PROGRAMMING IN JAVA

Lecture 10

Exception Handling and Defensive Programming

Exceptions, `try-catch-finally`, `throw`, `throws`, Custom Exceptions, Propagation, and Try-With-Resources

Lecturer: Arshyn Altybay

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

1. Learning Objectives

After completing this lecture, students should be able to:

1. explain what an exception is;
2. distinguish compile-time errors, runtime errors, and exceptions;
3. describe the Java exception hierarchy;
4. distinguish checked and unchecked exceptions;
5. use `try`, `catch`, and `finally`;
6. handle several exception types;
7. use multi-catch syntax;
8. create and throw exceptions using `throw`;
9. declare propagated exceptions using `throws`;
10. explain exception propagation through method calls;
11. define and use custom exceptions;
12. use try-with-resources correctly;
13. preserve useful diagnostic information;
14. apply defensive programming principles;
15. design robust Java programs that recover from invalid input.

2. Introduction

Even a syntactically correct program may encounter unexpected situations while running.

Examples include:

- division by zero;
- accessing an invalid array index;
- converting invalid text into a number;
- opening a file that does not exist;
- receiving invalid user input;
- trying to withdraw more money than is available;
- trying to use a `null` reference.

A reliable program should not simply terminate whenever such a situation occurs.

Java provides a structured mechanism called **exception handling**.

3. What Is an Exception?

An **exception** is an object that represents an abnormal or unexpected condition that occurs during program execution.

When an exceptional condition occurs, Java creates or throws an exception object.

If the exception is not handled, the program may terminate and display a stack trace.

Example:

```
public class Main {

    public static void main(String[] args) {

        int a = 10;
        int b = 0;

        int result = a / b;

        System.out.println(result);
    }
}
```

This causes:

Illustration

`ArithmeticException`

because integer division by zero is not allowed.

4. Errors vs Exceptions

It is useful to distinguish several kinds of problems.

4.1 Compile-Time Errors

These are detected by the compiler.

Example:

```
int x = "Java"; // incompatible types
```

The program cannot compile.

4.2 Runtime Exceptions

These occur after successful compilation while the program is running.

Examples:

- `ArithmeticException`
- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `NumberFormatException`

4.3 Serious Errors

Java also has `Error` subclasses, such as serious JVM or system-level problems.

Application code normally does not try to recover from most `Error` objects.

5. The Exception Hierarchy

At the top of the Java throwable hierarchy is:

Illustration

```
Throwable
  |
  +-- Error
  |
  +-- Exception
       |
       +-- RuntimeException
       |
       +-- other checked exceptions
```

Important ideas:

- `Throwable` is the common superclass;
- `Error` represents serious problems;
- `Exception` represents conditions that applications may handle;
- `RuntimeException` and its subclasses are unchecked exceptions.

6. Checked and Unchecked Exceptions

6.1 Checked Exceptions

Checked exceptions must be either:

- handled with `try-catch`; or
- declared using `throws`.

Examples include many file and database-related exceptions.

6.2 Unchecked Exceptions

Unchecked exceptions are subclasses of `RuntimeException`.

Examples:

- `ArithmeticException`
- `NullPointerException`

- `IllegalArgumentException`
- `ArrayIndexOutOfBoundsException`
- `NumberFormatException`

The compiler does not force the programmer to catch or declare them.

7. The `try-catch` Statement

The basic syntax is:

```
try {
    // risky code
} catch (ExceptionType e) {
    // handling code
}
```

The `try` block contains code that may fail.

The `catch` block handles a matching exception.

Example 1. Handling Division by Zero

```
1  public class Main {
2
3      public static void main(String[] args) {
4
5          try {
6
7              int result = 10 / 0;
8
9              System.out.println(result);
10
11         } catch (ArithmeticException e) {
12
13             System.out.println(
14                     "Division by zero is not allowed."
15             );
16         }
17
18         System.out.println(
19                 "Program continues."
20         );
21     }
22 }
```

Expected output

```
Division by zero is not allowed.
Program continues.
```

8. Accessing Exception Information

An exception object contains useful information.

