main()`

OUTPUT

```
try started
try ended
finally
Result = 5
try started
catch: / by zero
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
▶      System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
16   System.out.println("Result = " + divide(10, 0));
17 }
```

try { … }

catch (ArithmeticException e) ✓

finally { … }

pending return value: -1

`finally` runs this time too.

Step 12 / 13

CALL STACK

- `divide(10, 0)`
- `main()`

OUTPUT

```
try started
try ended
finally
Result = 5
try started
catch: / by zero
finally
```

finally and return: Step by Step

```
1  static int divide(int a, int b) {
2    try {
3      System.out.println("try started");
4      int s = a / b;
5      System.out.println("try ended");
6      return s;
7    } catch (ArithmeticException e) {
8      System.out.println("catch: " + e.getMessage());
9      return -1;
10   } finally {
11     System.out.println("finally");
12   }
13 }
14 public static void main(String[] args) {
15   System.out.println("Result = " + divide(10, 2));
▶    System.out.println("Result = " + divide(10, 0));
17 }
```

try { … }

catch (ArithmeticException e) ✓

finally { … }

pending return value: -1

-1 is returned. So finally ran **in both cases**.

Step 13 / 13

CALL STACK

main()

OUTPUT

```
try started
try ended
finally
Result = 5
try started
catch: / by zero
finally
Result = -1
```

try-with-resources (Java 7+)

```
class Resource implements AutoCloseable {
  private String name;
  Resource(String name) {
    this.name = name;
    System.out.println(name + " opened");
  }
  void use(int x) {
    System.out.println(name + " in use");
    int s = 10 / x;
  }
  @Override
  public void close() {
    System.out.println(name + " closed");
  }
}
```

```
try (Resource r = new Resource("file")) {
  r.use(0);
} catch (ArithmeticException e) {
  System.out.println("Error: " + e.getMessage());
}
```

Output

```
file opened
file in use
file closed
Error: / by zero
```

- A resource opened in `try (...)` is closed **automatically** (`close()`) when the block ends, even after an exception.
- The resource is closed **before catch** runs. Its class must implement `AutoCloseable` , as streams, `Scanner` , NIO channels and database connections do; no `finally` or `null` check is needed.
- This is the modern, safe way to do what `finally` did for closing resources.

Two Resources: Opening and Closing Order

`Resource`: its constructor prints "... opened", `use` prints "... in use", `close()` prints "... closed".

```
▶ try (Resource a = new Resource("file");
2       Resource b = new Resource("net")) {
3   b.use(0);
4   System.out.println("try end");
5 } catch (ArithmeticException e) {
6   System.out.println("Error: " +
  e.getMessage());
7 } finally {
8   System.out.println("finally");
9 }
```

Resources (opening order)



The first resource is created: `file opened`.

Step 1 / 8

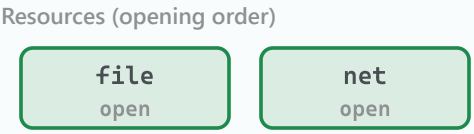
OUTPUT

```
file opened
```

Two Resources: Opening and Closing Order

`Resource`: its constructor prints "... opened", `use` prints "... in use", `close()` prints "... closed".

```
1 try (Resource a = new Resource("file");
▶       Resource b = new Resource("net")) {
3   b.use(0);
4   System.out.println("try end");
5 } catch (ArithmeticException e) {
6   System.out.println("Error: " +
  e.getMessage());
7 } finally {
8   System.out.println("finally");
9 }
```



The second resource is created. Resources are separated by **semicolons** and opened in the order written.

Step 2 / 8

OUTPUT

```
file opened
net opened
```

Two Resources: Opening and Closing Order

`Resource`: its constructor prints "... opened", `use` prints "... in use", `close()` prints "... closed".

```
1 try (Resource a = new Resource("file");
2       Resource b = new Resource("net")) {
▶   b.use(0);
4   System.out.println("try end");
5 } catch (ArithmeticException e) {
6   System.out.println("Error: " +
  e.getMessage());
7 } finally {
8   System.out.println("finally");
9 }
```

Resources (opening order)



`b.use(0)`: first "net in use" is printed, then `10 / 0`.

Step 3 / 8

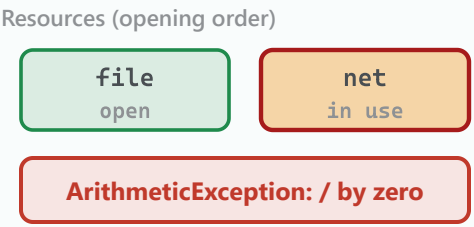
OUTPUT

```
file opened
net opened
net in use
```

Two Resources: Opening and Closing Order

`Resource`: its constructor prints "... opened", `use` prints "... in use", `close()` prints "... closed".

```
1 try (Resource a = new Resource("file");
2      Resource b = new Resource("net")) {
▶   b.use(0);
4   System.out.println("try end");
5 } catch (ArithmeticException e) {
6   System.out.println("Error: " +
  e.getMessage());
7 } finally {
8   System.out.println("finally");
9 }
```



Exception thrown: the rest of try ("try end") is skipped. But the resources are closed **before** catch.

Step 4 / 8

OUTPUT

```
file opened
net opened
net in use
```

Two Resources: Opening and Closing Order

`Resource`: its constructor prints "... opened", `use` prints "... in use", `close()` prints "... closed".

```
1 try (Resource a = new Resource("file");
▶       Resource b = new Resource("net")) {
3   b.use(0);
4   System.out.println("try end");
5 } catch (ArithmeticException e) {
6   System.out.println("Error: " +
  e.getMessage());
7 } finally {
8   System.out.println("finally");
9 }
```



Closing happens in **reverse order**: `b`, opened last, closes first (`close()`).

Step 5 / 8

OUTPUT

```
file opened
net opened
net in use
net closed
```

Two Resources: Opening and Closing Order

`Resource`: its constructor prints "... opened", `use` prints "... in use", `close()` prints "... closed".

```
▶ try (Resource a = new Resource("file");
2       Resource b = new Resource("net")) {
3   b.use(0);
4   System.out.println("try end");
5 } catch (ArithmeticException e) {
6   System.out.println("Error: " +
  e.getMessage());
7 } finally {
8   System.out.println("finally");
9 }
```



Then `a` closes. Both resources were closed despite the exception.

Step 6 / 8

OUTPUT

```
file opened
net opened
net in use
net closed
file closed
```

Two Resources: Opening and Closing Order

`Resource`: its constructor prints "... opened", `use` prints "... in use", `close()` prints "... closed".

```
1 try (Resource a = new Resource("file");
2       Resource b = new Resource("net")) {
3   b.use(0);
4   System.out.println("try end");
▶ } catch (ArithmeticException e) {
▶   System.out.println("Error: " +
  e.getMessage());
7 } finally {
8   System.out.println("finally");
9 }
```

Resources (opening order)



caught

Now catch runs: the exception is caught.

Step 7 / 8

OUTPUT

```
file opened
net opened
net in use
net closed
file closed
Error: / by zero
```

Two Resources: Opening and Closing Order

`Resource`: its constructor prints "... opened", `use` prints "... in use", `close()` prints "... closed".

```
1 try (Resource a = new Resource("file");
2      Resource b = new Resource("net")) {
3   b.use(0);
4   System.out.println("try end");
5 } catch (ArithmeticException e) {
6   System.out.println("Error: " +
  e.getMessage());
7 } finally {
▶   System.out.println("finally");
9 }
```

Resources (opening order)



`finally` runs last. Order: open → use → close (reversed) → catch → finally.

Step 8 / 8

OUTPUT

```
file opened
net opened
net in use
net closed
file closed
Error: / by zero
finally
```

Nested try Blocks

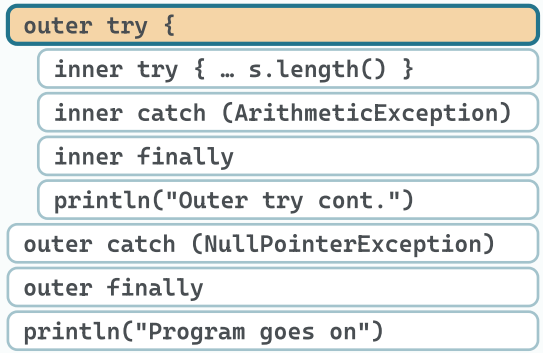
```
try {
  // Outer try block
  try {
    // Inner try block
    // Code that may fail
  } catch (Exception e) {
    // Inner catch block
  } finally {
    // Inner finally block
  }
} catch (Exception e) {
  // Outer catch block
} finally {
  // Outer finally block
}
```

A `try` block can contain another `try` block. If an exception occurs in the inner try:

1. If one of the inner try's catches **matches**, it runs.
2. Otherwise the exception moves **outward**; the first matching catch is found and runs.
3. If no catch matches, the exception leaves the method; if nobody catches it, the **JVM's default handler** ends the program.
4. In every case, the `finally` (if any) at each level the exception passes through runs.

Nested try: The Exception Moves Outward

```
1  public static void main(String[] args) {
2    try {
▶      System.out.println("Outer try");
4      try {
5        System.out.println("Inner try");
6        String s = null;
7        System.out.println(s.length());
8      } catch (ArithmeticException e) {
9        System.out.println("Inner catch");
10     } finally {
11       System.out.println("Inner finally");
12     }
13     System.out.println("Outer try cont.");
14   } catch (NullPointerException e) {
15     System.out.println("Outer catch:
   NPE");
16   } finally {
17     System.out.println("Outer finally");
18   }
19   System.out.println("Program goes on");
20 }
```



The outer try starts.

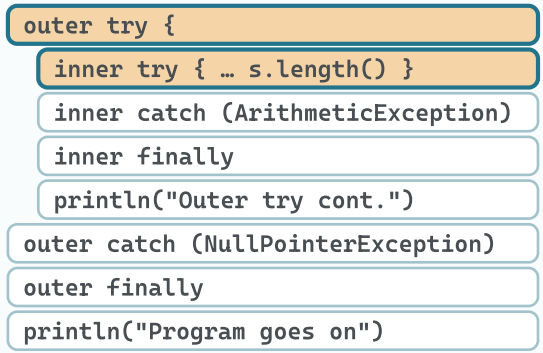
Step 1 / 8

OUTPUT

```
Outer try
```

Nested try: The Exception Moves Outward

```
1  public static void main(String[] args) {
2    try {
3      System.out.println("Outer try");
4      try {
▶        System.out.println("Inner try");
▶        String s = null;
7        System.out.println(s.length());
8      } catch (ArithmeticException e) {
9        System.out.println("Inner catch");
10     } finally {
11       System.out.println("Inner finally");
12     }
13     System.out.println("Outer try cont.");
14   } catch (NullPointerException e) {
15     System.out.println("Outer catch:
   NPE");
16   } finally {
17     System.out.println("Outer finally");
18   }
19   System.out.println("Program goes on");
20 }
```



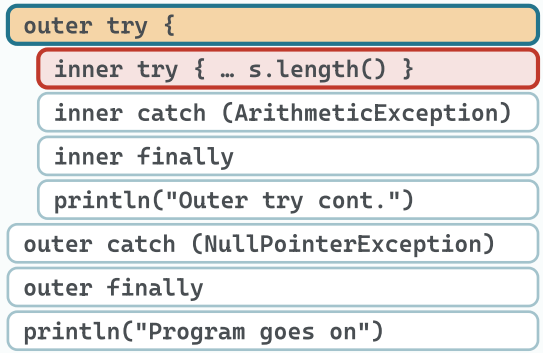
The inner try starts; `s = null`.

OUTPUT

```
Outer try
Inner try
```

Nested try: The Exception Moves Outward

```
1  public static void main(String[] args) {
2    try {
3      System.out.println("Outer try");
4      try {
5        System.out.println("Inner try");
6        String s = null;
▶        System.out.println(s.length());
8      } catch (ArithmeticException e) {
9        System.out.println("Inner catch");
10     } finally {
11       System.out.println("Inner finally");
12     }
13     System.out.println("Outer try cont.");
14   } catch (NullPointerException e) {
15     System.out.println("Outer catch:
   NPE");
16   } finally {
17     System.out.println("Outer finally");
18   }
19   System.out.println("Program goes on");
20 }
```



Calling a method on `null` → `NullPointerException`.

