

UNIT 5

Handling Error / Exception

(2 Hours) — Object Oriented Programming in Java (CACS204)

Programs inevitably encounter unexpected situations — invalid input, missing files, division by zero, and more. This unit covers Java's exception handling mechanism: what exceptions are, how to use them properly, how to define your own, and the try, catch, throw, throws, and finally keywords that let a program detect and recover from errors gracefully instead of crashing outright.

5.1 Basic Exceptions

An exception is an event, occurring during program execution, that disrupts the normal flow of instructions. When an error condition arises, Java creates an exception object containing information about the error and hands control to the runtime system, which searches for code able to handle it. If no handler is found, the program terminates and prints a stack trace.

All exception types in Java descend from the class Throwable, which branches into two main subclasses: Error, representing serious problems a normal application should not try to catch (such as running out of memory), and Exception, representing conditions a well-written program may reasonably want to catch and handle.

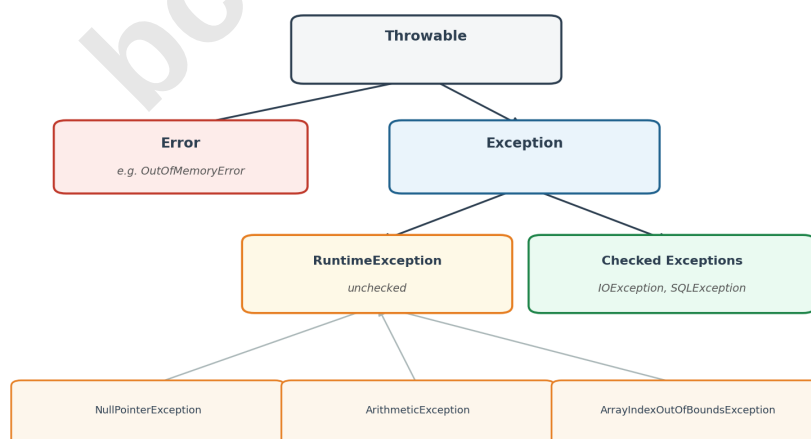


Figure 5.1 — The Throwable hierarchy: Error and Exception, with Exception further split into checked exceptions and the unchecked RuntimeException branch.

Category	Checked at compile time?	Typical cause
Checked Exception	Yes — must be caught or declared	File not found, network failure

Category	Checked at compile time?	Typical cause
RuntimeException (unchecked)	No — compiler does not enforce handling	Programming logic errors
Error	No — not meant to be caught	JVM-level failures (e.g. out of memory)

A few of the most frequently encountered built-in exceptions:

- `ArithmeticException` — e.g. integer division by zero
- `NullPointerException` — calling a method or accessing a field on a null reference
- `ArrayIndexOutOfBoundsException` — accessing an array index outside its valid range
- `NumberFormatException` — converting an invalid `String` to a numeric type
- `ClassCastException` — an invalid cast between incompatible reference types

5.2 Proper Use of Exceptions

Exceptions exist to signal genuinely exceptional, error-like conditions — not to control ordinary program flow. Used correctly, they keep error-handling code separate from the main logic and make failures explicit rather than silent.

Guidelines for Proper Exception Use

- ✓ Use exceptions for error conditions, not as a substitute for simple if/else logic
- ✓ Catch only the exceptions you can meaningfully handle or recover from
- ✓ Never leave a catch block empty — at minimum, log the error
- ✓ Throw the most specific exception type that accurately describes the problem
- ✓ Document any checked exceptions a method may throw
- ✓ Release resources (files, connections) in a finally block or try-with-resources

Note: Catching a broad type like `Exception` everywhere and silently discarding it hides genuine bugs and makes programs much harder to debug — this is sometimes called "swallowing" an exception, and should be avoided.

