Coding Cheat Sheet

This reading provides a reference list of code that you'll encounter as you work with object-oriented coding in Java. Understanding these concepts will help you write and debug your first Java programs. Let's explore the following Java coding concepts:

- Java File Handling / Working with File Input and Output Streams
- Using Java Byte Streams
- Managing Directories in Java

Keep this summary reading available as a reference as you progress through your course, and refer to this reading as you begin coding with Java after this course!

Java File Handling / Working with File Input and Output Streams

Using the File class

Description	Example
Import the `File` class, which provides methods for file and directory operations.	`import java.io.File;`

Define a class `FileExample` that contains the Java `main` method. Create a `File` object representing a file named example.txt. This does not create the actual file, just a reference to it. Call the `exists()` method on the `File` object to check whether the file physically exists in the specified location. If the file exists, prints "File exists.", otherwise print "File does not exist.".	`public class FileExample {`
	`    public static void main(String[] args) {`
	`        File myFile = new File("example.txt");`
	
	`        // Check if the file exists`
	`        if (myFile.exists()) {`
	`            System.out.println("File exists.");`
	`        } else {`
	`            System.out.println("File does not exist.");`
	`        }`

Close curly braces to end the `FileExample` class definition.	`    }`
	`}`

Explanation: This Java program demonstrates how to check whether a file exists in the filesystem using the `File` class from the java.io package.

Writing to Files

Description	Example

Import the `FileWriter` class for writing character data to a file, the `BufferedWriter` class that wraps `FileWriter` to provide efficient writing operations, and the `IOException` class to handle input/output exceptions.	`import java.io.BufferedWriter;`
	`import java.io.FileWriter;`
	`import java.io.IOException;`

Define a class `WriteToFile` that contains the Java `main` method. Create a `FileWriter` class to write to the file "output.txt". A `BufferedWriter` is wrapped around `FileWriter` for more efficient writing. Write text to the file using the `write()` method. The `newLine()` method inserts a newline character (\n). The `close()` method closes the writer to ensure all data is flushed to the file. A confirmation message is printed to the console. The `catch()` call catches `IOException` if any file operation fails (for example, permission issues, disk space) and prints an error message.	`public class WriteToFile {`
	`    public static void main(String[] args) {`
	`        try {`
	`            FileWriter writer = new FileWriter("output.txt");`
	`            BufferedWriter bufferedWriter = new BufferedWriter(writer);`
	
	`            bufferedWriter.write("Hello, World!");`
	`            bufferedWriter.newLine(); // Adds a new line`
	`            bufferedWriter.write("This is a Java file handling example.");`

Description	Example

	`        bufferedWriter.close(); // Always close the writer`
	`        System.out.println("Data written to file successfully.");`
	`    } catch (IOException e) {`
	`        System.out.println("An error occurred: " + e.getMessage());`
	`    }`

Close curly braces to end the `WriteToFile` class definition.	`    }`
	`}`

Explanation: This Java program demonstrates how to write text to a file using the `FileWriter` and `BufferedWriter` package. It writes multiple lines to the file, handles exceptions properly, and closes the file to prevent resource leaks.

Reading from Files

Description	Example

Import the `FileReader` class that reads character-based data from a file, the `BufferedReader` class that provides efficient reading capabilities by buffering input, and the `IOException` class to handle errors that may occur during file operations.	`import java.io.BufferedReader;`
	`import java.io.FileReader;`
	`import java.io.IOException;`

Define a class `ReadFromFile` that contains the Java `main` method. Create a `FileReader` class to read the file "output.txt". A `BufferedReader` is wrapped around `FileReader` for more efficient reading. Call `readLine()` reads one line at a time from the file. The loop continues until `readLine()` returns null (indicating the end of the file). Each line is printed to the console. The `bufferedReader.close()` method ensures the file resource is released after reading is complete. The `catch()` call catches `IOException` if any file operation fails (for example, permission issues, disk space) and prints an error message.	`public class ReadFromFile {`
	`    public static void main(String[] args) {`
	`        try {`
	`            FileReader reader = new FileReader("output.txt");`
	`            BufferedReader bufferedReader = new BufferedReader(reader);`
	
	`            String line;`
	`            while ((line = bufferedReader.readLine()) != null) {`
	`                System.out.println(line);`
	`            }`
	
	`            bufferedReader.close();`
	`        } catch (IOException e) {`
	`            System.out.println("An error occurred: " + e.getMessage());`
	`        }`

Close curly braces to end the `FileExample` class definition.	`    }`
	`}`

Explanation: This Java program reads a file line by line using `FileReader` and `BufferedReader` and prints its content to the console. It reads and prints lines the file line by line, handles exceptions properly, and closes the file to prevent resource leaks.