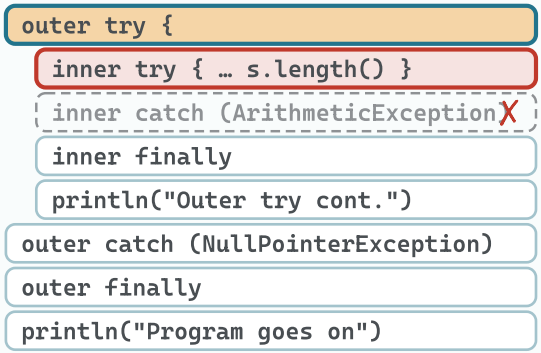
Step 3 / 8

OUTPUT

```
Outer try
Inner try
```

Nested try: The Exception Moves Outward

```
1  public static void main(String[] args) {
2    try {
3      System.out.println("Outer try");
4      try {
5        System.out.println("Inner try");
6        String s = null;
7        System.out.println(s.length());
▶      } catch (ArithmeticException e) {
9        System.out.println("Inner catch");
10     } finally {
11       System.out.println("Inner finally");
12     }
13     System.out.println("Outer try cont.");
14   } catch (NullPointerException e) {
15     System.out.println("Outer catch:
   NPE");
16   } finally {
17     System.out.println("Outer finally");
18   }
19   System.out.println("Program goes on");
20 }
```



The inner catch only catches `ArithmeticException` : **no match**.

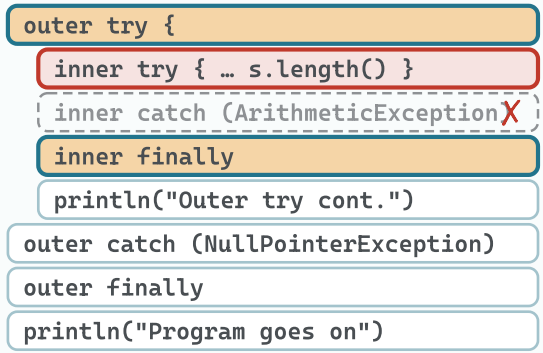
Step 4 / 8

OUTPUT

```
Outer try
Inner try
```

Nested try: The Exception Moves Outward

```
1  public static void main(String[] args) {
2    try {
3      System.out.println("Outer try");
4      try {
5        System.out.println("Inner try");
6        String s = null;
7        System.out.println(s.length());
8      } catch (ArithmeticException e) {
9        System.out.println("Inner catch");
10     } finally {
▶        System.out.println("Inner finally");
12     }
13     System.out.println("Outer try cont.");
14   } catch (NullPointerException e) {
15     System.out.println("Outer catch:
   NPE");
16   } finally {
17     System.out.println("Outer finally");
18   }
19   System.out.println("Program goes on");
20 }
```



Before the exception leaves, the **inner finally** runs.

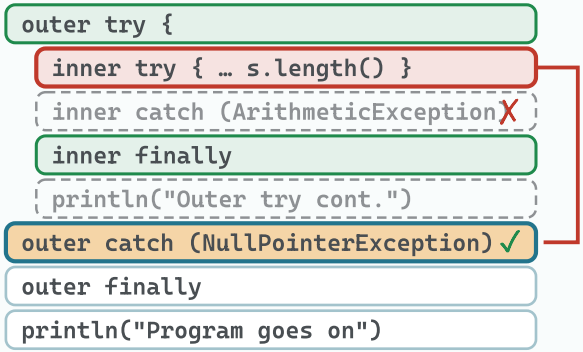
Step 5 / 8

OUTPUT

```
Outer try
Inner try
Inner finally
```

Nested try: The Exception Moves Outward

```
1  public static void main(String[] args) {
2    try {
3      System.out.println("Outer try");
4      try {
5        System.out.println("Inner try");
6        String s = null;
7        System.out.println(s.length());
8      } catch (ArithmeticException e) {
9        System.out.println("Inner catch");
10     } finally {
11       System.out.println("Inner finally");
12     }
13     System.out.println("Outer try cont.");
▶    } catch (NullPointerException e) {
▶      System.out.println("Outer catch:
   NPE");
16   } finally {
17     System.out.println("Outer finally");
18   }
19   System.out.println("Program goes on");
20 }
```



The exception reached the outer level; “Outer try cont.” was **skipped**. The outer catch matches.

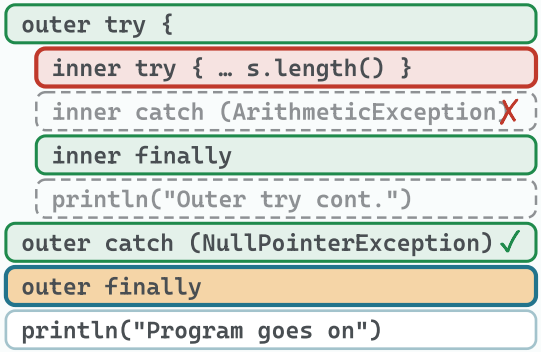
Step 6 / 8

OUTPUT

```
Outer try
Inner try
Inner finally
Outer catch: NPE
```

Nested try: The Exception Moves Outward

```
1  public static void main(String[] args) {
2    try {
3      System.out.println("Outer try");
4      try {
5        System.out.println("Inner try");
6        String s = null;
7        System.out.println(s.length());
8      } catch (ArithmeticException e) {
9        System.out.println("Inner catch");
10     } finally {
11       System.out.println("Inner finally");
12     }
13     System.out.println("Outer try cont.");
14   } catch (NullPointerException e) {
15     System.out.println("Outer catch:
   NPE");
16   } finally {
▶      System.out.println("Outer finally");
18   }
19   System.out.println("Program goes on");
20 }
```



The outer finally runs.

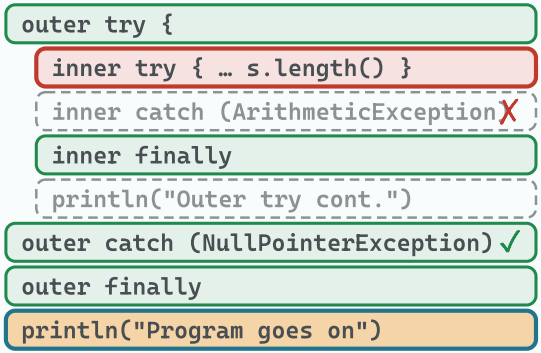
Step 7 / 8

OUTPUT

```
Outer try
Inner try
Inner finally
Outer catch: NPE
Outer finally
```

Nested try: The Exception Moves Outward

```
1  public static void main(String[] args) {
2    try {
3      System.out.println("Outer try");
4      try {
5        System.out.println("Inner try");
6        String s = null;
7        System.out.println(s.length());
8      } catch (ArithmeticException e) {
9        System.out.println("Inner catch");
10     } finally {
11       System.out.println("Inner finally");
12     }
13     System.out.println("Outer try cont.");
14   } catch (NullPointerException e) {
15     System.out.println("Outer catch:
   NPE");
16   } finally {
17     System.out.println("Outer finally");
18   }
▶    System.out.println("Program goes on");
20 }
```



The exception was handled; the program continues normally.

Step 8 / 8

OUTPUT

```
Outer try
Inner try
Inner finally
Outer catch: NPE
Outer finally
Program goes on
```

HIERARCHY

The Exception Hierarchy

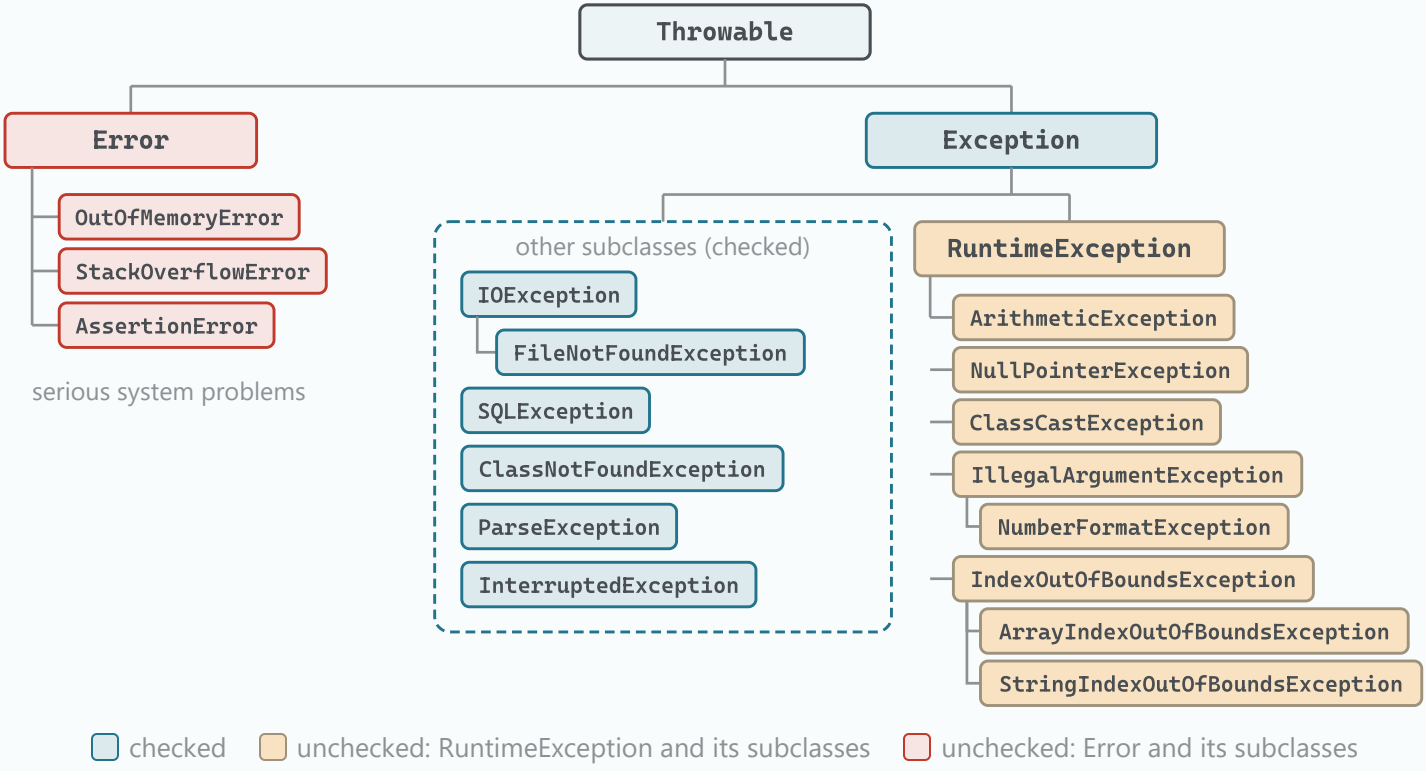
- Java's exception classes form a **class hierarchy**.
- `Throwable` is the **root** of everything that can be thrown: only objects of `Throwable` and its subclasses can be thrown with `throw` and caught with `catch`.
- `Throwable` has two direct subclasses: `Error` and `Exception`.
- A `catch` block catches the named type **and all its subclasses** (a natural result of inheritance).
- Java lets you define your own **custom exception classes**: classes for application-specific situations join this hierarchy.

Why a hierarchy?

Exceptions of one family can be caught with **a single supertype**: `catch (IOException e)` catches `FileNotFoundException` and other I/O errors alike.

HIERARCHY

The Exception Class Tree



HIERARCHY

Error and Exception

Error

- Signals that the program has hit a serious problem it **cannot recover** from.
- Such situations usually cannot be handled and typically end the program.
- Examples: extreme cases such as running out of memory (`OutOfMemoryError`), a call stack overflow, or a disk drive “vanishing”.
- Catching them in application code is **not recommended**.

Exception

+ Conditions the program can **anticipate** and often **recover** from.
+ A missing file, a dropped network connection, invalid input, a `null` reference…
+ The type you are expected to handle with `try` - `catch` .
+ Its subclass `RuntimeException` mostly represents **programmer errors**.

Both derive from `Throwable` , so both can be thrown. But the family that “should be handled” is `Exception` .