Common methods include:

- `getMessage()`
- `toString()`
- `printStackTrace()`

Example:

```
catch (Exception e) {
    System.out.println(e.getMessage());
}
```

Example 2. Displaying an Exception Message

```
public class Main {

    public static void main(String[] args) {

        try {

            int[] numbers = {10, 20, 30};

            System.out.println(
                    numbers[10]
            );

        } catch (
                ArrayIndexOutOfBoundsException e) {

            System.out.println(
                    "Problem: " +
                    e.getMessage()
            );
        }
    }
}
```

The exact message can differ slightly across Java versions.

9. Multiple Catch Blocks

One `try` block may be followed by several `catch` blocks.

```
try {
    // risky code
} catch (NumberFormatException e) {
    // ...
} catch (ArithmeticException e) {
    // ...
} catch (Exception e) {
    // ...
}
```

Catch blocks should be ordered from more specific to more general types.

Incorrect:

```
try {
    // ...
} catch (Exception e) {
    // ...
} catch (NumberFormatException e) {
    // unreachable
}
```

Example 3. Multiple Catch Blocks

```
public class Main {

    public static void main(String[] args) {

        String text = "abc";

        try {

            int number =
                    Integer.parseInt(text);

            int result = 100 / number;

            System.out.println(result);

        } catch (NumberFormatException e) {

            System.out.println(
                    "Invalid integer text."
            );

        } catch (ArithmeticException e) {

            System.out.println(
                    "Arithmetic problem."
            );
        }
    }
}
```

Expected output

```
Invalid integer text.
```

10. Multi-Catch

If several exception types should be handled in the same way, Java supports multi-catch syntax.

```
try {
    // ...
```

```
} catch (
        NumberFormatException |
        ArithmeticException e) {

    System.out.println("Invalid operation");
}
```

This can reduce duplicated handling code.

Example 4. Multi-Catch

```
1  public class Main {
2
3      public static void main(String[] args) {
4
5          String text = "0";
6
7          try {
8
9              int number =
10                     Integer.parseInt(text);
11
12             System.out.println(
13                     10 / number
14             );
15
16         } catch (
17                 NumberFormatException |
18                 ArithmeticException e) {
19
20             System.out.println(
21                     "The operation could not be completed."
22             );
23         }
24     }
25 }
```

Expected output

```
The operation could not be completed.
```

11. The `finally` Block

A `finally` block is normally executed whether an exception occurs or not.

Syntax:

```
try {
    // ...
} catch (Exception e) {
    // ...
} finally {
    // cleanup
}
```

It is often used for cleanup operations.

However, for many resources, modern Java prefers try-with-resources.

Example 5. Finally

```
1  public class Main {
2
3      public static void main(String[] args) {
4
5          try {
6
7              System.out.println(
8                      "Inside try"
9              );
10
11         } finally {
12
13             System.out.println(
14                     "Inside finally"
15             );
16         }
17     }
18 }
```

Expected output

```
Inside try
Inside finally
```

12. The `throw` Keyword

The keyword `throw` is used to explicitly create or propagate an exception.

Example:

```
if (age < 0) {
    throw new IllegalArgumentException(
            "Age cannot be negative"
    );
}
```

This is useful for enforcing object invariants and validating method arguments.

Example 6. Throwing an Exception

```
public class Main {

    static void setAge(int age) {

        if (age < 0) {

            throw new
                IllegalArgumentException(
                    "Age cannot be negative"
                );
        }

        System.out.println(
                "Age = " + age
        );
    }

    public static void main(String[] args) {

        try {

            setAge(-5);

        } catch (
                IllegalArgumentException e) {

            System.out.println(
                    e.getMessage()
            );
        }
    }
}
```

Expected output

```
Age cannot be negative
```

13. The `throws` Keyword

The keyword `throws` is used in a method declaration to indicate that the method may propagate one or more exceptions.

