CMPE 101
Object Oriented Programming

İstanbul
Bilgi University

Dr. Emel Küpçü

Department of Computer Engineering

İstanbul Bilgi University

Week-12: Exception Handling and File Input & Output Operations

7.5 Exception Handling: Processing the Incorrect Response

- **An exception indicates a problem that occurs while a program executes.**
- **The term "exception" implies that problems occur rarely—if the normal rule is that a statement executes correctly, then an exception represents a rare case where this rule is broken.**
- Exception is an object which is thrown at runtime.
- **Exception handling helps you create fault-tolerant programs that can resolve (or handle) exceptions.**
  - Programs that can keep running even when unexpected errors occur. Instead of crashing, the program can **catch** the error, **respond** to it appropriately (like showing an error message or trying a different solution), and continue operating. This way, exceptions are either **resolved** immediately or managed in a way that minimizes the impact on the user or system.
  - When the JVM or a method detects a problem, such as an invalid array index or an invalid method argument, it throws an exception—that is, an exception occurs.
- **Exception Handling is a mechanism to handle runtime errors.**
  - **Example**: ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Types of Errors

- There are three categories of errors:
  - **Syntax errors** - arise because the rules of the language have not been followed. They are detected by the compiler.
  - **Runtime errors** - occur while the program is running if the environment detects an operation that is impossible to carry out.
  - **Logic errors** - occur when a program doesn't perform the way it was intended to.

https://www.slideshare.net/slideshow/exception-handling-in-java-presentation/265830497?from_search=2#6

Types of Exception

There are mainly two types of exceptions:

- **Checked Exceptions**
  - They are checked at **compile-time**
  - They must either be caught or declared using **throws** keyword.
  - **Example**: IOException - Issues with input/output operations.
- **Unchecked Exceptions**
  - They are checked at **runtime.**
  - They do not require explicit handling or declaration, often caused by programming errors.
  - **Example**: ArithmeticException - For errors like division by zero.
- **Error**
  - Error represents serious problems that applications are generally not expected to catch. These errors are usually caused by issues in the JVM itself or external environment problems (e.g., out of memory, JVM crashes).
  - **Examples**: OutOfMemoryError – memory issues

https://www.slideshare.net/slideshow/exception-handling-in-java-presentation/265830497?from_search=2#6