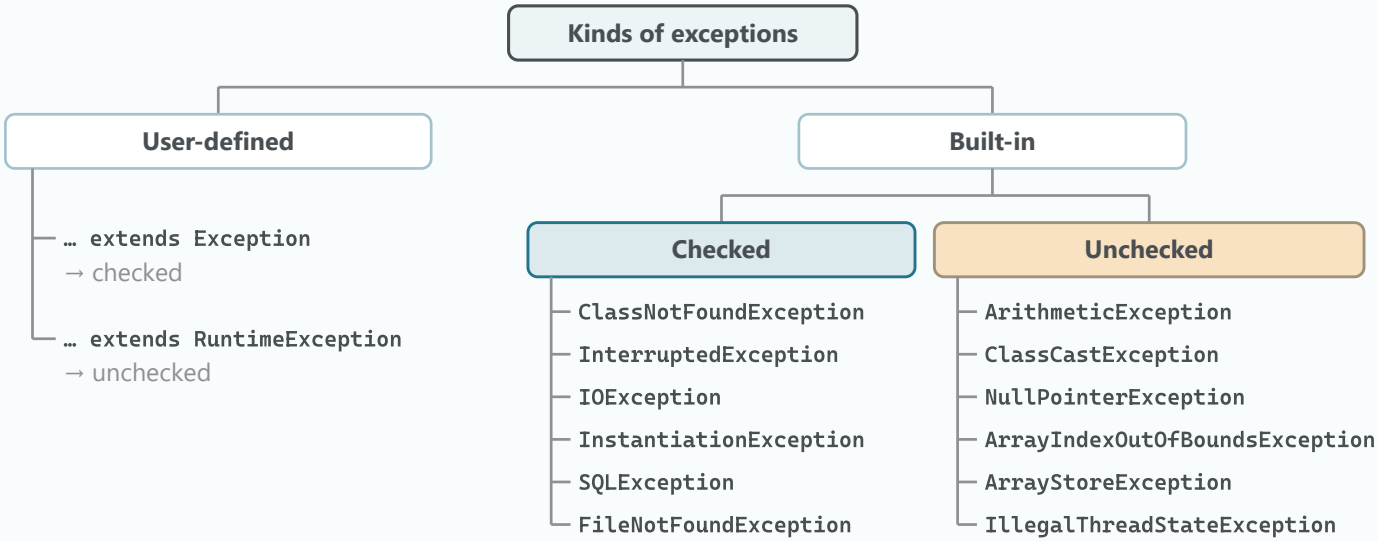
Kinds of Exceptions

Checked exceptions

Checked **by the compiler** at compile time; the programmer must handle or declare them. They cover situations known in advance to be possible.

Unchecked exceptions

Not checked at compile time; they show up at run time. `RuntimeException`, `Error` and their subclasses.



Checked Exceptions

- Must be **handled or declared** by the code: the "catch or declare" rule.
- Either list it with `throws` in the method header or handle it in a `try` - `catch`.
- Derives from `Exception` but **not** from `RuntimeException`.
- The rule is enforced **at compile time**: break it and the program won't compile.
- Example: reading a file that doesn't exist. However correct the program is, the file may have been deleted, so the compiler demands a plan.

✗ NEITHER HANDLED NOR DECLARED

```
public void readFile() {
  FileReader reader = new FileReader("data.txt");
}
```

```
error: unreported exception FileNotFoundException;
  must be caught or declared to be thrown
```

✓ DECLARED WITH THROWS

```
public void readFile() throws FileNotFoundException {
  // Read the file
  FileReader reader = new FileReader("data.txt");
}
```

Handling a Checked Exception

```
public static void readFile2() {
  try {
    FileReader reader = new FileReader("data.txt");
    // Read the file
  } catch (FileNotFoundException e) {
    // Checked exception handled here
    System.err.println("File not found: " + e.getMessage());
    // Rethrow if needed:
    // throw e;
  }
}
```

Output (Windows; on Linux: No such file or directory)

```
File not found: data.txt (The system cannot find the file specified)
```

- Here the method catches the exception **itself**; there is no `throws` in its signature. The caller doesn't have to do anything.
- `System.err` : the error output stream. Most IDEs show it in red.
- The message comes from the operating system, so it differs on Windows and Linux.
- `throw e;` : a caught exception can be **rethrown** if needed (say, after logging it). The method must then declare `throws FileNotFoundException` again.

Unchecked Exceptions

- Exceptions you are **not required** to handle or declare.
- They usually come from **logic errors** or **programmer errors**: dividing by zero, using `null`, going past an array's bounds.
- No `throws` clause or `try` - `catch` is required; the compiler stays silent.
- They derive from `RuntimeException` or `Error`.
- The right fix is usually not to catch them but to **prevent** them: check the divisor before dividing and `null` before use.

```
public void divide(int num, int den) {
  // Unchecked: ArithmeticException
  int result = num / den;
}
```

Compiles; no `throws` needed. If `den == 0`, an exception occurs at run time.

```
public void divide(int num, int den) {
  if (den == 0) {   // prevention
    System.out.println("Divisor cannot be 0");
    return;
  }
  int result = num / den;
}
```

KINDS OF EXCEPTIONS

Checked, Unchecked, Error

	Checked	Unchecked	Error
Superclass	`Exception` (except RuntimeException)	`RuntimeException`	`Error`
Checked by compiler?	**Yes**: handle or declare	**No**	**No**
Typical cause	Conditions outside the program (file, network, database)	Programmer error (null, bounds, division)	JVM / system problem
What to do	Catch and recover, or pass on with `throws`	Fix the code, validate input	Usually don't catch; end the program cleanly
Examples	`IOException`, `SQLException`, `ClassNotFoundException`	`NullPointerException`, `ArithmeticException`	`OutOfMemoryError`, `StackOverflowError`

Throwing an Exception

- When the Java runtime hits an unexpected situation, it **creates an exception object** and **throws** it to notify higher-level code.
- The programmer can do the same with the `throw` keyword.
- An exception **changes** the normal flow: the lines after `throw` don't run; control jumps to a matching `catch`.
- Many types can be thrown: `IOException`, `NullPointerException`, `IllegalArgumentException`, your own custom classes...
- The caller can catch a thrown exception with `try` - `catch`. If it is never handled, the JVM ends the program and reports **what** went wrong and **where**.

SYNTAX

```
throw new Exception("This is an exception!");
```

`new` creates the exception **object** (the message goes to the constructor); `throw` throws it.

```
try {
  throw new Exception("This is an exception!");
} catch (Exception e) {
  System.out.println(e);
}
```

Output

```
java.lang.Exception: This is an exception!
```

The Exception Type That Best Fits

```
class CustomException extends Exception {
  public CustomException(String message) {
    super(message);
  }
}
public static void check(int value)
    throws CustomException {
  if (value < 0) {
    // Check a condition and throw
    throw new CustomException("Value cannot be negative!");
  }
  // Value is OK: carry on
  System.out.println("Value OK: " + value);
}
// in main
try {
  check(5);
  check(-3);
  check(8);
} catch (CustomException e) {
  System.out.println("Caught: " + e.getMessage());
}
```

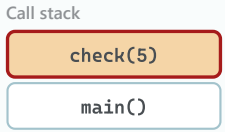
Output

```
Value OK: 5
Caught: Value cannot be negative!
```

- Choose the exception class that **best represents** the situation.
- For special scenarios you can create **custom exception classes**.
- `CustomException` is a subclass of `Exception`, so it is **checked**. That is why the method signature needs `throws CustomException`.
- Once `check(-3)` throws, the rest of try is skipped: `check(8)` is **never called**.

throw: An Exception Skips the Rest of try

```
1  static void check(int value) throws CustomException {
2    if (value < 0) {
3      throw new CustomException("Value cannot be negative!");
4    }
5    System.out.println("Value OK: " + value);
6  }
7  // in main
8  try {
▶    check(5);
10   check(-3);
11   check(8);
12 } catch (CustomException e) {
13   System.out.println("Caught: " + e.getMessage());
14 }
```



main calls `check(5)` inside try; a new frame is pushed on the stack.

Step 1 / 9

CALL STACK

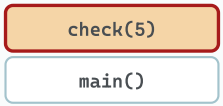
`main()`

OUTPUT

throw: An Exception Skips the Rest of try

```
1  static void check(int value) throws CustomException {
▶    if (value < 0) {
3      throw new CustomException("Value cannot be negative!");
4    }
5    System.out.println("Value OK: " + value);
6  }
7  // in main
8  try {
9    check(5);
10   check(-3);
11   check(8);
12 } catch (CustomException e) {
13   System.out.println("Caught: " + e.getMessage());
14 }
```

Call stack



`5 < 0` is false: the throw line is skipped.

Step 2 / 9

CALL STACK

check(5)

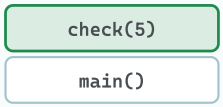
main()

OUTPUT

throw: An Exception Skips the Rest of try

```
1  static void check(int value) throws CustomException {
2    if (value < 0) {
3      throw new CustomException("Value cannot be negative!");
4    }
▶    System.out.println("Value OK: " + value);
6  }
7  // in main
8  try {
9    check(5);
10   check(-3);
11   check(8);
12 } catch (CustomException e) {
13   System.out.println("Caught: " + e.getMessage());
14 }
```

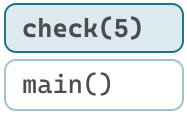
Call stack



“Value OK: 5” is printed; the method returns normally.

Step 3 / 9

CALL STACK

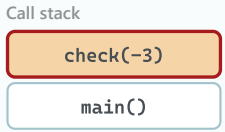


OUTPUT

```
Value OK: 5
```

throw: An Exception Skips the Rest of try

```
1  static void check(int value) throws CustomException {
2    if (value < 0) {
3      throw new CustomException("Value cannot be negative!");
4    }
5    System.out.println("Value OK: " + value);
6  }
7  // in main
8  try {
9    check(5);
   ▶ check(-3);
11   check(8);
12 } catch (CustomException e) {
13   System.out.println("Caught: " + e.getMessage());
14 }
```



Next call: `check(-3)`.

Step 4 / 9

CALL STACK

`main()`

OUTPUT

```
Value OK: 5
```

throw: An Exception Skips the Rest of try

```
1  static void check(int value) throws CustomException {
▶    if (value < 0) {
3      throw new CustomException("Value cannot be negative!");
4    }
5    System.out.println("Value OK: " + value);
6  }
7  // in main
8  try {
9    check(5);
10   check(-3);
11   check(8);
12 } catch (CustomException e) {
13   System.out.println("Caught: " + e.getMessage());
14 }
```

Call stack

check(-3)

main()

`-3 < 0` is true.

Step 5 / 9

CALL STACK

check(-3)

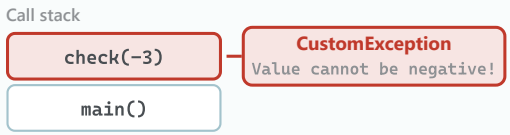
main()

OUTPUT

```
Value OK: 5
```

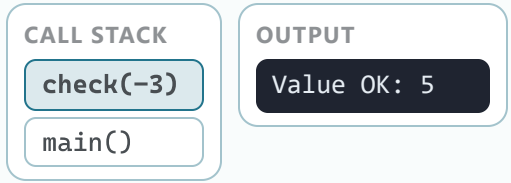
throw: An Exception Skips the Rest of try

```
1  static void check(int value) throws CustomException {
2    if (value < 0) {
▶      throw new CustomException("Value cannot be negative!");
4    }
5    System.out.println("Value OK: " + value);
6  }
7  // in main
8  try {
9    check(5);
10   check(-3);
11   check(8);
12 } catch (CustomException e) {
13   System.out.println("Caught: " + e.getMessage());
14 }
```



`new` creates the exception object and `throw` throws it. Line 5 doesn't run; the method has no catch, so its frame is discarded.

Step 6 / 9



throw: An Exception Skips the Rest of try

```
1  static void check(int value) throws CustomException {
2    if (value < 0) {
3      throw new CustomException("Value cannot be negative!");
4    }
5    System.out.println("Value OK: " + value);
6  }
7  // in main
8  try {
9    check(5);
10   check(-3);
11   check(8);
▶ } catch (CustomException e) {
13   System.out.println("Caught: " + e.getMessage());
14 }
```

Call stack

`main()`

The exception reached main; the rest of try (`check(8)`) is **skipped** and a matching catch is found.

Step 7 / 9

CALL STACK

`main()`

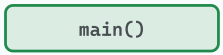
OUTPUT

```
Value OK: 5
```

throw: An Exception Skips the Rest of try

```
1  static void check(int value) throws CustomException {
2    if (value < 0) {
3      throw new CustomException("Value cannot be negative!");
4    }
5    System.out.println("Value OK: " + value);
6  }
7  // in main
8  try {
9    check(5);
10   check(-3);
11   check(8);
12 } catch (CustomException e) {
▶    System.out.println("Caught: " + e.getMessage());
14 }
```

Call stack



`getMessage()` returns the message given to the constructor.

Step 8 / 9

CALL STACK

`main()`

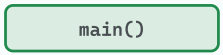
OUTPUT

```
Value OK: 5
Caught: Value
cannot be
negative!
```

throw: An Exception Skips the Rest of try

```
1  static void check(int value) throws CustomException {
2    if (value < 0) {
3      throw new CustomException("Value cannot be negative!");
4    }
5    System.out.println("Value OK: " + value);
6  }
7  // in main
8  try {
9    check(5);
10   check(-3);
11   check(8);
12 } catch (CustomException e) {
13   System.out.println("Caught: " + e.getMessage());
▶ }
```

Call stack



catch is done; the program continues its normal flow. `check(8)` was never called.