Example:

```
void readData() throws IOException {
    // ...
}
```

This means the caller must handle or further propagate the checked exception.

14. Throw vs Throws

Property	`throw`	`throws`
Used where	inside method/body	method declaration
Purpose	actually throws an exception	declares possible propagation
Followed by	exception object	exception type(s)
Example	`throw new IllegalArgumentException()`	`throws IOException`

15. Exception Propagation

If a method does not catch an exception, the exception can propagate to the calling method.

Consider:

Illustration

```
main()
  |
  +-- methodA()
       |
       +-- methodB()
            |
            +-- exception
```

If `methodB()` does not handle the exception, Java looks for a matching handler in `methodA()`,

and then in its caller, and so on.

Example 7. Exception Propagation

```
1  public class Main {
2
3      static void methodC() {
4
5          int result = 10 / 0;
6
7          System.out.println(result);
8      }
9
10     static void methodB() {
11         methodC();
12     }
13
14     static void methodA() {
15         methodB();
16     }
17
18     public static void main(String[] args) {
19
20         try {
21
22             methodA();
23
24         } catch (ArithmeticException e) {
25
26             System.out.println(
27                     "Handled in main."
28             );
29         }
30     }
31 }
```

Expected output

```
Handled in main.
```

16. Custom Exceptions

Java allows programmers to define their own exception types.

A custom checked exception normally extends `Exception`.

Example:

```
class InvalidAgeException
        extends Exception {

    public InvalidAgeException(
            String message) {

        super(message);
    }
}
```

A custom unchecked exception normally extends `RuntimeException`.

Example 8. Custom Checked Exception

```
1  public class Main {
2
3      static class InvalidGpaException
4              extends Exception {
5
6          public InvalidGpaException(
7                  String message) {
8
9              super(message);
10         }
11     }
12
13     static void validateGpa(double gpa)
14             throws InvalidGpaException {
15
16         if (gpa < 0.0 || gpa > 4.0) {
17
18             throw new InvalidGpaException(
19                     "GPA must be between 0 and 4."
20             );
21         }
22     }
23
24     public static void main(String[] args) {
```

Example 8 (continued)

```
25
26          try {
27
28              validateGpa(5.2);
29
30          } catch (InvalidGpaException e) {
31
32              System.out.println(
33                      e.getMessage()
34              );
35          }
36      }
37  }
```

Expected output

```
GPA must be between 0 and 4.
```

Example 9. Banking Custom Exception

```
1   public class Main {
2
3       static class
4               InsufficientFundsException
5               extends Exception {
6
7           public
8           InsufficientFundsException(
9                   String message) {
10
11              super(message);
12          }
13      }
14
15      static class BankAccount {
16
17          private double balance;
18
19          public BankAccount(
20                  double initialBalance) {
21
22              balance = initialBalance;
23          }
24
```

Example 9 (continued)

```
25          public void withdraw(
26                  double amount)
27                  throws
28                  InsufficientFundsException {
29
30              if (amount > balance) {
31
32                  throw new
33                      InsufficientFundsException(
34                          "Not enough money."
35                      );
36              }
37
38              balance -= amount;
39          }
40
41          public double getBalance() {
42              return balance;
43          }
44      }
45
46      public static void main(String[] args) {
47
48          BankAccount account =
49                  new BankAccount(50000);
50
51          try {
52
53              account.withdraw(70000);
54
55          } catch (
56                  InsufficientFundsException e) {
57
58              System.out.println(
59                      e.getMessage()
60              );
61          }
62
63          System.out.println(
64                  "Balance = " +
65                  account.getBalance()
66          );
67      }
68  }
```

Expected output

```
Not enough money.
Balance = 50000.0
```

17. Checked or Unchecked Custom Exception?

A useful rule of thumb:

Use a checked exception when:

- the caller is reasonably expected to recover;
- the condition is part of an external operation or recoverable workflow.

