Exception Handling and Text I/O

Introduction to Robust Programming

- Programs encounter errors:
  - Syntax Errors: Compiler catches these.
  - Logic Errors: Program runs, but output is incorrect.
  - Runtime Errors (Exceptions): Program terminates abnormally (e.g., trying to open a non-existent file).
- Exception Handling: A structured way to deal with runtime errors, preventing crashes and allowing graceful recovery.

What is an Exception?

- An event that occurs during the execution of a program that disrupts the normal flow of instructions.
- An object (instance of an Exception class) that is "thrown" by an abnormal event and can be "caught" by an exception handler.
- Java's mechanism to manage "exceptional" conditions.

Exception Hierarchy (The Throwable Class)

- All **exceptions** and **errors** in Java are subclasses of java.lang.Throwable.
- Error: Represents serious problems that applications should not try to catch (e.g., OutOfMemoryError, StackOverflowError). Indicate severe, often unrecoverable system problems.
- Exception: Represents conditions that applications should catch and handle (e.g., IOException, SQLException).
- RuntimeException (Unchecked Exceptions): Subclasses of RuntimeException. Not required to be handled or declared (e.g., NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException).
- Checked Exceptions: All other Exception subclasses. Must be caught or declared.

Checked vs. Unchecked Exceptions

- Checked Exceptions:
  - Compiler Enforced: Must be handled or declared in the method signature.
  - Occur in situations where the program can reasonably recover.
  - Examples: IOException, FileNotFoundException, SQLException, ClassNotFoundException.
- Unchecked Exceptions (RuntimeException):
  - No Compiler Enforcement: The compiler doesn't force handling.
  - Usually indicate programming logic errors (bugs).
  - Examples: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, IllegalArgumentException.

Handling Exceptions: The try-catch Block

- The core construct for exception handling.
- try block: Contains the code that might throw an exception.
- catch block(s): Contains code to handle a specific type of exception if thrown in the try block.

try-catch Syntax and Flow

```
try {
    // Code that might cause an exception
    System.out.println("Executing code inside try block.");
    int result = 10 / 0; // This line throws ArithmeticException
    System.out.println("Result: " + result); // This line will NOT be reached
} catch (ArithmeticException e) {
    // This block executes if an ArithmeticException is thrown
    System.out.println("Caught an ArithmeticException!");
    System.out.println("Error message: " + e.getMessage()); // / by zero
    e.printStackTrace(); // Prints the stack trace
}
System.out.println("Program continues after the try-catch block.");
```

Explanation: If an exception occurs in try, the rest of the try block is skipped, and control jumps to the appropriate catch block.

Multiple catch Blocks

- A try block can have multiple catch blocks to handle different types of exceptions.
- Order matters: Catch more specific exceptions first, then more general ones. (If Exception is first, it will catch everything, making subsequent specific catches unreachable).

Example: Multiple catch Blocks

```
import java.util.InputMismatchException; // For invalid input
import java.util.Scanner;

public class MultipleCatchExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Enter an integer: ");
            int num = scanner.nextInt(); // Potential InputMismatchException
            System.out.print("Enter a divisor: ");
            int divisor = scanner.nextInt();
            int result = num / divisor; // Potential ArithmeticException
            String s = null;
            System.out.println(s.length()); // Potential NullPointerException
            System.out.println("Result: " + result);
        } catch (InputMismatchException e) {
            System.out.println("Input Error: Please enter a valid integer.");
            scanner.next(); // Consume the invalid input
        } catch (ArithmeticException e) {
            System.out.println("Calculation Error: Cannot divide by zero.");
        } catch (NullPointerException e) {
            System.out.println("Programming Error: Tried to use a null reference.");
        } catch (Exception e) { // Catches any other exception (general catch-all)
            System.out.println("An unexpected error occurred: " + e.getMessage());
        } finally {
            scanner.close(); // Important: Close the scanner in finally
        }
        System.out.println("Program finished.");
    }
}
```

The finally Block

- An optional block that always executes, regardless of whether an exception occurred, was caught, or if a return statement was executed.

- Purpose: For cleanup code (e.g., closing files, database connections, releasing network resources).