Step 9 / 9

CALL STACK

`main()`

OUTPUT

```
Value OK: 5
Caught: Value
cannot be
negative!
```

throw vs throws

	`throw`	`throws`
Purpose	**Throws** an exception (now)	**Declares** the exceptions a method may throw (possibly later)
Where	In the method body, as a statement	In the method signature, after the parameters
Followed by	A single object: `throw new X("…");`	One or more types: `throws X, Y`
Example	`throw new ArithmeticException("Division by zero");`	`void divide(int x, int y) throws ArithmeticException`

✗ THROWS CAN'T BE USED IN THE BODY

```
if (x < 0) {
  throws new IllegalArgumentException("negative");
}
```

```
error: illegal start of expression
```

✗ CHECKED EXCEPTION THROWN WITHOUT BEING DECLARED

```
static void f(int x) {
  if (x < 0) throw new Exception("negative");
}
```

```
error: unreported exception Exception;
  must be caught or declared to be thrown
```

Calling Methods That Throw

A FILE-READING METHOD

```
public void readFile() throws FileNotFoundException {
  // Read the file
}
```

1) THE CALLER HANDLES IT (PREFERRED)

```
try {
  readFile();
} catch (FileNotFoundException e) {
  // Handle the missing file
  System.err.println("File not found: " + e.getMessage());
}
```

2) THE CALLER DECLARES IT TOO (PASSES IT ON)

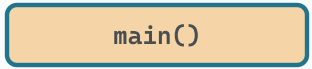
```
public void makeReport() throws FileNotFoundException {
  readFile();
}
```

- When calling a method, consider the exceptions **declared in its signature** and pick a suitable strategy.
- Put the call **inside** a suitable `try` - `catch` block.
- Or have the caller declare the same type with `throws`; the problem moves up a level. This only postpones the problem, so it is **less preferred** than catching.
- Even `main` can declare `throws`; an uncaught exception then ends the program.

Propagation: The Exception Unwinds the Call Stack

```
1  static int divide(int a, int b) {
2    System.out.println("divide started");
3    int s = a / b;
4    System.out.println("divide ended");
5    return s;
6  }
7  static int compute(int x) {
8    System.out.println("compute started");
9    int r = divide(x, 0);
10   System.out.println("compute ended");
11   return r;
12 }
13 public static void main(String[] args) {
14   try {
▶      int result = compute(10);
16     System.out.println("Result: " + result);
17   } catch (ArithmeticException e) {
18     System.out.println("main: " +
   e.getMessage());
19   }
20   System.out.println("Program ended");
21 }
```

Call stack



main calls `compute(10)` inside try.

Step 1 / 9

CALL STACK

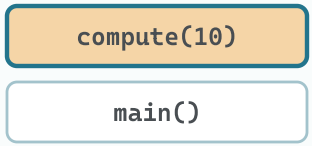


OUTPUT

Propagation: The Exception Unwinds the Call Stack

```
1  static int divide(int a, int b) {
2    System.out.println("divide started");
3    int s = a / b;
4    System.out.println("divide ended");
5    return s;
6  }
7  static int compute(int x) {
▶    System.out.println("compute started");
9    int r = divide(x, 0);
10   System.out.println("compute ended");
11   return r;
12 }
13 public static void main(String[] args) {
14   try {
15     int result = compute(10);
16     System.out.println("Result: " + result);
17   } catch (ArithmeticException e) {
18     System.out.println("main: " +
   e.getMessage());
19   }
20   System.out.println("Program ended");
21 }
```

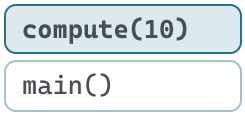
Call stack



A `compute` frame is pushed on the stack.

Step 2 / 9

CALL STACK



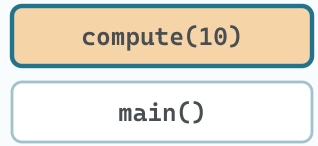
OUTPUT

```
compute started
```

Propagation: The Exception Unwinds the Call Stack

```
1  static int divide(int a, int b) {
2    System.out.println("divide started");
3    int s = a / b;
4    System.out.println("divide ended");
5    return s;
6  }
7  static int compute(int x) {
8    System.out.println("compute started");
▶    int r = divide(x, 0);
10   System.out.println("compute ended");
11   return r;
12 }
13 public static void main(String[] args) {
14   try {
15     int result = compute(10);
16     System.out.println("Result: " + result);
17   } catch (ArithmeticException e) {
18     System.out.println("main: " +
   e.getMessage());
19   }
20   System.out.println("Program ended");
21 }
```

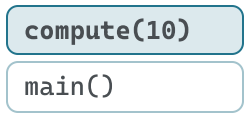
Call stack



`compute` calls `divide(10, 0)`.

Step 3 / 9

CALL STACK



OUTPUT

```
compute started
```

Propagation: The Exception Unwinds the Call Stack

```
 1 static int divide(int a, int b) {
 2   System.out.println("divide started");
 3   int s = a / b;
 4   System.out.println("divide ended");
 5   return s;
 6 }
 7 static int compute(int x) {
 8   System.out.println("compute started");
 9   int r = divide(x, 0);
10   System.out.println("compute ended");
11   return r;
12 }
13 public static void main(String[] args) {
14   try {
15     int result = compute(10);
16     System.out.println("Result: " + result);
17   } catch (ArithmeticException e) {
18     System.out.println("main: " +
   e.getMessage());
19   }
20   System.out.println("Program ended");
21 }
```

Call stack

- `divide(10, 0)`
- `compute(10)`
- `main()`

`divide` is now on top of the stack.

Step 4 / 9

CALL STACK

- `divide(10, 0)`
- `compute(10)`
- `main()`

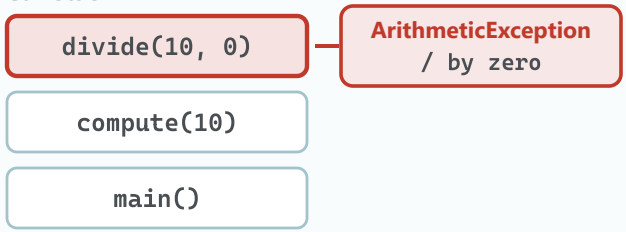
OUTPUT

```
compute started
divide started
```

Propagation: The Exception Unwinds the Call Stack

```
1  static int divide(int a, int b) {
2    System.out.println("divide started");
▶    int s = a / b;
4    System.out.println("divide ended");
5    return s;
6  }
7  static int compute(int x) {
8    System.out.println("compute started");
9    int r = divide(x, 0);
10   System.out.println("compute ended");
11   return r;
12 }
13 public static void main(String[] args) {
14   try {
15     int result = compute(10);
16     System.out.println("Result: " + result);
17   } catch (ArithmeticException e) {
18     System.out.println("main: " +
   e.getMessage());
19   }
20   System.out.println("Program ended");
21 }
```

Call stack



`10 / 0` : the exception is born in **divide**. divide has no try → the rest of divide doesn't run and its frame is **discarded**.

Step 5 / 9

CALL STACK

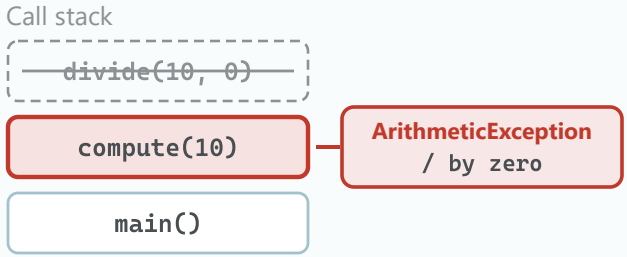
- `divide(10, 0)`
- `compute(10)`
- `main()`

OUTPUT

```
compute started
divide started
```

Propagation: The Exception Unwinds the Call Stack

```
1  static int divide(int a, int b) {
2    System.out.println("divide started");
3    int s = a / b;
4    System.out.println("divide ended");
5    return s;
6  }
7  static int compute(int x) {
8    System.out.println("compute started");
▶    int r = divide(x, 0);
10   System.out.println("compute ended");
11   return r;
12 }
13 public static void main(String[] args) {
14   try {
15     int result = compute(10);
16     System.out.println("Result: " + result);
17   } catch (ArithmeticException e) {
18     System.out.println("main: " +
   e.getMessage());
19   }
20   System.out.println("Program ended");
21 }
```



The exception passes to the caller, `compute`.
No try here either → “compute ended” is skipped and the frame is discarded.

Step 6 / 9

CALL STACK

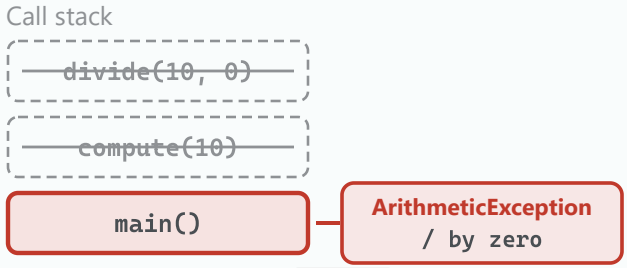
- compute(10)
- main()

OUTPUT

```
compute started
divide started
```

Propagation: The Exception Unwinds the Call Stack

```
1  static int divide(int a, int b) {
2    System.out.println("divide started");
3    int s = a / b;
4    System.out.println("divide ended");
5    return s;
6  }
7  static int compute(int x) {
8    System.out.println("compute started");
9    int r = divide(x, 0);
10   System.out.println("compute ended");
11   return r;
12 }
13 public static void main(String[] args) {
14   try {
▶      int result = compute(10);
16     System.out.println("Result: " + result);
17   } catch (ArithmeticException e) {
18     System.out.println("main: " +
   e.getMessage());
19   }
20   System.out.println("Program ended");
21 }
```



The exception reaches `main`. The call is inside a `try`, so a matching catch is sought. The “Result” line is skipped.

Step 7 / 9

CALL STACK

`main()`

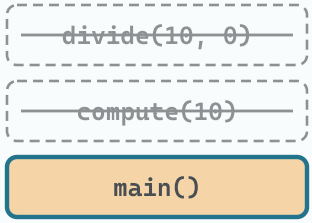
OUTPUT

```
compute started
divide started
```

Propagation: The Exception Unwinds the Call Stack

```
 1 static int divide(int a, int b) {
 2   System.out.println("divide started");
 3   int s = a / b;
 4   System.out.println("divide ended");
 5   return s;
 6 }
 7 static int compute(int x) {
 8   System.out.println("compute started");
 9   int r = divide(x, 0);
10   System.out.println("compute ended");
11   return r;
12 }
13 public static void main(String[] args) {
14   try {
15     int result = compute(10);
16     System.out.println("Result: " + result);
▶    } catch (ArithmeticException e) {
▶      System.out.println("main: " +
e.getMessage());
19   }
20   System.out.println("Program ended");
21 }
```

Call stack



catch matches: the exception is **caught** and propagation stops.

Step 8 / 9

CALL STACK

`main()`

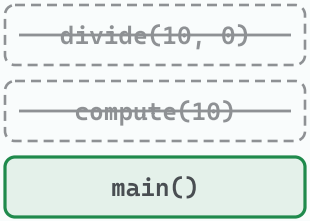
OUTPUT

```
compute started
divide started
main: / by zero
```

Propagation: The Exception Unwinds the Call Stack

```
1  static int divide(int a, int b) {
2    System.out.println("divide started");
3    int s = a / b;
4    System.out.println("divide ended");
5    return s;
6  }
7  static int compute(int x) {
8    System.out.println("compute started");
9    int r = divide(x, 0);
10   System.out.println("compute ended");
11   return r;
12 }
13 public static void main(String[] args) {
14   try {
15     int result = compute(10);
16     System.out.println("Result: " + result);
17   } catch (ArithmeticException e) {
18     System.out.println("main: " +
   e.getMessage());
19   }
▶    System.out.println("Program ended");
21 }
```

Call stack



The program ends normally. Had nobody caught it, the JVM would have ended the program.

Step 9 / 9

CALL STACK

`main()`

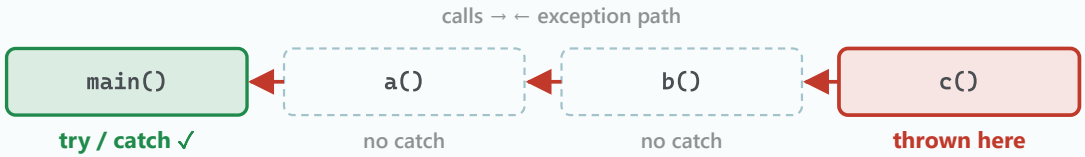
OUTPUT

```
compute started
divide started
main: / by zero
Program ended
```

How Does the JVM Handle an Exception?