Use an unchecked exception when:

- the caller violated a method contract;
- the problem represents invalid programming usage;
- forcing every caller to catch the exception would add little value.

There is no universal rule for every application.

The exception design should reflect the API contract.

18. Wrapping Exceptions

Sometimes a low-level exception should be translated into a higher-level exception while preserving the original cause.

Example:

```
throw new RuntimeException(
        "Could not load student data",
        originalException
);
```

The original exception is stored as the cause.

This preserves diagnostic information.

Example 10. Exception Wrapping

```
public class Main {

    static void lowLevelOperation() {

        throw new
            NumberFormatException(
                "Invalid number"
            );
    }

    static void highLevelOperation() {

        try {

            lowLevelOperation();

        } catch (
                NumberFormatException e) {

            throw new RuntimeException(
                    "Operation failed",
                    e
            );
        }
    }

    public static void main(String[] args) {

        try {

            highLevelOperation();

        } catch (RuntimeException e) {

            System.out.println(
                    e.getMessage()
            );

            System.out.println(
                    "Cause: " +
                    e.getCause()
                     .getClass()
                     .getSimpleName()
            );
```

Example 10 (continued)

```
45            }
46        }
47    }
```

Expected output

```
Operation failed
Cause: NumberFormatException
```

19. Try-With-Resources

Many resources must be closed after use.

Examples include:

- files;
- streams;
- sockets;
- database connections;
- database statements.

Java provides **try-with-resources**.

Syntax:

```
try (Resource r = createResource()) {

    // use resource
}
```

The resource is automatically closed.

The resource must implement `AutoCloseable`.

Example 11. Try-With-Resources

```
import java.io.BufferedReader;
import java.io.StringReader;

public class Main {

    public static void main(String[] args)
            throws Exception {

        String text =
                "Java\nProgramming";

        try (BufferedReader reader =
                new BufferedReader(
                    new StringReader(text))) {

            String line;

            while ((line =
                    reader.readLine())
                    != null) {

                System.out.println(line);
            }
        }
    }
}
```

Expected output

```
Java
Programming
```

20. Why Try-With-Resources Is Preferred

Compared with manual cleanup in `finally`, try-with-resources:

- reduces boilerplate code;
- automatically closes resources;
- handles cleanup more reliably;
- preserves suppressed exceptions when cleanup also fails.

21. Input Validation and Exceptions

User input often needs validation.

Example:

```
String input = "abc";

try {

    int value =
            Integer.parseInt(input);

} catch (NumberFormatException e) {

    System.out.println(
            "Please enter an integer."
    );
}
```

A user-facing program should usually give a helpful message and allow the user to try again.

Example 12. Robust Input Loop

```
1  import java.util.Scanner;
2
3  public class Main {
4
5      public static void main(String[] args) {
6
7          Scanner scanner =
8                  new Scanner(System.in);
9
10         int number;
11
12         while (true) {
13
14             System.out.print(
15                     "Enter an integer: "
16             );
17
18             String text =
19                     scanner.nextLine();
20
21             try {
22
```

Example 12 (continued)

```
                number =
                    Integer.parseInt(text);

                break;

            } catch (
                    NumberFormatException e) {

                System.out.println(
                    "Invalid input. Try again."
                );
            }
        }

        System.out.println(
                "You entered: " + number
        );
    }
}
```

22. Assertions

Assertions are used to check internal assumptions during development.

Syntax:

```
assert condition;
```

or:

```
assert condition : "message";
```

Example:

```
assert value >= 0 :
        "Value should not be negative";
```

Assertions are different from normal validation.

They should not be used to validate ordinary user input or required business conditions.

Assertions may also be disabled unless enabled at runtime.