Throwable's inheritance Hierarchy

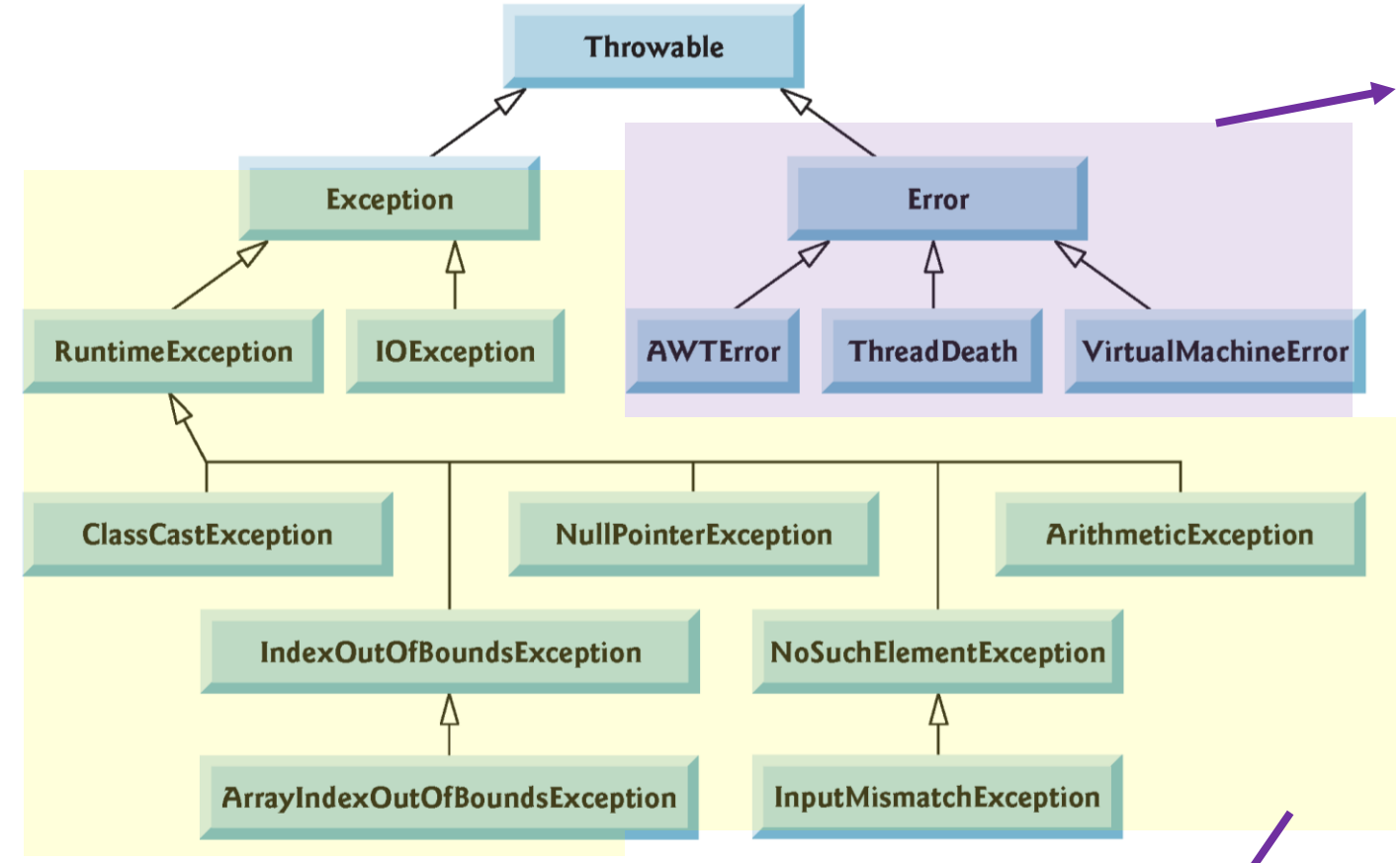


Fig. 11.4 | Portion of class `Throwable`'s inheritance hierarchy.

- System errors are thrown by JVM and represented in the Error class.
- The **Error** class describes **internal system errors**. Such errors rarely occur.

- **Exception** describes **errors caused by your program and external circumstances.**
- These errors can caught and handled by your program.

Advantages of Exception Handling

- Systematic Error Handling
  - It provides a structured approach to detect, report, and handle runtime errors properly.
- Clean Separation
  - You don't have to mix the normal logic with error-handling logic.
  - You can use try-catch blocks to keep your main code clean and readable.
- Error Propagation
  - Exceptions can be propagated up the call stack, allowing higher-level methods to decide how to handle the errors.
- Better Debugging
  - Exception messages and stack traces help developers quickly locate and understand the cause of errors.
- Flexible Handling
  - Different types of exceptions can be caught and handled differently based on the specific problem.
- Custom Exceptions
  - Developers can define their own custom exceptions to represent specific application-related issues more clearly.
- Resource Management
  - The finally block or try-with-resources ensures that important system resources are properly released, even if an error occurs.

7.5.1 The try Statement

- To handle an exception, place any code that might throw an exception in a `try` statement.
- **The `try` block contains the code that might throw an exception.**
- **The `catch` block contains the code that *handles* the exception if one occurs.** You can have many `catch` blocks to handle different *types* of exceptions that might be thrown in the corresponding `try` block.
- General Format:
  - ```
    try {
        // Code that might throw an exception
    } catch (ExceptionType1 e1) {
        // Code to handle ExceptionType1
    } catch (ExceptionType2 e2) {
        // Code to handle ExceptionType2
    } finally {
        // (Optional) Code that always runs after try and catch block
    }
    ```
    ```

Exception Handling Terms

- In Java, the **try** and **catch** blocks are used for **exception handling**. They allow you to **handle runtime errors (exceptions) in a controlled manner, ensuring your program doesn't crash unexpectedly**.
- **try :** The code that might throw an exception is placed inside the **try** block.
  - If an exception occurs, the control is immediately transferred to the catch block.
  - We can't use try block alone. The try block must be followed by either catch or finally
- **catch:** The "catch" block is used to handle the exception.
  - It must be preceded by try block which means we can't use catch block alone.
  - It can be followed by finally block later
- **throw:** The "throw" keyword is used to throw an exception.
- **Throws:** If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
  - Type **method-name** (parameter-list) **throws** exception-list

    //The body of method
- **finally:** (optional) The "finally" block is used to execute the necessary code of the program.
  - It is executed whether an exception is handled or not
  - Useful for **closing resources** (files, sockets).

7.5.2 Executing the catch Block

