Exception & Error Handling

Program Correctness

- **Should be every programmer's first and major goal**
  - Second goal: Readability
  - Third goal: Maintainability
  - Fourth goal: Efficiency
- **Type of Errors:**
  - Syntax Errors:
    - Written code does not represent a valid program in the chosen language.
    - Detected at compile time.
    - E.g. Missing semi-colons, etc.
  - Semantic Errors (i.e. Bugs):
    - The program doesn't do what it is supposed to do.
    - Mostly detected at runtime.
    - Sometimes, the compiler can detect some semantic errors (e.g. uninitialized variable).
    - E.g. Program crashes when provided with a certain input or gives wrong result.

Early Error Handling Strategies

- **Strategy 1: Global Error Code:**
  - Add a well-documented global variable.
  - Change the variable value when something goes wrong.
  - Define and use a system of error codes to indicate what error occurred.
  - The programmer is responsible for checking the value when appropriate.

- Example:

```
public class Main {
    public static int error_code; // Define a global erro

    public static void do_something(int a) {
        error_code = 0;

        // Set error_code
        if (a < 0) {
            error_code = -1;
        } else if (a % 2 == 0) {
            error_code = -2;
      } else {
          System.out.println("Did something");
      }
  }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        while (s.hasNextInt()) {
            do_someting(s.nextInt());
            // Check the value of the error code and handl
            if (error_code != 0) {
                if (error_code == -1) {
                    System.out.println("Negative number er
                } else if (error_code == -2) {
                    System.out.println("Even number error"
                }
            }
            s.close();
        }
    }
}
```

- **Strategy 2: Special Return Value:**
  - Designate special return values meant to be interpreted as errors.
  - Procedure conventions:
    - 0 indicates success.
    - Less than 0 indicates an error.
    - Different negative values indicate different errors.

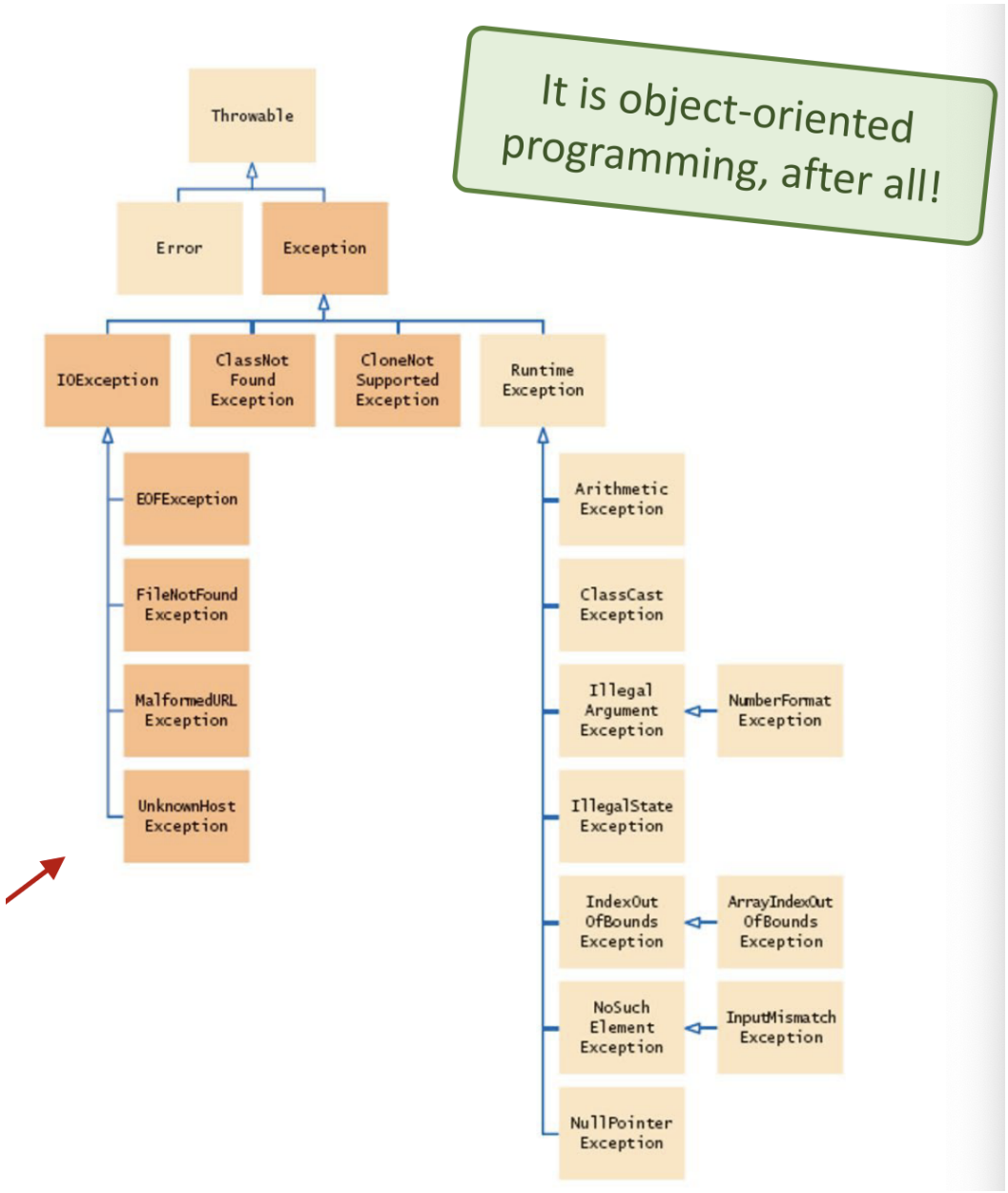
```
public class Main {
    public static void main(String[] args) {
            String haystack = "This is my haystack";
            Scanner s = new Scanner(System.in);
            while (s.hasNext()) {
                String needle = s.next();
                // indexOf returns the index of search val
                // Else, return -1
                int position = haystack.indexOf(needle);
                if (position == -1) {
                System.out.println("Error: could not find
                } else {
                System.out.println("Found needle at " + po
            }
        }
    s.close();
}
```