23. Exception Handling and Method Contracts

A well-designed method should clearly define:

- valid input;
- normal return behavior;
- exceptional conditions;
- whether exceptions are checked or unchecked.

Example contract:

```
public void withdraw(double amount)
```

Possible rules:

- amount must be positive;
- amount must not exceed the balance;
- invalid amount may cause `IllegalArgumentException`;
- insufficient balance may cause a domain-specific exception.

24. Defensive Programming

Defensive programming means writing code that anticipates invalid inputs and unexpected states.

Typical practices include:

1. validate method arguments;
2. preserve class invariants;
3. reject invalid states early;
4. use meaningful exception types;
5. avoid hiding errors silently;
6. preserve the original cause when translating exceptions;
7. release resources reliably;
8. provide useful error messages;
9. recover when recovery is realistic.

25. Do Not Catch Exceptions Too Broadly

This is often poor practice:

```
try {
    // large amount of code
} catch (Exception e) {
    // ignore everything
}
```

Problems:

- unrelated errors are mixed together;
- bugs may be hidden;
- diagnosis becomes difficult.

Prefer catching the specific exceptions you can meaningfully handle.

26. Do Not Use Exceptions for Normal Control Flow

Poor style:

```
try {
    while (true) {
        System.out.println(
                array[i++]
        );
    }
} catch (
        ArrayIndexOutOfBoundsException e) {

    // loop ends
}
```

Better:

```
for (int i = 0;
     i < array.length;
     i++) {

    System.out.println(array[i]);
```

```
}
```

Exceptions should represent exceptional situations, not ordinary loop control.

27. Do Not Ignore Exceptions

Poor:

```
catch (Exception e) {
}
```

At minimum, the application should normally:

- recover;
- report the problem;
- log it;
- or rethrow it appropriately.

Example 13. Domain Validation

```
1  public class Main {
2
3      static class Student {
4
5          private String name;
6          private double gpa;
7
8          public Student(
9                  String name,
10                 double gpa) {
11
12             setName(name);
13             setGpa(gpa);
14         }
15
16         public void setName(
17                 String name) {
18
19             if (name == null ||
20                 name.isBlank()) {
21
22                 throw new
```

Example 13 (continued)

```
                    IllegalArgumentException(
                        "Name is required."
                    );
            }

            this.name = name;
        }

        public void setGpa(
                double gpa) {

            if (gpa < 0 ||
                gpa > 4) {

                throw new
                    IllegalArgumentException(
                        "GPA must be 0..4."
                    );
            }

            this.gpa = gpa;
        }

        @Override
        public String toString() {

            return name +
                    " (" + gpa + ")";
        }
    }

    public static void main(String[] args) {

        try {

            Student student =
                new Student(
                    "Aruzhan", 4.5
                );

            System.out.println(student);

        } catch (
                IllegalArgumentException e) {

```

Example 13 (continued)

```
            System.out.println(
                    e.getMessage()
            );
        }
    }
}
```

Expected output

```
GPA must be 0..4.
```

Example 14. Custom Registration Exception

```
public class Main {

    static class
            RegistrationException
            extends Exception {

        public RegistrationException(
                String message) {

            super(message);
        }
    }

    static void registerStudent(
            int current,
            int capacity)
            throws
            RegistrationException {

        if (current >= capacity) {

            throw new
                RegistrationException(
                    "Course is full."
                );
        }

        System.out.println(
                "Registration successful."
        );
    }
```

Example 14 (continued)

```
32
33      public static void main(String[] args) {
34
35          try {
36
37              registerStudent(30, 30);
38
39          } catch (
40                  RegistrationException e) {
41
42              System.out.println(
43                      e.getMessage()
44              );
45          }
46      }
47  }
```

Expected output

```
Course is full.
```

Example 15. Centralized Service-Layer Handling

```
1   public class Main {
2
3       static class
4               StudentNotFoundException
5               extends RuntimeException {
6
7           public
8           StudentNotFoundException(
9                   String message) {
10
11              super(message);
12          }
13      }
14
15      static class StudentService {
16
17          public String findById(int id) {
18
19              if (id != 1) {
20
21                  throw new
```