```
public class SurveyResults_Error {
   public static void main(String[] args) {
      int[] responses = {1, 2, 5, 3, 5, 2, 14, 4}; // 14 is invalid
      int[] frequency = new int[6]; // valid indexes: 0–5

      for (int answer : responses) {
         frequency[answer]++;
      }

      System.out.println("\nFrequency of valid responses:");
      for (int i = 1; i < frequency.length; i++) {
         System.out.println(i + ": " + frequency[i]);
      }
   }
}
```

Output:
Exception in thread "main"
java.lang.**ArrayIndexOutOfBoundsException**: Index 14 out of bounds for length 6 at
SurveyResults_Error.main(SurveyResults_Error.java:7)

Step	Response	Action	equency array after step
1	1	`frequency[1]++`	`[0, 1, 0, 0, 0, 0]`
2	2	`frequency[2]++`	`[0, 1, 1, 0, 0, 0]`
3	5	`frequency[5]++`	`[0, 1, 1, 0, 0, 1]`
4	3	`frequency[3]++`	`[0, 1, 1, 1, 0, 1]`
5	5	`frequency[5]++` again	`[0, 1, 1, 1, 0, 2]`
6	2	`frequency[2]++` again	`[0, 1, 2, 1, 0, 2]`
7	14	Exception	

7.5.2 Executing the catch Block

- When the program encounters the invalid value 14 in the responses array, it attempts to add 1 to `frequency[14]`, which is *outside* the bounds of the array—the `frequency` array has only six elements (with indexes 0–5).
- Because array bounds checking is performed at execution time, the JVM generates an *exception*—specifically an `ArrayIndexOutOfBoundsException` to notify the program of this problem.
- At this point, the try block ends and the catch block starts running.

```
public class SurveyResults {
   public static void main(String[] args) {
      int[] responses = {1, 2, 5, 3, 5, 2, 14, 4}; // 14 is invalid
      int[] frequency = new int[6]; // valid indexes: 0–5

      for (int answer : responses) {
         try {
            frequency[answer]++;
         } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Invalid response: " + answer);
         }
      }
      System.out.println("\nFrequency of valid responses:");
      for (int i = 1; i < frequency.length; i++) {
         System.out.println(i + ": " + frequency[i]);
      }
   }
}
```

7.5.2 Executing the catch Block

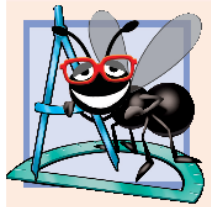
Step	Response	Action	equency array after step
1	1	`frequency[1]++`	`[0, 1, 0, 0, 0, 0]`
2	2	`frequency[2]++`	`[0, 1, 1, 0, 0, 0]`
3	5	`frequency[5]++`	`[0, 1, 1, 0, 0, 1]`
4	3	`frequency[3]++`	`[0, 1, 1, 1, 0, 1]`
5	5	`frequency[5]++` again	`[0, 1, 1, 1, 0, 2]`
6	2	`frequency[2]++` again	`[0, 1, 2, 1, 0, 2]`
7	14	invalid → exception → no change	
8	4	`frequency[4]++`	`[0, 1, 2, 1, 1, 2]`

```
public class SurveyResults {
   public static void main(String[] args) {
      int[] responses = {1, 2, 5, 3, 5, 2, 14, 4}; // 14 is invalid
      int[] frequency = new int[6]; // valid indexes: 0–5
      for (int answer : responses) {
         try {
            frequency[answer]++;
         } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Invalid response: " + answer);
         }
      }
      System.out.println("\nFrequency of valid responses:");
      for (int i = 1; i < frequency.length; i++) {
         System.out.println(i + ": " + frequency[i]);
      }
   }
}
```

Output:
Invalid response: 14
Frequency of valid responses:
1: 1
2: 2
3: 1
4: 1
5: 2

Error-Prevention Tip 7.1

When writing code to access an array element, ensure that the array index remains greater than or equal to 0 and less than the length of the array. This will prevent `ArrayIndexOutOfBoundsExceptions` if your program is correct.



Software Engineering Observation 7.1

Systems in industry that have undergone extensive testing are still likely to contain bugs. Our preference for industrial-strength systems is to catch and deal with runtime exceptions, such as `ArrayIndexOutOfBoundsExceptions`, to ensure that a system either stays up and running or degrades gracefully, and to inform the system's developers of the problem.

Example – Integer Division Without Exception Handling