- **Drawbacks of Early Strategies:**
  - They are inconsistent & convention-based.
  - Methods must have an out-of-possible-range return values to use for indicating that an error has occurred.
  - Relies on documentation to explain what each value means.
  - The programmer must remember to check for errors.

- Difficult to extend in future development.

Java Exceptions - Modern Error Handling

- Java has a sophisticated error reporting mechanism called **exceptions**.
  - **When a runtime error occurs, an exception is thrown.**
  - Method can catch and handle the exception itself or throw it to the caller method (default behavior).
- **Hierarchy of Exceptions in Java:**
  - Parent class of all Exceptions: Throwable.
  - Two subclasses of Exceptions: `Error` (reserved for JVM errors) & `Exceptions`.
  - Several subclasses of `Exception`, including `RuntimeException`, `IOException`, `InterruptedException`, etc.
  - Majority of exceptions are subclasses of `RuntimeException`, including `ArrayIndexOutOfBoundsException`, `IllegalArgumentException`, `NumberFormatException`.
  - Several exceptions are also subclasses of `IOException`, including: `EOFException`, `FileNotFoundException`, `MalformedURLException`, `MalformedURLException`.
  - Useful methods that exceptions inherit from `Throwable` include:
    - `String getMessage()`: Returns textual description.
    - `void printStackTrace(PrintStream s)`: Prints list of method calls.
    - We can use: `e.getMessage()` or `e.printStackTrace(System.err)` to get more info about an exception and why it occurred.



- **Exception Handling:**
  - A formal method for detecting, signaling, and responding to errors.
  - Built into the programming language, such as Java, C++, C#, Python, JS, etc.
  - Includes 2 Primary Parts:
    - **Throwing** (Detecting) an Exception.
    - **Catching** (Handling) an Exception.

- Benefits of Exceptions:
  - They are consistent, extensible, and modular
  - They are expressive, i.e. can express exactly what type of error occurred, while also encapsulating the details of the error.
  - They are dependable, and have obvious behavior.
  - They are safe, as they can allow critical code to execute even if an error occurs.

Throwing an Exception

- **Throwing an Exception:**
  - In Java: Use the `throw` keyword to indicate that an error occurred.
  - After `throw`, specify an `Exception` object. A **subclass type** of `Exception` can inform the caller about what happened.
    - The `Exception` object will be sent up to the caller, so it must be descriptive (i.e. describe as close as possible) about what error occurred.
  - As soon as an exception is thrown, the code block (or program) will immediately stop and the rest will not be executed.
  - Example:

```
private void setAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("Error descript
    }
    this.age = age; // Won't be executed if exception is t
}
```

Catching an Exception

- **Catching an Exception:**
  - In Java: Use the `try/catch` block to catch an exception as it occurs.