1. The JVM (or `throw`) creates an **exception object** holding its type, its message and a stack trace of where it occurred.
2. In the method where it occurred, the JVM looks for an enclosing `try` with a **compatible** `catch`.
3. If none is found, the method ends at once (its `finally` runs, if any) and the exception passes to the **calling method**. The search continues down the call stack.
4. If a matching catch is found, it runs; the program continues after the catch.
5. If `main` doesn't catch it either, the JVM's **default handler** prints the stack trace and ends the program (that thread).

This search is called **stack unwinding**. Once the catch block ends, the caught exception object is no longer used and is cleaned up by the garbage collector.



RuntimeException Classes

`RuntimeException` and its subclasses represent exceptions caused by **programmer errors** or **logic errors**. They are not checked at compile time and need not be handled; still, the programmer should **deliberately** prevent or handle them.

Class	When does it occur?	Example
`NullPointerException`	Using a member of a `null` reference	`String s = null; s.length();`
`ArrayIndexOutOfBoundsException`	Index outside the array bounds	`int[] a = new int[3]; a[5] = 1;`
`ArithmeticException`	Integer division by zero, overflow in exact arithmetic	`10 / 0`
`IllegalArgumentException`	A method gets an invalid argument	`new ArrayList<>(-1)`
`NumberFormatException`	Converting text that isn't a valid number	`Integer.parseInt("abc")`
`ClassCastException`	Casting an object to a type it doesn't have	`(Integer) obj`

NullPointerException

- Calling a method, accessing a field or taking `length` through a `null` reference causes this exception.
- One of the most common bugs. To prevent it:
  - An explicit check: `if (str != null)`
  - `Objects.requireNonNull(x, msg)`: if `null`, throws an NPE **immediately** with a meaningful message
  - `Objects.requireNonNullElse(x, fallback)` (Java 9+): returns the fallback if `null`
  - The `Optional` class (Java 8+): explicitly represents "there may be no value"
  - Libraries: Apache Commons Lang `StringUtils` offers `null`-safe string operations

```
String str = null;
int length = str.length();
```

Output (Java 14+ helpful message)

```
Exception in thread "main"
java.lang.NullPointerException:
  Cannot invoke "String.length()" because
"str" is null
```

Since Java 14 the message says **which** reference was `null`. The variable name appears when compiled with debug info (`javac -g`, IDEs); otherwise you see `"<local1>"`.

Working Safely with null

```
1  String str = null;
2  int length = str.length();
3  Objects.requireNonNull(str, "str must not be null");
4  Optional<String> optionalStr = Optional.ofNullable(str);
5  length = optionalStr.map(String::length).orElse(0);
6  String defaultValue = Objects.requireNonNullElse(str, "0");
7  if (str != null) {
8    System.out.println(str.length());
9  }
10 System.out.println("Ali".equals(str));
```

Line	Result (each line tried on its own)
2	**NullPointerException**: Cannot invoke "String.length()" because "str" is null
3	**NullPointerException**: str must not be null
5	**length = 0** (no value → `orElse(0)`)
6	**defaultValue = "0"**
7–9	Condition false; nothing printed, no exception
10	**false**: the literal comes first, so a `null` argument is fine

`"Ali".equals(str)` is safe; `str.equals("Ali")` throws an NPE when `str` is `null`. Use `requireNonNull` at the start of methods to catch the error **at its source** with a clear message.

RUNTIME EXCEPTIONS

ArrayIndexOutOfBoundsException

- Accessing an index outside the array's bounds: valid indexes are `0` … `length - 1`; a **negative** index is invalid too.
- A member of the `IndexOutOfBoundsException` family. Different classes of that family may be thrown:
  - array (`int[]`), `Vector`, an `Arrays.asList(...)` list → `ArrayIndexOutOfBoundsException`
  - `ArrayList.get` → `IndexOutOfBoundsException` (the superclass)
  - `String.charAt` → `StringIndexOutOfBoundsException`
- To catch them all: `catch (IndexOutOfBoundsException e)`.

PREVENTION: BOUND THE LOOP WITH `length`

```
int[] numbers = {1, 2, 3};
for (int i = 0; i < numbers.length; i++) {
  System.out.print(numbers[i] + " ");
}
```

Output

```
1 2 3
```

Most common cause: writing `i <= arr.length`. The last pass accesses `arr[length]` and throws.

Out of Bounds: Which Line, Which Exception?

```
1  int[] numbers = {1, 2, 3};
2  int value = numbers[5];
3  ArrayList<String> list = new ArrayList<>();
4  String svalue = list.get(5);
5  Vector<Integer> vector = new Vector<>();
6  value = vector.get(10);
7  String[] names = {"Alice", "Bob", "Charlie"};
8  svalue = Arrays.asList(names).get(5);
9  String str = "Hello";
10 char ch = str.charAt(10);
```

Line	Exception thrown and its message
2	`ArrayIndexOutOfBoundsException` Index 5 out of bounds for length 3
4	`IndexOutOfBoundsException` Index 5 out of bounds for length 0
6	`ArrayIndexOutOfBoundsException` Array index out of range: 10
8	`ArrayIndexOutOfBoundsException` Index 5 out of bounds for length 3
10	`StringIndexOutOfBoundsException` Index 10 out of bounds for length 5

Each line tried on its own. An empty `ArrayList` may have capacity, but its **size** is 0; `get` checks against the size.

ArithmeticException

- Represents an error during a mathematical operation; the best-known case is **integer division by zero** (`/` and `%`).
- The "exact" methods of `Math` throw on overflow: `addExact` , `multiplyExact` , `toIntExact` ...
- `BigDecimal` provides high-precision arithmetic; it throws on division by zero or when the quotient can't be represented exactly (1/3).
- `double` operations don't throw: `10.0 / 0` → `Infinity` , `Math.sqrt(-1)` → `NaN` .
- `Random.nextInt(0)` throws `IllegalArgumentException` , not `ArithmeticException` : the bound must be positive.

Silent overflow: when an ordinary `int` addition overflows, **no** exception occurs; the value wraps around: `Integer.MAX_VALUE + 1` → `-2147483648` . If overflow must be detected, use `Math.addExact` .

RUNTIME EXCEPTIONS

Arithmetic Errors: Line by Line

```
1  int result = 10 / 0;
2  double dresult = Math.sqrt(-1);
3  System.out.println(dresult);
4  System.out.println(10.0 / 0);
5  int t = Math.addExact(Integer.MAX_VALUE, 1);
6  Random random = new Random();
7  int randomNumber = random.nextInt(0);
8  BigDecimal num1 = new BigDecimal("10");
9  BigDecimal num2 = BigDecimal.ZERO;
10 BigDecimal bresult = num1.divide(num2);
```

Line	Result
1	`ArithmeticException` : / by zero
3	Prints **NaN**, no exception
4	Prints **Infinity**, no exception
5	`ArithmeticException` : integer overflow
7	`IllegalArgumentException` : bound must be positive
10	`ArithmeticException` : Division by zero

Each line tried on its own. With floating-point numbers, division by zero yields special values (`Infinity` , `NaN`) under IEEE 754 rules, so `double` results need their own checks (`Double.isNaN`).

IllegalArgumentException

- Signals that a method received an **invalid argument**. It is an unchecked exception.
- Checking arguments and throwing stops the error **at its source**.
- Library examples:
  - `Arrays.copyOfRange(d, 5, 2)` : start > end
  - `new ArrayList<>(-1)` : negative capacity
  - `setPriority(11)` on a `Thread` : priority must be 1–10
  - `new File(uri)` : the URI is not a `file:` URI

```
public void setAge(int age) {
  if (age < 0) {
    throw new IllegalArgumentException("Age cannot be negative");
  }
  this.age = age;
}
public void setName(String name) {
  if (name.length() < 3) {
    throw new IllegalArgumentException("Min 3 characters");
  }
  this.name = name;
}
```

Not every invalid case throws this: `new FileReader("invalid/file/path")` → `FileNotFoundException` ; `new Thread(null)` throws nothing. Subclass: `NumberFormatException` .

RUNTIME EXCEPTIONS

Illegal Argument: Line by Line

```
1 setAge(-5);
2 setName("Al");
3 int[] numbers = {1, 2, 3};
4 Arrays.copyOfRange(numbers, 5, 2);
5 List<Integer> list = new ArrayList<>(-1);
6 File file = new File(URI.create("http://example.com/a.txt"));
7 Thread thread = new Thread(task);
8 thread.setPriority(11);
```

Line	Exception thrown (each line on its own)
1	IllegalArgumentException: Age cannot be negative
2	IllegalArgumentException: Min 3 characters
4	IllegalArgumentException: 5 > 2
5	IllegalArgumentException: Illegal Capacity: -1
6	IllegalArgumentException: URI scheme is not "file"
8	IllegalArgumentException (no message)

A good message states the invalid value and what was expected: `"Age cannot be negative: " + age`. An exception with no message (line 8) makes debugging harder.

NumberFormatException

- Occurs when text being converted to a number **doesn't represent a valid number**.
- `Integer.parseInt`, `Double.parseDouble`, `Float.parseFloat`, `Short.parseShort`, `Long.parseLong`, `Byte.parseByte` and the `valueOf` methods can throw it.
- It also occurs when the value is outside the type's range: `Byte.parseByte("300")`.
- A subclass of `IllegalArgumentException`.
- User input always carries this risk: validate it first or catch the exception.

```
String strNumber = "abc";
int number = Integer.parseInt(strNumber);
strNumber = "123abc";
number = Integer.parseInt(strNumber);
String strDouble = "12.34abc";
double doubleValue = Double.parseDouble(strDouble);
String strShort = "45xyz";
short shortValue = Short.parseShort(strShort);
```

Call	NumberFormatException message
`Integer.parseInt("abc")`	For input string: "abc"
`Integer.parseInt("123abc")`	For input string: "123abc"
`Double.parseDouble("12.34abc")`	For input string: "12.34abc"
`Short.parseShort("45xyz")`	For input string: "45xyz"

Safe Number Parsing

```
static int safeParse(String text, int fallback) {
  try {
    return Integer.parseInt(text.trim());
  } catch (NumberFormatException e) {
    return fallback;
  }
}
// in main
System.out.println(safeParse(" 42 ", 0));
System.out.println(safeParse("forty", 0));
System.out.println(safeParse("10.5", -1));
```

Output

```
42
0
-1
```

- Parsing is checked in **one place**, a helper method; invalid input yields a default value.
- `trim()` removes leading and trailing spaces: `parseInt(" 42 ")` alone would throw.
- `"10.5"` is not an `int`; for decimals use `Double.parseDouble`.
- Pick a default that can't be confused with a valid result; if it can, passing the exception to the caller is the better choice.

parseInt and catch: Three Inputs

```
1  static int safeParse(String text, int fallback) {
2    try {
3      return Integer.parseInt(text.trim());
4    } catch (NumberFormatException e) {
5      return fallback;
6    }
7  }
▶ System.out.println(safeParse(" 7 ", 0));
9  System.out.println(safeParse("seven", 0));
10 System.out.println(safeParse("7.5", -1));
```

text



First call: text " 7 " (spaces at both ends), fallback 0.

Step 1 / 8

VARIABLES

—

OUTPUT

parseInt and catch: Three Inputs

```
1  static int safeParse(String text, int fallback) {
2    try {
▶      return Integer.parseInt(text.trim());
4    } catch (NumberFormatException e) {
5      return fallback;
6    }
7  }
8  System.out.println(safeParse(" 7 ", 0));
9  System.out.println(safeParse("seven", 0));
10 System.out.println(safeParse("7.5", -1));
```



parseInt("7") = 7 → return 7

`trim()` drops the spaces; "7" becomes a number and 7 is returned from inside try.

Step 2 / 8

VARIABLES

`text = " 7 "` `fallback = 0`

OUTPUT

```
7
```

parseInt and catch: Three Inputs

```
1  static int safeParse(String text, int fallback) {
2    try {
3      return Integer.parseInt(text.trim());
4    } catch (NumberFormatException e) {
5      return fallback;
6    }
7  }
8  System.out.println(safeParse(" 7 ", 0));
▶  System.out.println(safeParse("seven", 0));
10 System.out.println(safeParse("7.5", -1));
```

text



Second call: "seven".

Step 3 / 8

VARIABLES

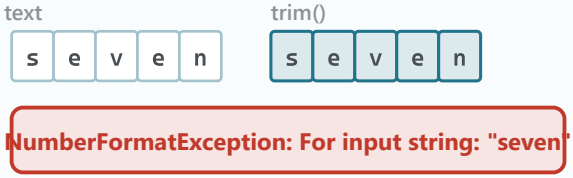
—

OUTPUT

```
7
```

parseInt and catch: Three Inputs

```
1  static int safeParse(String text, int fallback) {
2    try {
▶      return Integer.parseInt(text.trim());
4    } catch (NumberFormatException e) {
5      return fallback;
6    }
7  }
8  System.out.println(safeParse(" 7 ", 0));
9  System.out.println(safeParse("seven", 0));
10 System.out.println(safeParse("7.5", -1));
```



`parseInt` throws on text that isn't digits; `return` never ran.