```
import java.util.Scanner;
public class DivideByZeroNoExceptionHandling {
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient(int numerator, int denominator) {
      return numerator / denominator; // possible division by zero
   }
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);

      System.out.print("Please enter an integer numerator: ");
      int numerator = scanner.nextInt();
      System.out.print("Please enter an integer denominator: ");
      int denominator = scanner.nextInt();

      int result = quotient(numerator, denominator);
      System.out.printf("%nResult: " +
            "%d / %d = %d%n", numerator, denominator, result);
   }
}
```

Output:
Please enter an integer numerator: 4
Please enter an integer denominator: 2

Result: 4 / 2 = 2

Example – Integer Division Without Exception Handling

Output:

```
Please enter an integer numerator: 4
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(DivideByZeroNoExceptionHandling.java:8)
        at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:19)
```

Output:

```
Please enter an integer numerator: 4
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:964)
        at java.base/java.util.Scanner.next(Scanner.java:1619)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2284)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2238)
        at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:17)
```

Example - Handling ArithmeticExceptions and InputMismatchExceptions

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling {
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient(int numerator, int denominator) {
      return numerator / denominator; // possible division by zero
   }

   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      try {
         System.out.print("Please enter an integer numerator: ");
         int numerator = scanner.nextInt();

         System.out.print("Please enter an integer denominator: ");
         int denominator = scanner.nextInt();

         int result = quotient(numerator, denominator);
         System.out.printf("%nResult: %d / %d = %d%n", numerator, denominator, result);

      }
```

Example - Handling ArithmeticExceptions and InputMismatchExceptions

```
    catch (InputMismatchException inputMismatchException) {
            System.out.println("Error: You must enter integers. Please try again.");
    }
    catch (ArithmeticException arithmeticException) {
            System.out.println("Error: Cannot divide by zero. Please try again.");
    }
    scanner.close(); // to properly close the scanner at the end.
  }
}
```

Output:
Please enter an integer numerator: 4
Please enter an integer denominator: 2

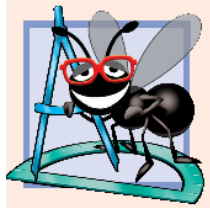
Result: 4 / 2 = 2

Output:
Please enter an integer numerator: 4
Please enter an integer denominator: 0
Error: Cannot divide by zero. Please try again.

Output:
Please enter an integer numerator: 4
Please enter an integer denominator: hello
Error: You must enter integers. Please try again.

Common Programming Error 11.1

It's a syntax error to place code between a `try` block and its corresponding `catch` blocks.



Software Engineering Observation 11.6

Sometimes you can prevent an exception by validating data first. For example, before you perform integer division, you can ensure that the denominator is not zero, which prevents the `ArithmeticException` that occurs when you divide by zero.

The Finally Block

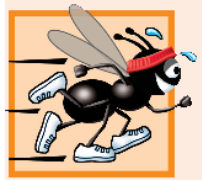
```
import java.util.Scanner;

public class DivideToNumber {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Enter a number to divide 100 by: ");
            int number = scanner.nextInt();
            int result = 100 / number;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        } catch (Exception e) {
            System.out.println("Invalid input! Please enter an integer.");
        } finally {
            System.out.println("Program ended.");
            scanner.close();
        }
    }
}
```

- **Program doesn't crash on wrong user input.**
  - Without try-catch, if the user types "hello" instead of a number, the program would throw an ugly error and stop.
  - With try-catch, you catch the error and show a friendly message to the user.
- **Specific Error Handling:**
  - Catching ArithmeticException separately lets you give a clear message for division by zero.
  - Catching a general Exception handles any other unexpected issues safely.
- **Makes the Program More Robust and User-Friendly**
  - Even if users make mistakes (zero input or wrong data), the program handles it gracefully without crashing.
  - Gives the user useful feedback instead of showing a technical stack trace.
- **Guaranteed Cleanup**
  - The **finally** block always executes, whether an exception happens or not.
  - **The finally is useful for closing resources (files, sockets).**
  - In this example: scanner.close() ensures that the input scanner is always closed, preventing resource leaks.