    - Execution within the `try` block stops as soon as an exception occurs.
- `catch` blocks are similar to `if/else` statements:
    - They check from top to bottom, one block at a time, for a match.
    - As soon as a match (matched exception) is found, the block is executed, and no other blocks are searched/executed.
- Can also catch exceptions of class `Exception` or any of its subclass types, e.g. `RuntimeException` or `IOException` or `ArithmeticException`.
- Syntax:

```
try {
    // Execution within try block stops as soon as an exce
    methodA();
    methodB();
    methodC();
} catch (ExceptionType e) {
    // Code to respond to ExceptionType
} catch (OtherExceptionType oe) {
    // Code to respond to OtherEcceptionType
}
```

⇒ **Explanation**:

- As soon as an exception is caught in `methodA()` , jumps right to the appropriate `catch` block, leaving `methodB()` and `methodC()` unexecuted.
- Meaning that `methodC()` will only be executed if `methodA()` and `methodB()` do not throw any exception.

- We can also combine `catch` blocks into a single one (useful if we handle them similarly):
    - Example:

```
try {
    // try something
} catch (ExceptionType1| Exception2 e) {
```

```
    // do something
}
```

- **The `finally` Block:**
  - The `finally` block is part of the Java exception handling mechanism. It is used in conjunction with the `try` and `catch` blocks.
  - The code inside the `finally` block will always execute after the `try` block, whether or not an exception is thrown, and regardless of whether it was caught in a `catch` block.
  - The `finally` block goes after the `catch` statements. Code in the `finally` block executes last, after the entire `try/catch` sequence finishes.
  - **Key Characteristics of the `finally` Block:**
    1. **Always Executes**: The `finally` block will execute after the `try` block, even if:
       - An exception is thrown and not caught.
       - The `catch` block is executed.
       - The `try` block contains a `return` statement.
       - The JVM exits during the execution of the `try` or `catch` blocks (except in specific cases, such as a call to `System.exit()`).
    2. **Resource Cleanup**: The primary use case for the `finally` block is to perform resource cleanup operations (e.g., closing files, releasing database connections) that must happen regardless of success or failure.
    3. **Multiple `finally` Blocks**: You cannot have more than one `finally` block associated with a single `try` block.
    4. **Combining with `try` and `catch`**: The `finally` block can be used with or without a `catch` block, but it's common to see it in conjunction with both.
  - Example:

```java
public static void main(String[] args) {
    try {
        System.out.println("A");      // Step 1
        method1();                    // Step 2
        System.out.println("B");      // Step 3
    } catch (NullPointerException e) {
        System.out.println("C");      // Step 4
    } catch (RuntimeException e) {
        System.out.println("D");      // Step 5
    } finally {
        System.out.println("E");      // Step 6
    }
    System.out.println("F");          // Step 7
}

public static void method1() {
    throw new RuntimeException();      // Step 8
}

// Output:
// A
// D
// E
// F
```

- **Code in `finally` Block vs. Code after `try-catch-finally` block:**

Code in `finally` block	**Code after `try-catch-finally` block**
Always executes, regardless of whether an exception occurs or is caught.	Only executes if no uncaught exceptions occur.
Typically used for cleanup operations (e.g., closing resources).	Used for continuing the normal flow of execution after error handling.
Even if the `try` or `catch` block has a `return` statement, the `finally` block will execute before returning.	If an uncaught exception occurs, the code after `finally` will not be executed.

Executes just before the method completes or returns, even in the presence of exceptions.	Executes only when the exception handling (if any) is completed successfully.

- Summary:
  - `finally` **block**: Executes always, regardless of whether an exception occurs or not.
  - **Code after** `finally` **block**: Executes only if no uncaught exceptions are handled in the `try` or `catch` blocks, and it's used for normal program flow after exception handling.

When to Throw/Catch an Exception

- Throwable Handling: We can either catch or re-throw an exception within a `try/catch` block.
- **Throwing Exceptions:**
  - You throw exceptions when an error condition occurs that the method **cannot or should not handle itself**. It signals to the caller of the method that something went wrong.
  - **When to Throw an Exception:**
    - **Invalid Arguments**: If a method receives parameters that are not acceptable or out of bounds, throwing an exception is appropriate (e.g., `IllegalArgumentException` or `NullPointerException`).
    - **Illegal States**: When an object is not in a valid state to perform an operation, throw an exception (e.g., `IllegalStateException`).
    - **Resource Issues**: When a resource like a file or network connection is unavailable, or there's a problem accessing it (e.g., `FileNotFoundException`, `IOException`).
    - **Unrecoverable Errors**: If a method cannot complete its task because of an unrecoverable issue, throwing an exception allows the caller to handle it or fail gracefully.
  - **Guidelines for Throwing:**

- **Use specific exceptions**: Throw specific exceptions rather than general ones like `Exception` or `Throwable`, which can make debugging harder.
- **Document using `throws` clause**: For checked exceptions, declare them in the method signature so that callers know what exceptions to expect.
- Example:

```
public void readFile(String filename) throws FileNotFoundException {
    // Code to read the file
}
```

  - This tells the compiler to force the caller method to catch/handle the checked exception.