Using Java Byte Streams

Reading bytes

Description	Example

Import the `FileInputStream` class for reading raw byte data from a file and the `IOException` class to handle input/output exceptions.	`import java.io.FileInputStream;`
	`import java.io.IOException;`

Define a class `ReadBytes` that contains the Java `main` method. Declare a `FileInputStream` variable, but don't initialize it. Open "example.txt" for reading. Read one byte at a time until the end of the file is reached. The method `byteData()` converts the byte into a character and prints it. If an I/O error occurs, an error stack trace is printed. The `finally` block ensures the file stream is closed, preventing resource leaks. The method `fileInputStream.close()` closes the file to free system resources.	`public class ReadBytes {`
	`    public static void main(String[] args) {`
	`        FileInputStream fileInputStream = null;`
	`        try {`
	`            // Create a FileInputStream to read from a file`
	`            fileInputStream = new FileInputStream("example.txt");`
	
	`            // Variable to hold the byte data`
	`            int byteData;`
	`            // Read bytes until end of file`
	`            while ((byteData = fileInputStream.read()) != -1) {`
	`                // Print the byte data as characters`
	`                System.out.print((char) byteData);`
	`            }`
	`        } catch (IOException e) {`
	`            e.printStackTrace();`
	`        } finally {`
	`            // Close the stream to free resources`
	`            if (fileInputStream != null) {`
	`                try {`
	`                    fileInputStream.close();`
	`                } catch (IOException e) {`
	`                    e.printStackTrace();`
	`                }`
	`            }`
	`        }`

Close curly braces to end the `FileExample` class definition.	`    }`
	`}`

Explanation: This Java program reads a file byte by byte using `FileInputStream` and prints its contents to the console.

Writing bytes

Description	Example

Import the `FileOutputStream` class for writing raw byte data to a file and the `IOException` class to handle input/output exceptions.	`import java.io.FileOutputStream;`
	`import java.io.IOException;`

Define a class `WriteBytes` that contains the Java `main` method. Declare an `FileOutputStream` variable but don't initialize it. Open a `FileOutputStream` for the file "output.txt". If the file does not exist, create a new one. Define a String ("Hello, World!") to write to the file. Convert the string into a byte array using `.getBytes()`. Write the byte array to the file using `fileOutputStream.write(byteData)`. The `IOException` method catches and prints any exceptions during file writing. The `finally` block ensures that the `FileOutputStream` is properly closed to free system resources and uses a null check before calling `.close()`, preventing a `NullPointerException`. If closing the stream fails, it prints the exception.	`public class WriteBytes {`
	`    public static void main(String[] args) {`
	`       FileOutputStream fileOutputStream = null;`
	`        try {`
	`            // Create a FileOutputStream to write to a file`
	`            fileOutputStream = new FileOutputStream("output.txt");`
	
	`            // Data to write`
	`            String data = "Hello, World!";`
	`            // Convert the string to bytes`
	`            byte[] byteData = data.getBytes();`
	
	`            // Write bytes to the file`

Description	Example

	`fileOutputStream.write(byteData);`
	`} catch (IOException e) {`
	`    e.printStackTrace();`
	`} finally {`
	`    // Close the stream to free resources`
	`    if (fileOutputStream != null) {`
	`        try {`
	`            fileOutputStream.close();`
	`        } catch (IOException e) {`
	`            e.printStackTrace();`
	`        }`
	`    }`
	`}`

Close curly braces to end the `FileExample` class definition.	`    }`
	`}`

Explanation: This Java program, writes the string "Hello, World!" to a file named output.txt using a `FileOutputStream`. It uses exception handling to catch possible file operation errors and uses a `finally` block to ensure the file stream is always closed.