Error-Prevention Tip 11.5

The `finally` block is an ideal place to release resources acquired in a `try` block (such as opened files), which helps eliminate resource leaks.



Performance Tip 11.1

Always release a resource explicitly and at the earliest possible moment at which it's no longer needed. This makes resources available for reuse as early as possible, thus improving resource utilization and program performance.

Throwing Exceptions (throw Keyword)

```
public class CheckAgeExample {
    static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Access denied " +
                    "- You must be 18 or older.");
        } else {
            System.out.println("Access granted!");
        }
    }
    public static void main(String[] args) {
        checkAge(16);
    }
}
```

After a **throw** statement, **the method immediately stops executing — no code after the throw is executed unless caught by a try-catch.**

If the age is **less than 18**, **the code don't just print an error** — It **immediately stops execution** by **throwing** an **exception**.

throw **creates** (or "throws") a new ArithmeticException object with a **custom error message**:
"Access denied - You must be 18 or older."

When Java **throws** an exception, it:

- **Interrupts** the normal flow of the program.
- **Jumps out** of the method where the error occurred.
- If the exception isn't caught (and here it isn't), the program **crashes** and prints an error message.
- If age is **18 or older**, we simply print "Access granted!", and no exception is thrown.

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be 18 or older.
        at CheckAgeExample.checkAge(CheckAgeExample.java:4)
        at CheckAgeExample.main(CheckAgeExample.java:11)
```

Declaring Exceptions (throws Keyword) - Throwing Exceptions (throw)

```
public class WithdrawExample {

    static void withdraw(double amount) throws Exception {
        if (amount > 500) {
            throw new Exception("Withdrawal limit exceeded!");
        } else {
            System.out.println("Withdrawal of $"
                    + amount + " successful.");
        }
    }

    public static void main(String[] args) {
        try {
            withdraw(600); // Try to withdraw more than the limit
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output: Error: Withdrawal limit exceeded!

Throw :

- Actually **creates** and **throws** an Exception object during program execution.
- throw is the keyword used inside a method to actually throw an exception.
- throw new Exception("Something went wrong!");

Throws :

- throws is used in the method signature to declare that a method might throw an exception.
- public void myMethod() throws IOException {
- it's a warning to whoever calls the method.
- Whoever calls withdraw() must either:

➔ Handle the Exception using try-catch,
➔ Or declare throws Exception again if not handling.

```
public static void main(String[] args) throws Exception {
    withdraw(600);  // No try-catch here
}
```

You will get a *compile-time error*.

Exception Catching Example

```
public class WithdrawExample2 {
    // Lower level method - actually throws the Exception
    static void withdraw(double amount) throws Exception {
        if (amount > 500) {
            throw new Exception("Withdrawal limit exceeded!");
        }
        System.out.println("Withdrawal of $" + amount + " successful.");
    }
    // Middle level method - calls withdraw()
    //but just declares throws Exception
    static void bankTransaction(double amount) throws Exception {
        withdraw(amount);  // No try-catch here, just passing the exception upward
    }
    // Top level method - main() catches and handles the Exception
    public static void main(String[] args) {
        try {
            bankTransaction(600);  // Attempting to withdraw more than limit
        } catch (Exception e) {
            System.out.println("Transaction failed: " + e.getMessage());
        }
    }
}
```

Output: Transaction failed: Withdrawal limit exceeded!

- Sometimes you are not ready to handle exceptions immediately.

You might want to pass the problem to a higher-level method that can handle it better.

withdraw(double amount)

- If invalid, it throws a new Exception. So it must declare throws Exception.

bankTransaction(double amount)

- But it does not handle the Exception using try-catch. Therefore, it must also declare throws Exception to pass the responsibility upward to whoever called bankTransaction().

main()

- Finally handles the Exception with try-catch.
- So no need for throws in main() (you could also declare throws Exception in main(), but here you decided to catch).