Step 4 / 8

VARIABLES

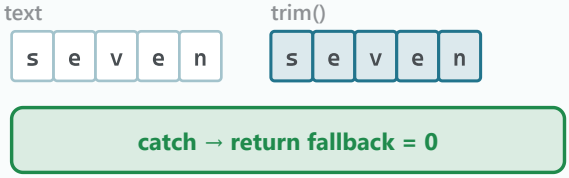
`text = "seven"` `fallback = 0`

OUTPUT

```
7
```

parseInt and catch: Three Inputs

```
1  static int safeParse(String text, int fallback) {
2    try {
3      return Integer.parseInt(text.trim());
▶    } catch (NumberFormatException e) {
▶      return fallback;
6    }
7  }
8  System.out.println(safeParse(" 7 ", 0));
9  System.out.println(safeParse("seven", 0));
10 System.out.println(safeParse("7.5", -1));
```



catch matches: the fallback 0 is returned. The exception doesn't leave the method.

Step 5 / 8

VARIABLES

`text = "seven"` `fallback = 0`

`e = NumberFormatException`

OUTPUT

```
7
0
```

parseInt and catch: Three Inputs

```
1 static int safeParse(String text, int fallback) {
2   try {
3     return Integer.parseInt(text.trim());
4   } catch (NumberFormatException e) {
5     return fallback;
6   }
7 }
8 System.out.println(safeParse(" 7 ", 0));
9 System.out.println(safeParse("seven", 0));
▶ System.out.println(safeParse("7.5", -1));
```

text



Third call: "7.5", fallback −1.

Step 6 / 8

VARIABLES

—

OUTPUT

```
7
0
```

parseInt and catch: Three Inputs

```
1  static int safeParse(String text, int fallback) {
2    try {
▶      return Integer.parseInt(text.trim());
4    } catch (NumberFormatException e) {
5      return fallback;
6    }
7  }
8  System.out.println(safeParse(" 7 ", 0));
9  System.out.println(safeParse("seven", 0));
10 System.out.println(safeParse("7.5", -1));
```



NumberFormatException: For input string: "7.5"

A dot is not a digit: "7.5" is not a valid `int`.

Step 7 / 8

VARIABLES

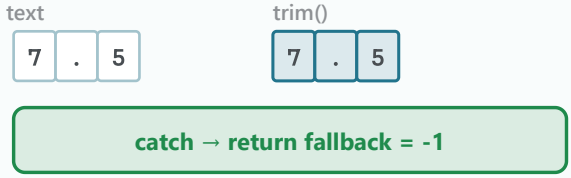
`text = "7.5"` `fallback = -1`

OUTPUT

```
7
0
```

parseInt and catch: Three Inputs

```
1  static int safeParse(String text, int fallback) {
2    try {
3      return Integer.parseInt(text.trim());
▶    } catch (NumberFormatException e) {
▶      return fallback;
6    }
7  }
8  System.out.println(safeParse(" 7 ", 0));
9  System.out.println(safeParse("seven", 0));
10 System.out.println(safeParse("7.5", -1));
```



−1 is returned. The caller carries on with the fallback instead of an exception.

Step 8 / 8

VARIABLES

`text = "7.5"` `fallback = -1`

`e = NumberFormatException`

OUTPUT

```
7
0
-1
```

ClassCastException

- Occurs when you cast an object to a type it **is not**.
- What counts is the **object's actual type**, not the variable's: `Object obj = "Hello"` is a `String` object.
- Raw collections (`ArrayList` , `Map` , `Set`) accept anything; the cast fails when an element of the wrong type is **retrieved**. Sorted collections (`TreeSet` , `TreeMap`) also throw when an incomparable element is **added**.
- Prevention: **generics** (`ArrayList<String>`) catch the wrong type **at compile time**; check with `instanceof` before casting.

```
1  Object obj = "Hello";
2  Integer number = (Integer) obj;
3  ArrayList list = new ArrayList();
4  list.add("Hello");
5  number = (Integer) list.get(0);
6  Map map = new HashMap();
7  map.put("key", "Value");
8  Integer value = (Integer) map.get("key");
9  Set set = new HashSet();
10 set.add("Hello");
11 value = (Integer) set.iterator().next();
12 Object[] objArray = new String[5];
13 Integer[] intArray = (Integer[]) objArray;
```

Lines 2, 5, 8, 11 and 13 each throw `ClassCastException` , for example:

```
class java.lang.String cannot be cast to class
java.lang.Integer
```

instanceof Before Casting

CLASSIC FORM

```
Object obj = "Hello";
if (obj instanceof Integer) {
  Integer num = (Integer) obj;
  System.out.println("Number: " + num);
} else {
  System.out.println("obj is not an Integer");
}
```

Output

```
obj is not an Integer
```

PATTERN MATCHING (JAVA 16+)

```
if (obj instanceof Integer num) {
  System.out.println("Number: " + num);
}
```

WITH GENERICS, AT COMPILE TIME

```
ArrayList<String> list = new ArrayList<>();
list.add("Hello");
Integer n = (Integer) list.get(0);
```

```
error: incompatible types: String cannot be
  converted to Integer
```

A **compile error** instead of a runtime exception: the bug is found earlier, without running the program. That's why generic types are preferred over raw types.

Checked Exception Classes

They represent situations, mostly caused by conditions **outside** the program, that the code must handle and **recover** from properly. The compiler requires them to be handled.

Class	Package	When?
`FileNotFoundException`	`java.io`	The file to open doesn't exist or can't be opened
`IOException`	`java.io`	General input/output error (superclass)
`SQLException`	`java.sql`	Database access error
`ParseException`	`java.text`	Error while parsing text against a format
`ClassNotFoundException`	`java.lang`	A class to be loaded by name was not found
`InterruptedException`	`java.lang`	A waiting thread was interrupted (`Thread.sleep`)
`NotSerializableException`	`java.io`	An attempt to write a non-serializable object

CHECKED CLASSES

FileNotFoundException

```
try {
  FileReader fileReader = new FileReader("file.txt");
} catch (FileNotFoundException e) {
  // Handle the missing file
  System.out.println(e);
}
```

Output (no file, Windows)

```
java.io.FileNotFoundException: file.txt
  (The system cannot find the file specified)
```

- Occurs when a file to be opened **cannot be found** (or is a folder, or can't be read).
- A subclass of `IOException`; `catch (IOException e)` catches it too.
- A relative path (`"file.txt"`) is resolved against the program's **working directory**. In IDEs this is usually the project folder, so the file may be "in the wrong folder".
- Typical response: tell the user and ask for another name, or carry on with defaults.

CHECKED CLASSES

IOException

```
try {
  BufferedReader reader =
    new BufferedReader(new FileReader(path));
  String line;
  while ((line = reader.readLine()) != null) {
    System.out.println(line);
  }
  reader.close();
} catch (IOException e) {
  // Handle the I/O error
  e.printStackTrace();
}
```

Output (when notes.txt exists)

```
first line
second line
```

- Represents a **general** I/O error; it has many subclasses such as `FileNotFoundException`.
- **Problem**: if `readLine()` throws, `reader.close()` never runs; the file stays open (a resource leak).

✓ WITH TRY-WITH-RESOURCES

```
try (BufferedReader reader =
    new BufferedReader(new FileReader(path))) {
  String line;
  while ((line = reader.readLine()) != null) {
    System.out.println(line);
  }
} catch (IOException e) {
  System.out.println(e.getMessage());
}
```

CHECKED CLASSES

SQLException

```
try {
  Connection connection =
    DriverManager.getConnection("jdbc:mysql://localhost/school");
  PreparedStatement preparedStatement =
    connection.prepareStatement("SELECT user_id FROM users");
  // Run the query and process the results
  ResultSet resultSet = preparedStatement.executeQuery();
  while (resultSet.next()) {
    // Process the results
    System.out.println("User ID: " + resultSet.getInt("user_id"));
  }
  // Close the connection
  connection.close();
} catch (SQLException e) {
  // Handle the SQLException
  System.out.println(e);
}
```

- Thrown when a database operation fails: no connection, bad SQL syntax, missing table, missing driver…
- `getMessage()`, `getSQLState()` and `getErrorCode()` give details.

Output (no MySQL driver)

```
java.sql.SQLException: No
suitable driver
found for
jdbc:mysql://localhost/school
```

A connection is a resource too; close it with try-with-resources.

ParseException and ClassNotFoundException

`ParseException` (`java.text`)

Thrown when text doesn't match the expected **format** during parsing.

```
try {
  SimpleDateFormat form =
    new SimpleDateFormat("dd/MM/yy");
  Date date = form.parse("01-01-2022");
} catch (ParseException e) {
  // Handle the parse error
  System.out.println(e);
}
```

Output

```
java.text.ParseException: Unparseable date:
"01-01-2022"
```

The format expects `/` ; the text has `-` .

`ClassNotFoundException`

Thrown when a class to be loaded **by name** (`Class.forName`) can't be found.

```
try {
  Class.forName("java.util.Scanner");
  System.out.println("Scanner loaded");
  // Try to load a missing class
  Class.forName("com.example.NonExistentClass");
} catch (ClassNotFoundException e) {
  System.out.println(e);
}
```

Output

```
Scanner loaded
java.lang.ClassNotFoundException:
  com.example.NonExistentClass
```

CHECKED CLASSES

NotSerializableException

```
class Point {                // not Serializable
  int x = 3, y = 4;
}
// in main
try (ObjectOutputStream outputStream =
    new ObjectOutputStream(new ByteArrayOutputStream())) {
  // Try to serialize the object
  Point notSerializableObj = new Point();
  outputStream.writeObject(notSerializableObj);
} catch (NotSerializableException e) {
  System.out.println(e);
} catch (IOException e) {
  System.out.println("Other I/O error: " + e);
}
```

Output

```
java.io.NotSerializableException: Point
```

- **Serialization**: converting an object to a byte stream (to write it to a file or send it over a network) and restoring it later.
- A class that implements `Serializable` is serializable; writing an object of a class that doesn't throws this exception.
- An (indirect) subclass of `IOException`, so its catch goes **before** the `IOException` catch.

✓ FIX

```
class Point implements Serializable {
  int x = 3, y = 4;
}
```

Error Classes

- Unchecked types for **system-level** problems that **fundamentally** affect the program's normal operation.
- They usually stem not from a line of application code but from the state of the JVM and its environment.
- Thrown by the JVM; handling or declaring them is **not recommended**. Letting the program end cleanly is the better approach.
- Still, since they are `Throwable`s, they can technically be caught.

Error	When?
`OutOfMemoryError`	Heap memory is exhausted
`StackOverflowError`	The call stack overflowed: usually infinite recursion
`AssertionError`	An `assert` condition is false (only with `-ea`)
`NoClassDefFoundError`	A class present at compile time is missing at run time



Error Examples

```java
static void recurse() {
  recurse();          // no base case
}
public static void main(String[] args) {
  try {
    recurse();
  } catch (StackOverflowError e) {
    System.out.println("SOE -> " + e);
  }
  try {
    long[] arr = new long[Integer.MAX_VALUE];
  } catch (OutOfMemoryError e) {
    System.out.println("OOM -> " + e);
  }
  int age = -5;
  assert age >= 0 : "Negative age: " + age;
  System.out.println("assert passed");
}
```

java Demo

```
SOE -> java.lang.StackOverflowError
OOM -> java.lang.OutOfMemoryError:
Requested array size exceeds VM limit
assert passed
```

java -ea Demo

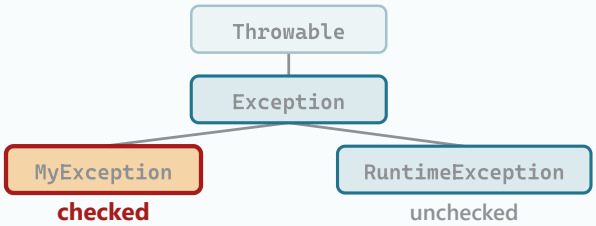
```
SOE -> java.lang.StackOverflowError
OOM -> java.lang.OutOfMemoryError:
Requested array size exceeds VM limit
Exception in thread "main"
java.lang.AssertionError: Negative age: -5
```

- If recursion has no base case, every call adds a new frame until the stack is full.
- `assert` is **disabled** by default; enable it with `-ea`. It checks assumptions during development. (Catching here is for demonstration only.)