Example: finally Block

```
try {
    System.out.println("Inside try block. Value: " + value);
    if (value < 0) {
        throw new IllegalArgumentException("Value cannot be negative.");
    }
    if (value == 0) {
        return; // Exits the method
    }
    System.out.println("Processing value: " + (100 / value));
} catch (IllegalArgumentException e) {
    System.out.println("Caught exception: " + e.getMessage());
} finally {
    System.out.println("Finally block always executes.");
}
```

Declaring Exceptions: The throws Keyword

- If a method can throw a checked exception but chooses not to handle it internally, it must declare this using the throws keyword in its signature.

- This is a "contract" that tells calling methods: "Be aware, I might throw this exception, and you need to handle it or declare it too."

throws Syntax and Example

```
import java.io.FileNotFoundException;
import java.io.File;
import java.util.Scanner;
public class ThrowsExample {
    // This method declares that it MIGHT throw FileNotFoundException
    public static void processFile(String fileName) throws FileNotFoundException {
        File file = new File(fileName);
        Scanner fileScanner = new Scanner(file); // This can throw FileNotFoundException
        System.out.println("Successfully opened " + fileName);
        // ... process file content ...
        fileScanner.close();
    }
public static void main(String[] args) {
        try {
            processFile("data.txt"); // Caller must handle or declare
            processFile("nonExistent.txt"); // This will throw FileNotFoundException
        } catch (FileNotFoundException e) {
            System.out.println("Error: File not found - " + e.getMessage());
        } catch (Exception e) { // General catch for other potential issues
            System.out.println("An unexpected error occurred: " + e.getMessage());
        }
    }
}
```

Throwing Exceptions: The throw Keyword

- Used to explicitly create and throw an exception object.
- Can throw built-in exception types or custom exception types.
- Syntax:

  throw new ExceptionType("Error message");

Example: Using throw

```
public class GradeCalculator {
    public static char calculateGrade(int score) {
        if (score < 0 || score > 100) {
            // Explicitly throwing an unchecked exception
            throw new IllegalArgumentException("Score must be between 0 and 100. Got: " + score);
        }
        if (score >= 90) return 'A';
        else if (score >= 80) return 'B';
        else return 'F';
    }

    public static void main(String[] args) {
        try {
            System.out.println("Score 95: Grade " + calculateGrade(95));
            System.out.println("Score 70: Grade " + calculateGrade(70));
            System.out.println("Score -5: Grade " + calculateGrade(-5)); // Throws exception here
            System.out.println("Score 105: Grade " + calculateGrade(105)); // This line won't be reached
        } catch (IllegalArgumentException e) {
            System.out.println("Error calculating grade: " + e.getMessage());
        }
        System.out.println("Grade calculation complete.");
    }
}
```

Creating Custom Exception Classes

- Define your own specific exception types for your application's unique error conditions.
- Extend Exception for checked exceptions.
- Extend RuntimeException for unchecked exceptions.
- Provide meaningful constructors (typically with a message).

Example: Custom Checked Exception

```
// Custom Checked Exception: Requires caller to handle or declare
public class InsufficientFundsException extends Exception {
    private double currentBalance;
    private double withdrawalAmount;

    public InsufficientFundsException(String message, double balance, double amount) {
        super(message);
        this.currentBalance = balance;
        this.withdrawalAmount = amount;
    }

    public double getCurrentBalance() { return currentBalance; }
    public double getWithdrawalAmount() { return withdrawalAmount; }
}
```

Example: Custom Checked Exception

```
public class BankAccount {
    private double balance;
    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }
    public void withdraw(double amount) throws InsufficientFundsException { // Declares the exception
        if (amount <= 0) {
            throw new IllegalArgumentException("Withdrawal amount must be positive."); // Unchecked
        }
        if (amount > balance) {
            // Throws our custom checked exception
            throw new InsufficientFundsException("Insufficient funds for withdrawal.", balance, amount);
        }
        balance -= amount;
        System.out.printf("Withdrew %.2f. New balance: %.2f%n", amount, balance);
    }
    public double getBalance() { return balance; }
```

Example: Custom Checked Exception

```
public static void main(String[] args) {
        BankAccount account = new BankAccount(500.0);
        try {
            account.withdraw(200.0);
            account.withdraw(400.0); // This will throw InsufficientFundsException
            System.out.println("This line won't be printed after exception.");
        } catch (InsufficientFundsException e) {
            System.err.println("Caught InsufficientFundsException:");
            System.err.println("  " + e.getMessage());
            System.err.printf("  Current Balance: %.2f, Attempted Withdrawal: %.2f%n",
                              e.getCurrentBalance(), e.getWithdrawalAmount());
        } catch (IllegalArgumentException e) { // Catch the unchecked one too
            System.err.println("Caught IllegalArgumentException: " + e.getMessage());
        }
        System.out.printf("Final balance: %.2f%n", account.getBalance());
    }
}
```

try-with-resources Statement (Java 7+)