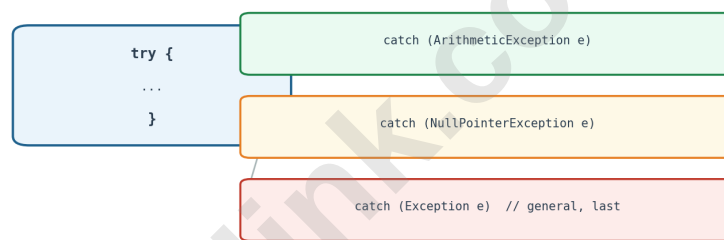
5.3 Catching Exceptions: try and catch

Code that might raise an exception is placed inside a try block. Immediately following it, one or more catch blocks specify the exception type(s) they are prepared to handle. If the code in the try block raises a matching exception, execution jumps immediately to the corresponding catch block; if no exception occurs, the catch blocks are simply skipped.

```
try {
    int result = 10 / 0; // raises ArithmeticException
    System.out.println(result); // never reached
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero: " + e.getMessage());
}
```

5.3.1 Multiple Catch Blocks

A single try block may be followed by several catch blocks, each handling a different exception type. Java tests them from top to bottom and executes the first one whose type matches (or is a superclass of) the exception that was actually thrown, so more specific exception types must be listed before more general ones — placing a general handler first would make the later, more specific handlers unreachable and cause a compile error.



Multiple catch blocks let a try statement handle different exception types differently. Java checks them top to bottom and runs the FIRST one whose type matches the thrown exception — more specific types should be listed before more general ones.

Figure 5.2 — Multiple catch blocks attached to one try statement, checked in order.

```
try {
    int[] arr = new int[5];
    arr[10] = 100; // out of bounds
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Index problem: " + e.getMessage());
} catch (Exception e) { // general handler, listed LAST
    System.out.println("Something else went wrong");
}
```

Example 5.1

Write code that safely converts a user-supplied String to an int, printing a friendly message instead of crashing if the input is not a valid number.

Solution

```
String input = "abc";

try {
    int n = Integer.parseInt(input);
    System.out.println("Parsed: " + n);
} catch (NumberFormatException e) {
    System.out.println("'" + input + "' is not a valid number.");
}
```

5.4 Throwing and Rethrowing: throw and throws

Java provides two related but distinct keywords for working with exceptions proactively, rather than only reacting to ones raised automatically by the runtime.

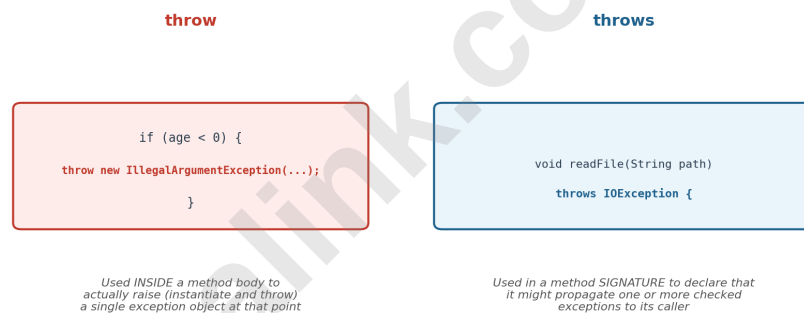


Figure 5.3 — `throw` raises one exception at a specific point; `throws` declares what a method might propagate.

5.4.1 The 'throw' Statement

`throw` is used inside a method body to explicitly raise an exception — either a newly created one or a previously caught one being rethrown. Execution of the current block stops immediately and control passes to the nearest matching catch block (or up the call stack if none exists locally).

```
void setAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("Age cannot be negative");
    }
    this.age = age;
}
```

5.4.2 The 'throws' Clause

throws appears in a method's signature to declare that the method might propagate one or more checked exceptions to its caller instead of handling them itself. This is compulsory for checked exceptions: the compiler requires that any checked exception a method might raise is either caught inside the method or listed in its throws clause.

```
void readFile(String path) throws IOException {
    FileReader reader = new FileReader(path); // may throw IOException
    // ... read contents ...
}
// A caller of readFile() must now either catch IOException
// or declare throws IOException itself
void process() throws IOException {
    readFile("data.txt");
}
```

5.4.3 Rethrowing an Exception

A catch block may perform some handling (such as logging) and then rethrow the same exception — or wrap it inside a different, more meaningful exception type — so that code further up the call stack still gets a chance to respond.

```
try {
    readFile("config.txt");
} catch (IOException e) {
    System.out.println("Logging error: " + e.getMessage());
    throw e; // rethrow the same exception upward
}
```

Note: A checked exception's presence in a throws clause is part of a method's public contract — changing it later can break code that calls the method, so it should be chosen deliberately.