Custom Exception Classes

```
public class MyException extends Exception {
  // Constructor
  public MyException(String message) {
    super(message);
  }
}
```

Superclass	Result
`extends Exception`	Checked: the caller must handle it; recoverable business-rule violations
`extends RuntimeException`	Unchecked: misuse, programmer error



- Developers can create **custom exception classes** to manage unexpected situations specific to their programs.
- An object of any class extending `Throwable` can be thrown. Usually, though, you extend `Exception` or `RuntimeException`, not `Error` or `Throwable` directly.
- The constructor passes the message up with `super(message)`, so `getMessage()` works.
- End the name with `…Exception`; it should describe the situation.
- You can add extra fields (a missing amount, a bad value): an exception is an ordinary class.

Throwing and Catching a Custom Exception

```
public class MyExceptionTest {
  public static void main(String[] args) {
    try {
      doWork();
    } catch (MyException e) {
      System.out.println("Caught: " + e.getMessage());
    }
  }
  public static void doWork() throws MyException {
    if (true) {
      throw new MyException("Custom error occurred.");
    }
  }
}
```

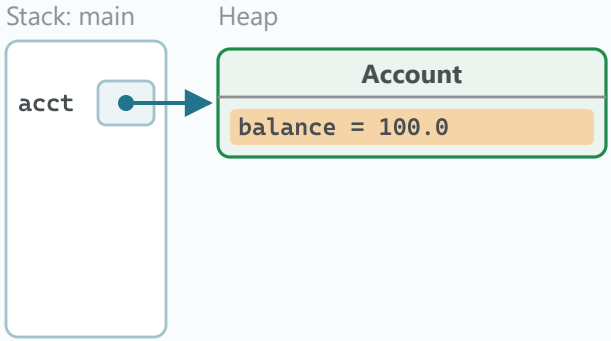
Output

```
Caught: Custom error occurred.
```

- `MyException` extends `Exception`, so it is checked: `throws MyException` is **required** in the signature of `doWork()`.
- `main` wraps the call in `try`, choosing to "handle" it.
- `if (true)` is only for the demo: code after an unconditional `throw` would cause an "unreachable statement" error.
- In real programs the condition checks a **business rule**: is the balance enough, is the grade within 0–100, is the user name empty…

Custom Exception: A Bank Account

```
1  class Account {
2    private double balance;
3    Account(double balance) { this.balance =
   balance; }
4    public void withdraw(double amount)
5        throws LowBalanceException {
6      if (amount > balance) {
7        throw new LowBalanceException(
8          "Insufficient funds", amount - balance);
9      }
10     balance -= amount;
11     System.out.println("Balance: " + balance);
12   }
13 }
14 // in main
▶ Account acct = new Account(100);
16 try {
17   acct.withdraw(30);
18   acct.withdraw(90);
19   acct.withdraw(10);
20 } catch (LowBalanceException e) {
21   System.out.println(e.getMessage()
22     + ", deficit: " + e.getDeficit());
23 }
```



The Account object is created on the heap; `acct` refers to it.

Step 1 / 7

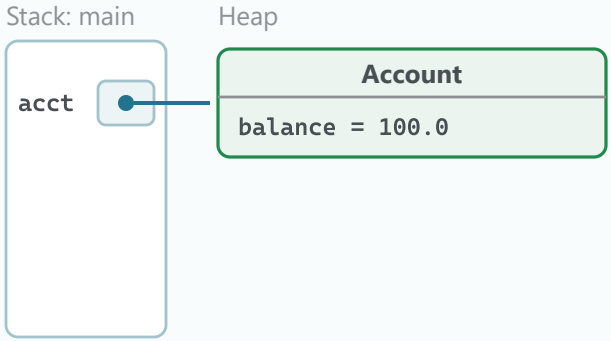
CALL STACK

`main()`

OUTPUT

Custom Exception: A Bank Account

```
1  class Account {
2    private double balance;
3    Account(double balance) { this.balance =
   balance; }
4    public void withdraw(double amount)
5        throws LowBalanceException {
6      if (amount > balance) {
7        throw new LowBalanceException(
8          "Insufficient funds", amount - balance);
9      }
10     balance -= amount;
11     System.out.println("Balance: " + balance);
12   }
13 }
14 // in main
15 Account acct = new Account(100);
16 try {
17   acct.withdraw(30);
18   acct.withdraw(90);
19   acct.withdraw(10);
20 } catch (LowBalanceException e) {
21   System.out.println(e.getMessage()
22     + ", deficit: " + e.getDeficit());
23 }
```



`withdraw(30)` : is 30 > 100? **No**; no exception.

Step 2 / 7

CALL STACK

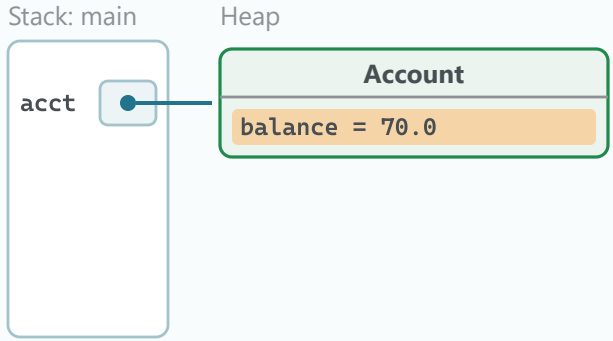
`withdraw(30.0)`

`main()`

OUTPUT

Custom Exception: A Bank Account

```
1  class Account {
2    private double balance;
3    Account(double balance) { this.balance =
   balance; }
4    public void withdraw(double amount)
5        throws LowBalanceException {
6      if (amount > balance) {
7        throw new LowBalanceException(
8          "Insufficient funds", amount - balance);
9      }
▶      balance -= amount;
▶      System.out.println("Balance: " + balance);
12   }
13 }
14 // in main
15 Account acct = new Account(100);
16 try {
17   acct.withdraw(30);
18   acct.withdraw(90);
19   acct.withdraw(10);
20 } catch (LowBalanceException e) {
21   System.out.println(e.getMessage()
22     + ", deficit: " + e.getDeficit());
23 }
```



The balance becomes 70.0 and is printed.

Step 3 / 7

CALL STACK

withdraw(30.0)

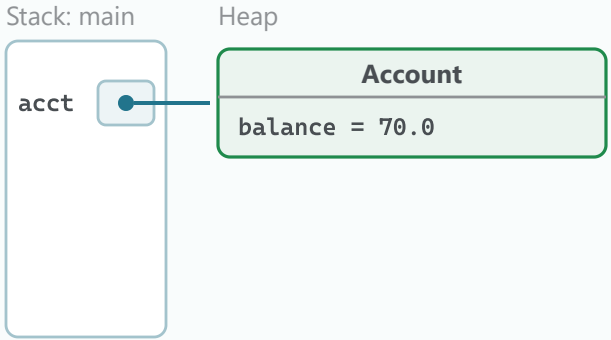
main()

OUTPUT

```
Balance: 70.0
```

Custom Exception: A Bank Account

```
1  class Account {
2    private double balance;
3    Account(double balance) { this.balance =
   balance; }
4    public void withdraw(double amount)
5        throws LowBalanceException {
▶      if (amount > balance) {
7        throw new LowBalanceException(
8          "Insufficient funds", amount - balance);
9      }
10     balance -= amount;
11     System.out.println("Balance: " + balance);
12   }
13 }
14 // in main
15 Account acct = new Account(100);
16 try {
17   acct.withdraw(30);
18   acct.withdraw(90);
19   acct.withdraw(10);
20 } catch (LowBalanceException e) {
21   System.out.println(e.getMessage()
22     + ", deficit: " + e.getDeficit());
23 }
```



`withdraw(90)` : is 90 > 70? **Yes**.

Step 4 / 7

CALL STACK

`withdraw(90.0)`

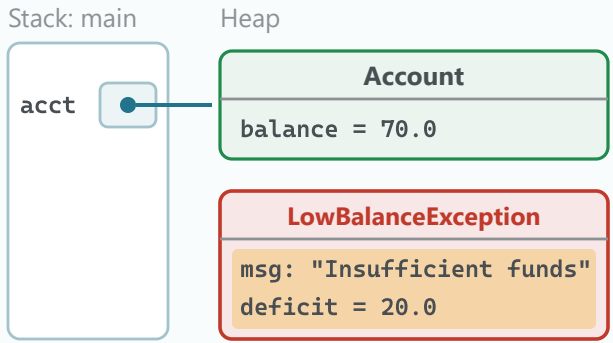
`main()`

OUTPUT

```
Balance: 70.0
```

Custom Exception: A Bank Account

```
1  class Account {
2    private double balance;
3    Account(double balance) { this.balance =
   balance; }
4    public void withdraw(double amount)
5        throws LowBalanceException {
6      if (amount > balance) {
▶        throw new LowBalanceException(
▶          "Insufficient funds", amount - balance);
9      }
10     balance -= amount;
11     System.out.println("Balance: " + balance);
12   }
13 }
14 // in main
15 Account acct = new Account(100);
16 try {
17   acct.withdraw(30);
18   acct.withdraw(90);
19   acct.withdraw(10);
20 } catch (LowBalanceException e) {
21   System.out.println(e.getMessage()
22     + ", deficit: " + e.getDeficit());
23 }
```



An **exception object** is created on the heap (message and deficit = 20.0) and thrown. The balance is **unchanged**.

Step 5 / 7

CALL STACK

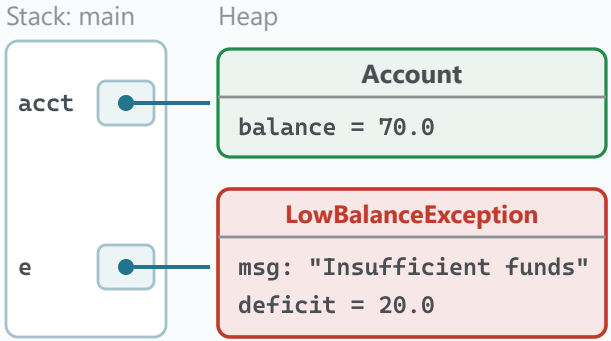
```
withdraw(90.0)
main()
```

OUTPUT

```
Balance: 70.0
```

Custom Exception: A Bank Account

```
1  class Account {
2    private double balance;
3    Account(double balance) { this.balance =
   balance; }
4    public void withdraw(double amount)
5        throws LowBalanceException {
6      if (amount > balance) {
7        throw new LowBalanceException(
8          "Insufficient funds", amount - balance);
9      }
10     balance -= amount;
11     System.out.println("Balance: " + balance);
12   }
13 }
14 // in main
15 Account acct = new Account(100);
16 try {
17   acct.withdraw(30);
18   acct.withdraw(90);
19   acct.withdraw(10);
▶ } catch (LowBalanceException e) {
21   System.out.println(e.getMessage()
22     + ", deficit: " + e.getDeficit());
23 }
```



withdraw's frame is discarded; `withdraw(10)` is **skipped**. catch matches: `e` refers to the same object.

Step 6 / 7

CALL STACK

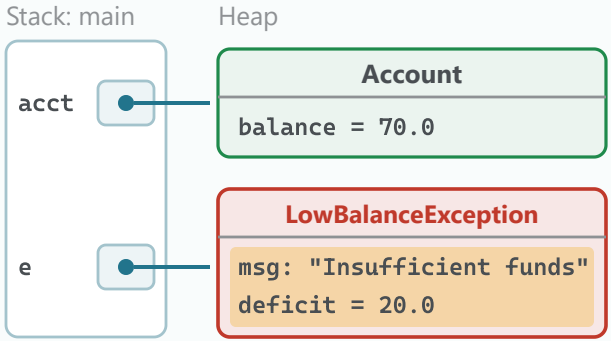
```
main()
```

OUTPUT

```
Balance: 70.0
```

Custom Exception: A Bank Account

```
1  class Account {
2    private double balance;
3    Account(double balance) { this.balance =
   balance; }
4    public void withdraw(double amount)
5        throws LowBalanceException {
6      if (amount > balance) {
7        throw new LowBalanceException(
8          "Insufficient funds", amount - balance);
9      }
10     balance -= amount;
11     System.out.println("Balance: " + balance);
12   }
13 }
14 // in main
15 Account acct = new Account(100);
16 try {
17   acct.withdraw(30);
18   acct.withdraw(90);
19   acct.withdraw(10);
20 } catch (LowBalanceException e) {
▶    System.out.println(e.getMessage()
▶      + ", deficit: " + e.getDeficit());
23 }
```



The exception object's data is used. An exception is just an ordinary object.

Step 7 / 7

CALL STACK

```
main()
```

OUTPUT

```
Balance: 70.0
Insufficient
funds, deficit:
20.0
```

A Custom Exception That Carries Data

```
class LowBalanceException extends Exception {
  private double deficit;

  public LowBalanceException(String msg, double deficit) {
    super(msg);            // kept in Throwable
    this.deficit = deficit;
  }

  public double getDeficit() {
    return deficit;
  }
}
```

- The class used in the previous trace: besides the message, it carries the **missing amount**.
- The catching code can use it to respond sensibly: "You are 20 TL short; try anyway?"
- A return code like `-1` could never carry this much information.
- A checked type was chosen: insufficient balance is a **foreseeable** situation the caller must deal with.