- Simplifies resource management by automatically closing resources that implement the AutoCloseable interface.
- Eliminates the need for explicit finally blocks for resource cleanup.

try-with-resources Syntax and Example

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
public class TryWithResourcesExample {
    public static void main(String[] args) {
        // Resources declared here will be automatically closed
        try (PrintWriter writer = new PrintWriter(new FileWriter("output.txt"));
             FileWriter fw = new FileWriter("another.txt")) { // Can declare multiple resources
            writer.println("Hello, this is line 1.");
            writer.println("This is line 2.");
            fw.write("Another file's content.");
            System.out.println("Data written to files successfully.");
        } catch (IOException e) {
            System.out.println("An I/O error occurred: " + e.getMessage());
        }
        // writer and fw are automatically closed here, even if an exception occurs
    }
}
```

Text I/O: Reading and Writing Files

- Programs often need to interact with persistent data stored in files.
- Input/Output (I/O): The process of transferring data between a program and an external source (like a file).
- Java provides classes in the java.io package for file I/O.
- For text files, Scanner (for reading) and PrintWriter (for writing) are commonly used.

Writing Text to a File: PrintWriter

- PrintWriter is a convenient class for writing formatted text to a file.
- It wraps other Writer objects and handles character encoding.
- Key steps:
  1. Create a File object (optional, can use filename directly).
  2. Create a PrintWriter object. This constructor can throw FileNotFoundException (a checked exception if a file cannot be opened for writing).
  3. Use print(), println(), printf() methods (similar to System.out).
  4. Crucial: Close the PrintWriter to flush buffered data and release file resources. (Best done with try-with-resources).

Example: Writing to a File with PrintWriter

```
 String filename = "names.txt";
// Using try-with-resources to ensure PrintWriter is closed
try (PrintWriter output = new PrintWriter(filename)) {
    System.out.println("Writing names to " + filename + "...");
    output.println("Alice Wonderland 25");
    output.println("Bob The Builder 30");
    output.printf("Charlie Brown %d%n", 10); // %n for platform-independent newline
    output.println("Diana Prince 35");
    System.out.println("Names written successfully.");
} catch (IOException ex) { // Catching IOException as PrintWriter constructor can throw it
    System.err.println("Error writing to file " + filename + ": " + ex.getMessage());
    ex.printStackTrace();
}
```

Reading Text from a File: Scanner

- The Scanner class (from java.util) can parse primitive types and strings from a file.
- Key steps:
  1. Create a File object for the file to be read.
  2. Create a Scanner object, passing the File object. This constructor can throw FileNotFoundException.
  3. Use hasNext(), next(), nextInt(), nextDouble(), nextLine(), etc., to read data.
  4. Crucial: Close the Scanner when done. (Best done with try-with-resources).

Example: Reading from a File with Scanner

```
String filename = "names.txt"; // Assume this file was created by previous example
File file = new File(filename);
// Using try-with-resources to ensure Scanner is closed
try (Scanner input = new Scanner(file)) {
    System.out.println("Reading names from " + filename + ":");
    while (input.hasNext()) { // Loop while there's more data
        String firstName = input.next();
        String lastName = input.next();
        int age = input.nextInt();
        System.out.printf("Name: %s %s, Age: %d%n", firstName, lastName, age);
    }
    System.out.println("Finished reading file.");
} catch (FileNotFoundException ex) {
    System.err.println("Error: File not found - " + filename);
} catch (Exception ex) { // Catch other potential issues, e.g., InputMismatchException
    System.err.println("An error occurred while reading: " + ex.getMessage());
    ex.printStackTrace();
}
```

The File Class

- The java.io.File class represents a file or directory path within the file system.
- It does not provide methods for reading/writing data.
- Methods for File Information/Manipulation:
  - exists(): Checks if the file/directory exists.
  - getAbsolutePath(): Returns the absolute path.
  - getName(): Returns the simple name of the file/directory.
  - length(): Returns the size of the file in bytes.
  - canRead(), canWrite(), canExecute(): Check permissions.
  - isDirectory(), isFile(): Check type.
  - mkdir(), mkdirs(): Create directories.
  - delete(): Deletes the file or empty directory.
  - renameTo(File newName): Renames/moves a file.