File Input & Output Operations

- **File Input/Output (I/O)** in Java is essential for reading from and writing to files.
- Java provides classes in the **java.io** and **java.nio.file** packages to handle these operations.
- What is File I/O?
  - File I/O refers to the process of:
    - **Reading**: Taking data from a file and bringing it into a program.
    - **Writing**: Sending data from a program to a file for storage.
- **Why File I/O?**
  - To store data persistently.
  - To share data between programs.
  - To process large datasets not suitable for in-memory use.
- **Commonly Used Classes for File Operations**:

Operation	Class	Package
Reading	FileReader, BufferedReader	java.io
Writing	FileWriter, BufferedWriter	java.io
Path Management	Files, Paths, Path	java.nio.file

File Reading

example.txt:
hello students!
How are you?

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class FileReadExample {
    public static void main(String[] args) {
        String filePath = "example.txt";

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

BufferedReader ➔ Helps **read text efficiently**, line by line.
FileReader ➔ **Connects** to the file and **reads its characters**.
IOException ➔ For **handling input/output errors** (e.g., file not found)

- Inside try(...), you can **create or pass resources** (like file readers, streams, sockets, database connections).
- **Java will automatically call .close()** on them when the block finishes — even if an exception happens.
- No need to manually close the resource!

- **Opens the file safely** using a **try-with-resources** block:
  - FileReader(filePath) ➔ **Opens** the file.
  - BufferedReader ➔ **Wraps** the FileReader for **faster reading** line by line.
- try-with-resources ensures that the reader will **automatically close** after the block is done, even if an error occurs.

reader.readLine() ➔ Reads **one line**.

- **Catches any IOException** that happens during file reading:
  - For example, file not found, file permission denied, etc.

Prints error details if something goes wrong (helpful for debugging).

File Writing

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class FileWriteExample {
    public static void main(String[] args) {
        String filePath = "output.txt";

        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
            writer.write("Hello, World!");
            writer.newLine();
            writer.write("This is a file writing example.");
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

- BufferedWriter ➔ Writes text to a file efficiently (with buffering).
- FileWriter ➔ Connects to a file for writing raw text.

- **Opens the file safely** using a **try-with-resources** block:
- FileWriter(filePath) ➔ Creates a connection to the file.
- BufferedWriter ➔ Wraps FileWriter for **faster writing**.
- **try**-with-resources automatically **closes** the writer when done.

- **Writes** "Hello, World!" to the file.
- **Inserts a new line** (like pressing Enter) after the first line.
- **Writes another sentence** after the new line into the file.

Some Notes:

- Always close files (or use try-with-resources).
- Handle exceptions (like FileNotFoundException, IOException).

Group Discussion Question

Student Grades Processing

- Create a Java program that processes a list of student names and their corresponding scores.
- students = {"Alice", "Bob", "Charlie", "David", "Eve"}
- scores = {85, 92, 77, 105, 60}
- The program should write valid student names and scores to a text file, skipping any invalid scores (scores outside the range of 0 to 100). Additionally, the program should print the number of valid and handle potential file writing errors. All of this will be processed inside the main method.

Console Output:

Skipping invalid score for David: Invalid score: 105

4 valid records written successfully!

1 invalid records skipped.

File (grades_output.txt) Output:

Alice,85

. . .

Define the Output File Path and Data:

- Declare a String variable to store the path of the output file (grades_output.txt).
- Create an array for student names (students) and another array for their corresponding scores (scores).

Initialize Counters:

- Declare two integer variables (validCount and invalidCount) to keep track of the number of valid and invalid records processed.

Set Up BufferedWriter:

- Use a BufferedWriter wrapped around a FileWriter to write the student records to the output file. This is done inside a try-with-resources block to ensure that the file is properly closed after writing.

Iterate Over the Students and Scores:

- Use a for loop to iterate through each student and their score.
- Inside the loop, check if the score is valid (i.e., between 0 and 100). If it's invalid, throw an IllegalArgumentException.

Write Valid Records to the File:

- For valid scores, write the student name and score to the output file, separated by a comma, and move to the next line after each entry.
- Increment the validCount for each valid record.

Handle Invalid Records:

- If an invalid score is encountered, catch the IllegalArgumentException, print a message indicating the invalid score, and increment the invalidCount.

Handle File Writing Errors:

- Use a catch block to catch any IOException that may occur when opening or writing to the file, and print an error message if this happens.

Output the Results:

- After processing all records, print how many valid and invalid records were processed.

Questions?