Example 15 (continued)

```
                    StudentNotFoundException(
                        "Student not found: " +
                        id
                    );
            }

            return "Aruzhan";
        }
    }

    public static void main(String[] args) {

        StudentService service =
                new StudentService();

        try {

            System.out.println(
                    service.findById(5)
            );

        } catch (
                StudentNotFoundException e) {

            System.out.println(
                    "Service error: " +
                    e.getMessage()
            );
        }
    }
}
```

Expected output

```
Service error: Student not found: 5
```

28. Common Exceptions in Java

Exception	Typical cause
`ArithmeticException`	illegal arithmetic such as integer division by zero
`NullPointerException`	using a member through a null reference
`ArrayIndexOutOfBoundsException`	invalid array index
`NumberFormatException`	invalid numeric text conversion
`IllegalArgumentException`	invalid method argument
`IllegalStateException`	operation invalid for current object state
`ClassCastException`	invalid reference downcast
`IOException`	input/output failure

29. Common Mistakes

29.1 Mistake 1: Catching Too Much

Avoid large `try` blocks with one general catch when more precise handling is possible.

29.2 Mistake 2: Empty Catch Blocks

Never silently swallow errors without a reason.

29.3 Mistake 3: Using Exceptions Instead of Validation

If a condition can be checked clearly before an operation, normal validation may be more appropriate.

29.4 Mistake 4: Losing the Original Cause

When translating exceptions, preserve the original exception as the cause.

29.5 Mistake 5: Throwing Generic `Exception` Everywhere

Prefer meaningful exception types.

29.6 Mistake 6: Catching an Exception and Immediately Rethrowing It Without Value

If you catch an exception, add value through recovery, translation, logging, or additional context.

30. Design Guidelines

When designing exception handling:

1. validate inputs at boundaries;
2. use specific exception types;
3. catch exceptions only where meaningful handling is possible;
4. keep `try` blocks reasonably small;
5. avoid empty catch blocks;
6. do not expose unnecessary low-level details to end users;
7. preserve exception causes;
8. use try-with-resources for closeable resources;
9. document important exceptional behavior;
10. distinguish user errors from programming errors;
11. keep normal program logic separate from exceptional logic.

31. Summary

In this lecture, we learned that:

- exceptions represent abnormal runtime conditions;
- Java exceptions belong to the `Throwable` hierarchy;
- checked exceptions must be handled or declared;
- unchecked exceptions are subclasses of `RuntimeException`;
- `try-catch` handles exceptions;
- multiple catch blocks can handle different exception types;
- multi-catch can combine related handling;
- `finally` supports cleanup logic;
- `throw` explicitly throws an exception;
- `throws` declares exception propagation;
- exceptions propagate through the call stack when not handled;
- custom exceptions can model domain-specific problems;
- try-with-resources automatically closes resources;
- exception wrapping can preserve low-level causes;
- defensive programming reduces invalid states and improves reliability.

32. Review Questions

1. What is an exception?
2. What is the difference between a compile-time error and a runtime exception?
3. What is the root class of Java exceptions and errors?
4. What is the difference between `Error` and `Exception`?
5. What is a checked exception?
6. What is an unchecked exception?
7. Name five common unchecked exceptions.
8. What is the purpose of a `try` block?
9. What is the purpose of a `catch` block?
10. What is the purpose of `finally`?
11. Why should specific catch blocks appear before general catch blocks?
12. What is multi-catch?
13. What does `throw` do?
14. What does `throws` do?
15. What is the difference between `throw` and `throws`?
16. What is exception propagation?
17. What is a stack trace?
18. What does `getMessage()` return?
19. How is a custom checked exception created?
20. How is a custom unchecked exception created?
21. When should a custom exception be used?
22. What is exception wrapping?
23. Why should the original cause be preserved?
24. What is try-with-resources?
25. Which interface must a resource implement?
26. Why is try-with-resources preferred for many resources?
27. What is defensive programming?
28. Why are empty catch blocks dangerous?
29. Why should exceptions not be used for normal control flow?
30. What is an assertion?
31. When should assertions not be used?