Stack Trace

- A report of **which method calls** were active and **where** the exception occurred.
- It holds the exception's type and message, and for each call the class and method name, the file name and the line number.
- For an uncaught exception the JVM prints it to the console (`System.err`); you can also print it with `e.printStackTrace()` .
- It is the **most important** clue for understanding and fixing a bug.

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.MyClass.myMethod(MyClass.java:10)
    at com.example.MyClass.main(MyClass.java:6)
```

Reading it: `main` (line 6) called `myMethod` ; the exception occurred at line 10 of `myMethod` . `com.example` is the package, `MyClass` the class.

STACK TRACE

Reading a Stack Trace

```
1  public class TraceDemo {
2    static int len(String text) {
3      return text.length();
4    }
5    static void show(String name) {
6      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
10     show(null);
11     System.out.println("Done");
12   }
13 }
```

1. **First line**: type and message: `text` is `null`.
2. **First `at`**: where the exception was **born** (line 3).
3. Reading down, the call chain: `main` (10) → `show` (6) → `len` (3).
4. Look at **your own** lines first; the real cause is often further down: line 10 passed the `null`.

Output

```
Ali: 3
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because
"text" is null
    at TraceDemo.len(TraceDemo.java:3)
    at TraceDemo.show(TraceDemo.java:6)
    at TraceDemo.main(TraceDemo.java:10)
```

STACK TRACE · STEP BY STEP

Where Does a Stack Trace Come From?

```
1  public class TraceDemo {
2    static int len(String text) {
3      return text.length();
4    }
5    static void show(String name) {
6      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
▶      show("Ali");
10     show(null);
11     System.out.println("Done");
12   }
13 }
```

Call stack (rightmost on top)

`main`

main calls `show("Ali")` at line 9.

Step 1 / 9

OUTPUT

STACK TRACE · STEP BY STEP

Where Does a Stack Trace Come From?

```
1  public class TraceDemo {
2    static int len(String text) {
3      return text.length();
4    }
5    static void show(String name) {
▶      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
10     show(null);
11     System.out.println("Done");
12   }
13 }
```

Call stack (rightmost on top)



show calls `len(name)` at line 6. Each frame remembers its current line.

Step 2 / 9

OUTPUT

Where Does a Stack Trace Come From?

```
1  public class TraceDemo {
2    static int len(String text) {
▶      return text.length();
4    }
5    static void show(String name) {
6      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
10     show(null);
11     System.out.println("Done");
12   }
13 }
```

Call stack (rightmost on top)



`"Ali".length()` = 3 is returned; the frame is removed.

Step 3 / 9

OUTPUT

Where Does a Stack Trace Come From?

```
1  public class TraceDemo {
2    static int len(String text) {
3      return text.length();
4    }
5    static void show(String name) {
▶      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
10     show(null);
11     System.out.println("Done");
12   }
13 }
```

Call stack (rightmost on top)



“Ali: 3” is printed and show ends.

Step 4 / 9

OUTPUT

```
Ali: 3
```

Where Does a Stack Trace Come From?

```
1  public class TraceDemo {
2    static int len(String text) {
3      return text.length();
4    }
5    static void show(String name) {
6      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
▶      show(null);
11     System.out.println("Done");
12   }
13 }
```

Call stack (rightmost on top)

`main`

main calls `show(null)` at line 10.

Step 5 / 9

OUTPUT

```
Ali: 3
```

Where Does a Stack Trace Come From?

```
1  public class TraceDemo {
2    static int len(String text) {
3      return text.length();
4    }
5    static void show(String name) {
▶      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
10     show(null);
11     System.out.println("Done");
12   }
13 }
```

Call stack (rightmost on top)



`null` is passed as an argument; no problem yet.

Step 6 / 9

OUTPUT

```
Ali: 3
```

Where Does a Stack Trace Come From?

```
1  public class TraceDemo {
2    static int len(String text) {
▶      return text.length();
4    }
5    static void show(String name) {
6      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
10     show(null);
11     System.out.println("Done");
12   }
13 }
```

Call stack (rightmost on top)



NullPointerException

`null.length()` : an NPE is thrown. The stack now holds three frames: len:3, show:6, main:10.

Step 7 / 9

OUTPUT

```
Ali: 3
```

Where Does a Stack Trace Come From?

```
1  public class TraceDemo {
2    static int len(String text) {
▶      return text.length();
4    }
5    static void show(String name) {
6      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
10     show(null);
11     System.out.println("Done");
12   }
13 }
```

Stack trace = snapshot of the stack

```
at len(TraceDemo.java:3)
at show(TraceDemo.java:6)
at main(TraceDemo.java:10)
```

When the exception object is born, it records the stack: each frame becomes an `at` line, the top one first.

Step 8 / 9

OUTPUT

```
Ali: 3
```

Where Does a Stack Trace Come From?

```
1  public class TraceDemo {
2    static int len(String text) {
3      return text.length();
4    }
5    static void show(String name) {
6      System.out.println(name + ": " + len(name));
7    }
8    public static void main(String[] args) {
9      show("Ali");
▶      show(null);
11     System.out.println("Done");
12   }
13 }
```

Stack trace = snapshot of the stack

```
at len(TraceDemo.java:3)
at show(TraceDemo.java:6)
at main(TraceDemo.java:10)
```

Nobody caught it: all three frames were discarded; the JVM printed the stack trace and ended the program. “Done” is never printed.

Step 9 / 9

OUTPUT

```
Ali: 3
Exception in thread "main"
java.lang.NullPointerException: Cannot
invoke "String.length()" because "text"
is null
    at TraceDemo.len(TraceDemo.java:3)
    at TraceDemo.show(TraceDemo.java:6)
    at
TraceDemo.main(TraceDemo.java:10)
```

Best Practices for Exception Handling

Catch specific types

Where possible, catch the expected type instead of `catch (Exception e)`, and respond to each kind appropriately.

Don't swallow exceptions

An empty `catch { }` hides the error. At least log it or tell the user.

Close resources

Close files, connections and other resources with try-with-resources or `finally`.

Prevent what you can

Check for `null`, bounds and zero; fix the code instead of catching `RuntimeException`.

Not for control flow

Don't use exceptions for ordinary tasks such as leaving a loop; exceptions are for **exceptional** situations.

Meaningful messages

State the bad value and what was expected: `"Grade must be 0–100: 120"`.

Handle in the right place

Catch an exception where it can be resolved; otherwise pass it on with `throws`.

Log them

Logging exceptions to a file makes it easier to trace errors later.

STACK TRACE

final, finally and finalize

	What?	Role
`final`	Keyword	Variable: can't be reassigned · method: can't be overridden · class: can't be subclassed
`finally`	Keyword (block)	Cleanup block after `try` that runs whether or not an exception occurs
`finalize()`	Method of `Object`	The garbage collector could call it before removing an object. When it runs is unpredictable; **deprecated** since Java 9.

Similar names, completely different jobs. For resource cleanup, use **try-with-resources** or `finally`, not `finalize()`.

Summary

Concept	Key point
Exception	Disrupts normal flow at run time; an object with type, message and location
`try` / `catch`	Risky code in `try` ; the **first** matching catch runs, rest of try skipped
catch order	Subclass before superclass; the reverse won't compile. `A | B` (Java 7+)
`finally`	Always runs, even before `return` ; for cleanup
try-with-resources	`try (Resource r = …)` : an `AutoCloseable` resource closes itself
Hierarchy	`Throwable` → `Error` | `Exception` → … `RuntimeException`
Checked	`Exception` (not Runtime): handle **or** declare with `throws`
Unchecked	`RuntimeException` , `Error` : not required; prevent
`throw` / `throws`	Throw (body, object) / declare (signature, types)
Propagation	Uncaught exception passes to the caller; if `main` doesn't catch it, the program ends
Custom exception	`class X extends Exception` + `super(message)`
Stack trace	Type + message, then the call chain; the first `at` line is the origin

SUMMARY

Common Mistakes

Forgetting a checked exception

`new FileReader(name)` with no try/throws: `unreported exception FileNotFoundException; must be caught or declared to be thrown`.

General catch first

`catch (Exception e)`, then `catch (ArithmeticException e)`: `exception ArithmeticException has already been caught`.

Catching what is never thrown

`catch (FileNotFoundException e)` on a try with no file operation: `exception FileNotFoundException is never thrown in body of corresponding try statement`.

A lone try

A try with no catch or finally: `'try' without 'catch', 'finally' or resource declarations`.

Code after throw

A line right after `throw` in the same block: `unreachable statement`.

Using a try variable outside

Printing `result`, declared inside try, after the block: `cannot find symbol`. Declare it before the try.

Swallowing the exception

`catch (Exception e) { }` compiles, but the error vanishes without a trace.

Throwing a string

`throw "error";` → `incompatible types: String cannot be converted to Throwable`.

SUMMARY

Practice Problems (1)

PROBLEM 1: WHAT IS THE OUTPUT?

```
static int f() {
  try {
    System.out.print("A");
    int x = 5 / 0;
    System.out.print("B");
    return 1;
  } catch (ArithmeticException e) {
    System.out.print("C");
    return 2;
  } finally {
    System.out.print("D");
  }
}
// in main
System.out.println(" " + f());
```

PROBLEM 2: WHAT IS THE OUTPUT?

```
try {
  String s = "12a";
  int n = Integer.parseInt(s);
  System.out.println("Number: " + n);
} catch (IllegalArgumentException e) {
  System.out.println("IAE");
} catch (Exception e) {
  System.out.println("Exception");
}
```

PROBLEM 3

What happens if the two catch blocks in Problem 2 swap places?

▶ **Show answers**

SUMMARY

Practice Problems (2)

PROBLEM 4: WHAT IS THE OUTPUT?

```
static void a() {
  System.out.print("a1 ");
  b();
  System.out.print("a2 ");
}
static void b() {
  System.out.print("b1 ");
  throw new IllegalStateException("!");
}
// in main
try {
  a();
  System.out.print("m ");
} catch (IllegalStateException e) {
  System.out.print("caught" + e.getMessage());
} finally {
  System.out.println(" end");
}
```

The exception is born in `b()`; neither `b()` nor `a()` has a try. Which `print`s run and which are skipped? When does `finally` run?

▶ **Show answer**

SUMMARY

Practice Problems (3)

PROBLEM 5: FINAL VALUE OF SUM?

```
int[] d = {4, 0, 2};
int sum = 0;
for (int i = 0; i <= d.length; i++) {
  try {
    sum += 12 / d[i];
  } catch (ArithmeticException e) {
    sum += 100;
  } catch (ArrayIndexOutOfBoundsException e) {
    sum += 1000;
  }
}
System.out.println(sum);
```

PROBLEM 6: WHAT IS THE OUTPUT?

```
try {
  throw new RuntimeException("X");
} finally {
  System.out.println("F");
}
```

▶ **Show answers**

SUMMARY

Exercises (1)

1. Write `int readAge(String text)`: if the text isn't a number, let `NumberFormatException` occur; if negative, throw `IllegalArgumentException`. Use two catches to print messages for `"20"`, `"-3"`, `"twenty"`.
2. Write a checked exception `InvalidGradeException` and `setGrade(int grade)`, which throws it for grades outside 0–100.
3. In Exercise 1, what if the catch order is `IllegalArgumentException`, then `NumberFormatException`?

▶ **Answers / hints**

SUMMARY

Exercises (2)

4. Classify as checked or unchecked: `IOException`, `NumberFormatException`, `SQLException`, `NullPointerException`, `FileNotFoundException`, `ClassCastException`, `StackOverflowError`.
5. Rewrite code that closes a `BufferedReader` in `finally` using try-with-resources.
6. What should `double average(int[] grades)` do for an empty array? Return `0` or throw an exception? Which one?

▶ **Answers / hints**

SUMMARY

References

- Oracle, *The Java Tutorials*: Lesson: Exceptions (Catching and Handling Exceptions; The try-with-resources Statement; Unchecked Exceptions — The Controversy).
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition, Chapter 11: Exceptions; 14.20 The try statement.
- Oracle, *Java SE 21 API Documentation*: `java.lang.Throwable`, `Exception`, `RuntimeException`, `Error`.
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 11: Exception Handling: A Deeper Look).
- K. Sierra, B. Bates and T. Gee, *Head First Java*, 3rd ed., O'Reilly (Risky Behavior: exception handling).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapter 7: Exceptions, Assertions, and Logging).
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley (Chapter 10: Exceptions, Items 69–77).

The End

Chapter 6: Exceptions

Previous: Chapter 5: Collections · Next: Chapter 7: Testing and Debugging