```
import java.io.File;
import java.io.IOException;

public class FileClassExample {
    public static void main(String[] args) throws IOException {
        // Create a File object for a non-existent file for demonstration
        File file = new File("testdir/mydata.txt");
        System.out.println("File path: " + file.getPath());
        System.out.println("Absolute path: " + file.getAbsolutePath());
        System.out.println("File exists: " + file.exists()); // false initially

        // Create a directory
        File dir = new File("testdir");
        if (!dir.exists()) {
            dir.mkdir(); // or mkdirs() for nested directories
            System.out.println("Directory 'testdir' created: " + dir.exists());
        }
```

```
// Create a new file within the directory
        if (file.createNewFile()) { // This method can throw IOException
            System.out.println("File 'mydata.txt' created: " + file.exists()); // true now
        }

        System.out.println("File name: " + file.getName());
        System.out.println("Is directory: " + file.isDirectory());
        System.out.println("Is file: " + file.isFile());
        System.out.println("File size (bytes): " + file.length()); // 0 after creation

        // Clean up: delete the file and then the directory
        if (file.delete()) {
            System.out.println("File 'mydata.txt' deleted.");
        }
        if (dir.delete()) { // Directory must be empty to be deleted by delete()
            System.out.println("Directory 'testdir' deleted.");
        }
    }
}
```

Relative vs. Absolute Paths

- Absolute Path: The full path from the root directory of the file system (e.g., C:\Users\John\document.txt on Windows, /home/john/document.txt on Linux/macOS).
- Relative Path: A path defined relative to the current working directory of the Java program.
- If you just specify "myFile.txt", Java looks in the directory where the program is executed.
- It's good practice to specify paths carefully, especially in production applications.

Common I/O Exceptions

- FileNotFoundException: A checked exception thrown by FileReader, Scanner, FileInputStream, FileOutputStream, PrintWriter constructors when a file cannot be opened (e.g., doesn't exist for reading, or access denied for writing).
- IOException: A general checked exception, parent of FileNotFoundException, covers most I/O errors (e.g., disk full, network connection lost).
- InputMismatchException: An unchecked exception thrown by Scanner methods (like nextInt(), nextDouble()) if the input does not match the expected type.

Best Practices for Exception Handling

- Handle Specific Exceptions First: Catch specific exception types before more general ones.
- Don't "Swallow" Exceptions: Don't catch an exception and do nothing. At a minimum, log the error or print a message.
- Use try-with-resources for AutoCloseable: Ensures resources are always closed, even if exceptions occur.
- Throw Early, Catch Late: Detect errors as early as possible. Handle them at a higher level where they can be meaningfully addressed or reported.
- Provide Informative Messages: Custom exception messages and e.getMessage() or e.printStackTrace() help in debugging.
- Document Exceptions: Use Javadoc @throws tag to document exceptions a method might throw.

Introduction to Reading Data from the Web

- Beyond local files: Programs often need to access data from remote sources.
- The Internet is a vast source of information (text, HTML, JSON, XML).
- Java provides classes to establish network connections and read data from URLs.
- This process involves network I/O, which inherently means **checked exceptions** are common.

Key Classes for Web Data Reading

- java.net.URL: Represents a Uniform Resource Locator (a web address). Used to create a URL object from a string.
- java.net.URLConnection: An abstract class representing a communication link between the URL and the application. You usually work with its concrete subclass, HttpURLConnection for HTTP/HTTPS.
- java.io.InputStream / java.io.Reader: Used to read data from the connected URL.
- java.util.Scanner: A convenient tool to read and parse data from the InputStream obtained from the URL.

Steps to Read Data from a URL

- Create a URL object: URL url = new URL("http://example.com/data.txt");
- Open a connection: url.openStream() returns an InputStream. For more control (e.g., setting headers), use url.openConnection() which returns a URLConnection.
- Wrap the stream in a Scanner: Scanner input = new Scanner(url.openStream()); (This is the simpler way for text data).
- Read data: Use Scanner methods (hasNextLine(), nextLine(), next(), etc.).
- Close the Scanner: Crucial for resource management (best with try-with-resources).

Example: Reading Text from a Web Page (Simple)