Byte streams example

Description	Example

Import the `FileInputStream` class for reading faw byte data from a file, `FileOutputStream` class for writing raw byte data to a file, and the `IOException` class to handle input/output exceptions.	`import java.io.FileInputStream;`
	`import java.io.FileOutputStream;`
	`import java.io.IOException;`

Decleare `FileInputStream inputFile` to reads data from source.txt and `FileOutputStream outputFile` to write data to destination.txt. The try block initializes `inputFile` to read from source.txt, initializes `outputFile` to write to destination.txt, reads bytes from source.txt one byte at a time using `inputFile.read()`, writes each byte to destination.txt using `outputFile.write(byteData)`, continues until reaching the end of the file (-1), and prints "File copied successfully!" after completion. The catch block prints the stack trace if an `IOException` occurs (for example, file not found, read/write error). The `finally` bock ensures both `inputFile` and `outputFile` are closed to free system resources. It uses null checks to prevent `NullPointerException`.	`public class FileCopy {`
	`    public static void main(String[] args) {`
	`        FileInputStream inputFile = null;`
	`        FileOutputStream outputFile = null;`
	
	`        try {`
	`            // Create FileInputStream to read from "source.txt"`
	`            inputFile = new FileInputStream("source.txt");`
	`            // Create FileOutputStream to write to "destination.txt"`
	`            outputFile = new FileOutputStream("destination.txt");`
	
	`            int byteData;`
	`            // Read bytes from source and write them to destination`
	`            while ((byteData = inputFile.read()) != -1) {`
	`                outputFile.write(byteData);`
	`            }`
	
	`            System.out.println("File copied successfully!");`
	
	`        } catch (IOException e) {`
	`            e.printStackTrace();`
	`        } finally {`
	`            // Close both streams`
	`            try {`
	`                if (inputFile != null) inputFile.close();`
	`                if (outputFile != null) outputFile.close();`
	`            } catch (IOException e) {`
	`                e.printStackTrace();`
	`            }`
	`        }`

Description	Example

Close curly braces to end the `FileCopy` class definition.	
	```
	    }
	}
	```

Explanation: This Java program copies the contents of a file named source.txt into another file named destination.txt using `FileInputStream` and `FileOutputStream`. It reads and writes files one byte at a time and uses `finally` to always close file streams. The program catches `IOException` to prevent crashes.

Managing Directories in Java

Creating a directory

Description	Example
Import the `java.io.File` package to represent file and directory paths.	`import java.io.File;`

Define a class `CreateDirectory` that contains the Java `main` method. The `String directoryPath = "Projects/Java"` specifies the directory to be created. This means that the program will try to create a folder named "Java" inside a folder named "Projects". Create a `File` object for the directory by calling the `File(directoryPath)` method. The File object represents the directory but does not create it yet. The `if (!directory.exists())` method ensures the directory is created only if it does not already exist. The method `mkdirs()` ensures all parent directories are also created. If creation is successful, the message "Directory created successfully: Projects/Java" is printed to the console. If creation fails, the message "Failed to create directory" is printed to the console. If the directory already exists, the message "Directory aready exists: Projects/Java" is printed to the console.	
	```
	public class CreateDirectory {
	    public static void main(String[] args) {
	        // Define the directory path
	        String directoryPath = "Projects/Java";

	        // Create a File object
	        File directory = new File(directoryPath);

	        // Create the directory
	        if (!directory.exists()) {
	            boolean created = directory.mkdirs(); // Use mkdirs() to create nested directories
	            if (created) {
	                System.out.println("Directory created successfully: " + directoryPath);
	            } else {
	                System.out.println("Failed to create directory.");
	            }
	        } else {
	            System.out.println("Directory already exists: " + directoryPath);
	        }
	```

Close curly braces to end the `CreateDirectory` class definition.	
	```
	    }
	}
	```

Explanation: This Java program creates a directory (including nested directories) if it does not already exist. It handles success and failure cases gracefully.