5.5 User-Defined Exceptions

Java allows programmers to define their own exception classes for application-specific error conditions that none of the built-in exception types describe well. A custom exception is created by extending either `Exception` (to make it checked) or `RuntimeException` (to make it unchecked), and typically supplies one or more constructors that forward a descriptive message to the superclass.

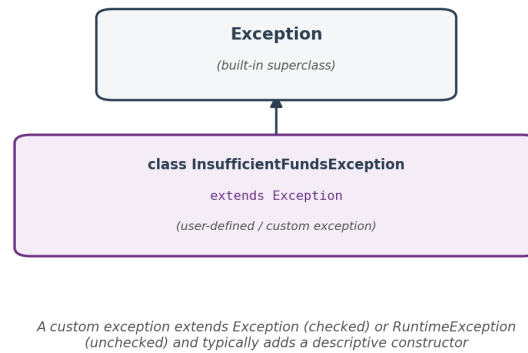


Figure 5.4 — A user-defined exception extending the built-in *Exception* class.

```
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message); // forward message to Exception's constructor
    }
}
```

It can then be thrown and caught exactly like a built-in exception:

```
class Account {
    double balance;
    void withdraw(double amt) throws InsufficientFundsException {
        if (amt > balance) {
            throw new InsufficientFundsException("Not enough balance");
        }
        balance -= amt;
    }
}

try {
    acc.withdraw(500);
} catch (InsufficientFundsException e) {
    System.out.println(e.getMessage());
}
```

Example 5.2

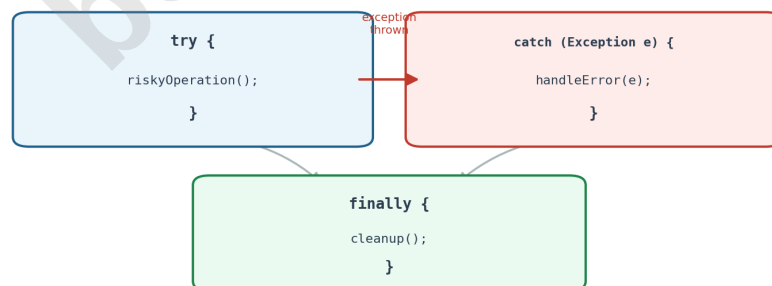
Define a custom unchecked exception *InvalidGradeException* for a grading system, where a grade must be between 0 and 100.

Solution

```
class InvalidGradeException extends RuntimeException {  
    public InvalidGradeException(String message) {  
        super(message);  
    }  
}  
  
void setGrade(int grade) {  
    if (grade < 0 || grade > 100) {  
        throw new InvalidGradeException("Grade must be 0-100");  
    }  
    this.grade = grade;  
}
```

5.6 Cleaning Up Using the 'finally' Clause

A finally block, placed after the try/catch blocks, always executes — whether the try block completed normally, an exception was thrown and caught, or an exception propagated out uncaught. It is the standard place to release resources such as open files, network connections, or database handles, guaranteeing cleanup happens exactly once no matter what path execution takes.



finally always runs — whether an exception was thrown and caught, or the try block completed normally

Figure 5.5 — Control flow through try, catch, and finally; finally is never skipped.

```

FileReader reader = null;
try {
    reader = new FileReader("data.txt");
    // ... use reader ...
} catch (IOException e) {
    System.out.println("File error: " + e.getMessage());
} finally {
    if (reader != null) {
        try { reader.close(); } catch (IOException e) { }
    }
    System.out.println("Cleanup complete.");
}

```

Note: A try statement must have at least one catch block or a finally block (or both); a bare try with neither does not compile. A try can also be paired with finally alone, with no catch block at all, when the goal is only guaranteed cleanup.

Key Takeaways

- ✓ All exceptions descend from Throwable, split into Error and Exception
- ✓ Checked exceptions must be caught or declared; RuntimeException (unchecked) need not be
- ✓ try wraps risky code; catch handles a specific exception type; order specific before general
- ✓ throw raises a single exception at a point in code; throws declares what a method may propagate
- ✓ A caught exception can be rethrown, or wrapped in a new exception, to inform callers further up
- ✓ Custom exceptions extend Exception (checked) or RuntimeException (unchecked) for domain-specific errors
- ✓ finally always runs, making it the right place for guaranteed cleanup of resources

— End of Unit 5 —