```
import java.io.IOException;
import java.net.URL;
import java.util.Scanner;
public class ReadWebPageSimple {
    public static void main(String[] args) {
        String urlString = "https://www.gutenberg.org/files/2701/2701-0.txt"; // Moby Dick short text
        System.out.println("Attempting to read from URL: " + urlString);
        try (Scanner input = new Scanner(new URL(urlString).openStream())) {
            // Read and print the first 10 lines
            int count = 0;
            while (input.hasNextLine() && count < 10) {
                System.out.println(input.nextLine());
                count++;
            }
            System.out.println("\nSuccessfully read first 10 lines from the web.");
        } catch (IOException e) { // Catches MalformedURLException and other I/O errors
            System.err.println("Error reading from web: " + e.getMessage());
            e.printStackTrace();
        }   } }
```

Expected output (first 10 lines of Moby Dick text file):
The Project Gutenberg eBook of Moby Dick; or The Whale, by Herman Melville

Handling MalformedURLException

- Occurs if the string passed to the URL constructor is not a valid URL format.
- This is a checked exception, so it must be caught or declared.
- Example :

```
import java.net.MalformedURLException;
import java.net.URL;

public class MalformedURLExample {
    public static void main(String[] args) {
        String badUrl = "not-a-valid-url";
        try {
            URL url = new URL(badUrl);
            System.out.println("URL created successfully: " + url);
        } catch (MalformedURLException e) {
            System.err.println("Error: Invalid URL format provided!");
            System.err.println("Message: " + e.getMessage());
        }
        System.out.println("Program finished.");
    }
}
```

Handling Network I/O Exceptions (IOException)

- Network operations are prone to various issues:
  - Server unavailable/down.
  - No internet connection.
  - Timeout.
  - Invalid permissions or access issues.
- These are generally caught by IOException (or its subclasses).

Example (implicitly handled in previous slide, but showing explicit catch):

```
String urlString = "http://this-domain-does-not-exist-123xyz.com/data.txt";


    try (Scanner input = new Scanner(new URL(urlString).openStream())) {
        while (input.hasNextLine()) {
            System.out.println(input.nextLine());
        }
    } catch (MalformedURLException e) {
        System.err.println("Caught MalformedURLException: " + e.getMessage());
    } catch (UnknownHostException e) { // Specific for host not found
        System.err.println("Caught UnknownHostException: Cannot resolve host '" + e.getMessage() + "'");
    } catch (IOException e) { // General catch for other I/O errors
        System.err.println("Caught general IOException: " + e.getMessage());
        e.printStackTrace();
    }
```

Finding a specific string in an HTML page

```
String urlString = "https://www.cs.armstrong.edu/liang/data/Welcome.html";
    String searchString = "Welcome to Liang's Website!";
    System.out.println("Searching for '" + searchString + "' on: " + urlString);
    boolean found = false;
    try (Scanner input = new Scanner(new URL(urlString).openStream())) {
        while (input.hasNextLine()) {
            String line = input.nextLine();
            if (line.contains(searchString)) {
                found = true;
                System.out.println("Found the string! Line: " + line.trim());
                break;
            }}
        if (!found) {                System.out.println("String '" + searchString + "' not found on the page.");            }
    } catch (IOException e) {
        System.err.println("Error reading HTML content: " + e.getMessage());
        e.printStackTrace();
    }
```

Best Practices for Web Data Reading

- Handle Exceptions: Always wrap web I/O in try-catch blocks, as network operations are inherently unreliable. Be specific (e.g., MalformedURLException, UnknownHostException).
- Use try-with-resources: For automatic closing of InputStream and Scanner objects.
- Respect Server Policies: Don't hammer servers with requests. Implement delays if necessary. Check robots.txt.
- Parse Carefully: Web data is often unstructured or semi-structured. For complex parsing (HTML, JSON, XML), consider dedicated libraries (e.g., Jsoup for HTML, Jackson/Gson for JSON).
- Network Permissions: For non-local network access, your Java application might require specific security permissions (less common in modern standalone apps, but relevant for applets or highly restricted environments).

Conclusion

- Exception handling makes your Java applications robust and resilient to runtime errors.
- try-catch-finally and try-with-resources are essential constructs for managing exceptions and resources.
- Text I/O (Scanner, PrintWriter, File) enables your programs to interact with external data, crucial for many real-world applications.
- Mastering these concepts is fundamental for building reliable and functional software.