Listing directory contents

Description	Example
Import the `java.io.File` package to represent file and directory paths.	`import java.io.File;`

Description	**Example**

Define a class `ListDirectoryContents` that contains the Java `main` method. The `String directoryPath = "Projects/Java"` specifies the directory whose contents will be listed. Create a `File` object for the directory by calling the `File(directoryPath)` method. The File object represents the directory but does not perform any operations yet. The `directory.list()` method returns an array of filenames that exist in the directory. If the directory does not exist or is empty, list() returns null. The `if (contents != null)` method ensures the directory exists and is not empty before proceeding. If contents is null, it prints: "The directory is empty or does not exist." If the directory contains files/subdirectories, the program prints "Contents of Projects/Java:", iterates through the `contents` array and prints each filename.	
	```
	public class ListDirectoryContents {
	    public static void main(String[] args) {
	        String directoryPath = "Projects/Java";
	        File directory = new File(directoryPath);

	        // List all files and directories in the specified directory
	        String[] contents = directory.list();

	        if (contents != null) {
	            System.out.println("Contents of " + directoryPath + ":");
	            for (String fileName : contents) {
	                System.out.println(fileName);
	            }
	        } else {
	            System.out.println("The directory is empty or does not exist.");
	        }
	```

Close curly braces to end the `ListDirectoryContents` class definition.	
	```
	    }
	}
	```

Explanation: This Java program lists all files and subdirectories inside a directory and handles cases where the directory is empty or does not exist
It uses the `File.list()` method to retrieve directory contents efficiently.

Deleting a directory

Description	**Example**
Import the `java.io.File` package to represent file and directory paths.	`import java.io.File;`

Define a class `ListDirectoryContents` that contains the Java `main` method. The `String directoryPath = "Projects/Java"` specifies the directory to be deleted. Create a `File` object for the directory by calling the `File(directoryPath)` method. The File object represents the directory but does not perform any operations yet. The `if (directory.exists()` method ensures the directory exists before attempting deletion. The `.delete()` method deletes the directory only if it is empty. If successful, it prints "Directory deleted successfully: Projects/Java". If it fails (for example, because it contains files/subdirectories), it prints "Failed to delete directory. It may not be empty.". If the directory is missing, it prints: "Directory does not exist: Projects/Java".	
	```
	public class ListDirectoryContents {
	    public static void main(String[] args) {
	        String directoryPath = "Projects/Java";
	        File directory = new File(directoryPath);

	        // List all files and directories in the specified directory
	        String[] contents = directory.list();

	        if (contents != null) {
	            System.out.println("Contents of " + directoryPath + ":");
	            for (String fileName : contents) {
	                System.out.println(fileName);
	            }
	        } else {
	            System.out.println("The directory is empty or does not exist.");
	        }
	```

Close curly braces to end the `ListDirectoryContents` class definition.	
	```
	    }
	}
	```

Description	Example

Explanation: This Java program uses the `File.delete()` method to delete a specified directory if it exists. The program handles success and failure cases gracefully.

Creating a directory with NIO

Description	Example

Import Java class `java.nio.file.Files` for file and directory operations, `java.nio.file.Path` to represent file and directory paths in a platform-independent way, `java.nio.file.Paths` to create `Path` instances, and `java.io.IOException` to handle potentila I/O errors.	
	```
	import java.nio.file.Files;
	import java.nio.file.Path;
	import java.nio.file.Paths;
	import java.io.IOException;
	```

Define a class `CreateDirectory` that contains the Java `main` method. The method `Paths.get("Projects/NioExample")` creates a `Path` object representing the directory to be created. The `try` block uses `Files.createDirectories()` instead of `File.mkdirs()` to creates all necessary parent directories if they don't exist. It does not throw an error if the directory already exists and stores the created directory path in `createdDir`. The program prints "Directory created successfully: Projects/NioExample" if it is successful. The `catch` block catches `IOException` if directory creation fails (for example, insufficient permissions) and prints an error message: "Failed to create directory: <error_message>".	
	```
	public class CreateDirectory {
	    public static void main(String[] args) {
	        // Define the directory path
	        String directoryPath = "Projects/Java";

	        // Create a File object
	        File directory = new File(directoryPath);

	        // Create the directory
	        if (!directory.exists()) {
	            boolean created = directory.mkdirs(); // Use mkdirs() to create nested directories
	            if (created) {
	                System.out.println("Directory created successfully: " + directoryPath);
	            } else {
	                System.out.println("Failed to create directory.");
	            }
	        } else {
	            System.out.println("Directory already exists: " + directoryPath);
	        }
	```

Close curly braces to end the `CreateDirectory` class definition.	
	```
	    }
	}
	```

Explanation: This Java program creates a directory using Java NIO (New Input/Output) instead of the traditional File class. It handles success and failure cases gracefully and works cross-platform.