33. Programming Exercises

Beginner Level

1. Handle division by zero.
2. Handle an invalid array index.
3. Handle invalid integer conversion with `Integer.parseInt()`.
4. Print the message of an exception.
5. Use a `finally` block in a small program.
6. Create two catch blocks for two different exception types.
7. Rewrite two catch blocks using multi-catch.
8. Throw an `IllegalArgumentException` for a negative age.
9. Throw an exception for a negative product price.
10. Create a method that declares `throws Exception`.

Intermediate Level

1. Create `InvalidAgeException`.
2. Create `InvalidGpaException`.
3. Create `InsufficientFundsException`.
4. Use a custom exception in a `BankAccount` class.
5. Demonstrate exception propagation through three methods.
6. Wrap a lower-level exception inside a higher-level exception.
7. Create a calculator that continues after invalid input.
8. Create a password validator using exceptions.
9. Create a course registration validator with a custom exception.
10. Use try-with-resources with a simple reader.

Advanced Level

1. Design a custom exception hierarchy for a banking application.
2. Design a custom exception hierarchy for a university registration system.
3. Create a robust ATM simulation with several failure conditions.
4. Build an input-validation framework using reusable methods.
5. Refactor a program that uses error codes into one that uses exceptions.
6. Create a service layer that translates low-level exceptions into domain exceptions.
7. Build a console application that recovers from invalid user input without terminating.
8. Create a transaction-like simulation that performs cleanup in all execution paths.
9. Create a file-processing application using try-with-resources and specific exception handling.
10. Develop a student registration system that handles duplicate IDs, invalid GPA values, and unavailable courses.

34. Homework

Task: Build a Robust Student Registration Application

Create a console-based application with the following classes:

Illustration

```
Student
Course
RegistrationService
InvalidStudentException
InvalidCourseException
RegistrationException
Main
```

Requirements:

1. `Student` must validate:
   - non-empty name;
   - age between 16 and 100;
   - GPA between 0 and 4.
2. `Course` must validate:
   - non-empty course name;
   - capacity greater than zero.
3. `RegistrationService` must:

- register students;
- reject registration when a course is full;
- reject duplicate registration;
- throw meaningful custom exceptions.

4. `Main` must:
   - demonstrate at least three successful registrations;
   - demonstrate at least three different exception cases;
   - handle exceptions without terminating the whole program;
   - display clear user-friendly messages.
5. At least one exception must be wrapped with a higher-level message.
6. At least one resource-handling example must use try-with-resources.

Submission requirements:

- all Java source files;
- program output;
- short explanation of each custom exception;
- explanation of where exception propagation occurs;
- explanation of where recovery is performed;
- explanation of how defensive programming improves the design.

35. Key Terms

Exception
An object representing an abnormal runtime condition.

Checked exception
An exception the compiler requires to be handled or declared.

Unchecked exception
A runtime exception not subject to compile-time handling rules.

`try` Block containing code that may throw an exception.

`catch`
Block that handles a matching exception.

`finally`
Block normally executed after try/catch processing.

`throw`
Statement that explicitly throws an exception object.

`throws`
Method declaration clause indicating possible exception propagation.

Propagation
Movement of an unhandled exception up the call stack.

Custom exception
A programmer-defined exception type.

Multi-catch
One catch block handling several exception types.

Try-with-resources
Construct that automatically closes resources.

Stack trace
Diagnostic record of method calls leading to an exception.

Exception wrapping
Throwing a higher-level exception while preserving the original cause.

Defensive programming
Designing code to anticipate invalid inputs and unexpected states.