Real World example of Document Management System

Description	Example

Import Java class `java.nio.file.Files` for file and directory operations, `java.nio.file.Path` to represent file and directory paths in a platform-independent way, `java.nio.file.Paths` to create `Path` instances, `java.io.IOException` to handle potentila I/O errors, and `java.util.Scanner` for handling user input.	
	```
	import java.nio.file.Files;
	import java.nio.file.Path;
	import java.nio.file.Paths;
	import java.io.IOException;
	import java.util.Scanner;
	```

Define a class `DocumentManagementSystem` that contains the Java `main` method. All directory	
	```
	public class DocumentManagementSystem {
	    private static final String BASE_DIRECTORY = "Documents";
	```

Description	**Example**

operations will occur within the "Documents" folder defined by the String `BASE_DIRECTORY`. The `main` method continuously prompts the user to choose an option and calls the corresponding method based on user input: "1" creates a new directory inside "Documents", "2" lists contents of a specified directory, "3" deletes a specified directory, and "4" exits the program.	`    public static void main(String[] args) {`
	`        Scanner scanner = new Scanner(System.in);`
	`        String command;`
	
	`        while (true) {`
	`            System.out.println("1. Create directory\n2. List documents\n3. Delete directory\n4. Exit");`
	`            command = scanner.nextLine();`
	
	`            switch (command) {`
	`                case "1": createDirectory(scanner); break;`
	`                case "2": listDirectory(scanner); break;`
	`                case "3": deleteDirectory(scanner); break;`
	`                case "4": scanner.close(); return;`
	`                default: System.out.println("Invalid choice.");`
	`            }`
	`        }`
	`    }`

The `createDirectory()` method creates a new directory and reads directory name from user input. It uses `Files.createDirectories(path)` to create the directory (including missing parent directories). If successful, it prints "Created: path". If an error occurs, it prints "Error: message".	`    private static void createDirectory(Scanner scanner) {`
	`        System.out.print("New directory name: ");`
	`        Path path = Paths.get(BASE_DIRECTORY, scanner.nextLine());`
	`        try {`
	`             System.out.println("Created: " + Files.createDirectories(path));`
	`         } catch (IOException e) {`
	`             System.err.println("Error: " + e.getMessage());`
	`         }`
	`    }`

The `listDirectory()` method lists the contents of a directory and reads directory name from user input. It uses `Files.list(path)` to retrieve the directory and prints each file/subdirectory. If the directory doesn't exist or an error occurs, it prints "Error: message".	`    private static void listDirectory(Scanner scanner) {`
	`        System.out.print("Directory to list: ");`
	`        Path path = Paths.get(BASE_DIRECTORY, scanner.nextLine());`
	`        try {`
	
	`             Files.list(path).forEach(System.out::println);`
	`         } catch (IOException e) {`
	`             System.err.println("Error: " + e.getMessage());`
	`         }`
	`    }`

The `deleteDirectory()` method deletes a directory and reads directory name from user input. It uses `Files.delete(path)` to delete the specified directory. If successful, it prints "Deleted: path". The `Files.delete(path)` will fail if the directory is not empty. It only works on empty directories.	`    private static void deleteDirectory(Scanner scanner) {`
	`        System.out.print("Directory to delete: ");`
	`        Path path = Paths.get(BASE_DIRECTORY, scanner.nextLine());`
	`        try {`
	`             Files.delete(path);`
	`             System.out.println("Deleted: " + path);`
	`        } catch (IOException e) {`
	`             System.err.println("Error: " + e.getMessage());`
	`         }`

Close curly braces to end the `main` class definition.	`    }`
	`}`

Explanation: This Java program provides a simple command-line interface for managing directories inside a "Documents" folder. It allows users to create, list, and delete directories using Java NIO (New Input/Output).

Author(s)

Ramanujam Srinivasan
Lavanya Thiruvali Sunderarajan

Skills Network