- **Catching Exceptions:**
  - Catching exceptions involves handling errors within a `try/catch` block to prevent the program from crashing and to potentially recover from the issue. However, you should only catch exceptions when your code can **recover from the error** or take some meaningful action.
  - **When to Catch an Exception:**
    - **Recoverable Errors**: If the code can fix the issue or provide an alternative path (e.g., retry logic for network failures), catching the exception makes sense.
      - Example: Retry mechanism on a failed network connection.

```
try {
    connectToServer();
} catch (IOException e) {
    retryConnection();
}
```

- **Graceful Degradation**: If the program can't recover but should handle the error gracefully (e.g., logging the error, displaying a friendly message, or falling back to a default behavior).
    - Example: If a file is missing, provide a default configuration:

```
try {
    loadConfiguration("config.txt");
} catch (FileNotFoundException e) {
    loadDefaultConfiguration();
}
```

- **Handling Known Exceptional Scenarios**: If your code expects specific exceptions under certain conditions (e.g., `ArrayIndexOutOfBoundsException` during an array operation), you can catch and handle them appropriately.

- **Guidelines for Catching:**
    - **Catch only what you can handle**: Do not catch exceptions just to suppress them. Catching and ignoring exceptions can lead to hidden bugs and unhandled states.
    - **Avoid catching `Throwable`**: `Throwable` includes both `Exception` and `Error`. Errors are generally unrecoverable (like `OutOfMemoryError`), so avoid catching `Throwable` unless you're implementing a global error handler for logging.
    - **Catch the most specific exception**: Avoid catching general exceptions like `Exception` or `RuntimeException` unless it's at a high level (e.g., in a centralized error handler). Catch specific exceptions to handle particular cases.
    - Bad practice:

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

```
        e.printStackTrace();
    }
```

- **Example of Catching and Throwing Properly:**

```
public void processFile(String filename) {
    try {
        readFile(filename); // This method may throw FileN
otFoundException
    } catch (FileNotFoundException e) {
        // Handle exception by providing a default file or
logging the error
        System.out.println("File not found, using default
configuration.");
        loadDefaultConfiguration();
    } catch (IOException e) {
        // Handle other IOExceptions like file read errors
        System.out.println("An error occurred while readin
g the file.");
    }
}
```

- **Conclusion**:
  - **Throw an exception** when the method **cannot handle the error** or when you want to signal an exceptional condition to the calling code.
  - **Catch an exception** when you can **recover** from the error or **gracefully degrade** the functionality of your program.

Creating Custom Exception Classes

- **Why Create Custom Exceptions?**
  - To handle specific error situations that are not covered by standard exceptions.
  - To provide more meaningful error messages that can improve debugging.

- **Choosing a Superclass:**
  - Always choose a superclass that is semantically closest to the type of exception you want to represent. For example:
    - If your exception is due to invalid arguments, extend `IllegalArgumentException`.
    - If it is an error condition in your application logic, consider extending `RuntimeException`.
  - Most of the time, we extend `RuntimeException`.
- **Class Contents:**
  - Typically minimalistic, containing:
    - A default constructor.
    - An overloaded constructor that accepts an error message.
- **Example**:

```
public class NegativeArgumentException extends IllegalArgumer
    // Default constructor
    public NegativeArgumentException() {
        super("Negative argument not allowed");
    }

    // Constructor that accepts a custom message
    public NegativeArgumentException(String msg) {
        super(msg);
    }
}
```

- **Throwing Custom Exceptions:**
  - You can throw your custom exception using the `throw` statement when a specific condition is met.
  - For instance:

```
public void calculateFactorial(int number) {
    if (number < 0) {
        throw new NegativeArgumentException("Factorial is not
    }
}
```

- **Catching Custom Exceptions:**
  - You can catch your custom exception in a try/catch block just like any other exception.
  - For example:

```
try {
    calculateFactorial(-5);
} catch (NegativeArgumentException nae) {
    System.out.println("Caught exception: " + nae.getMessage(
}
```

Nested Try/Catch

- Nested try/catch blocks allow handling exceptions at different levels in your application.
- Example:

```
public void methodA() {
    try {
        methodB();
        System.out.println("A");
    } catch (RuntimeException e) {
        System.out.println("B");
    } catch (Exception e) {
        System.out.println("C");
    } finally {
        System.out.println("D");
    }
```

```
    System.out.println("E");
}

public void methodB() {
    try {
        List<String> list = null;
        System.out.println(list.size());  // This will throw Nul
    } catch (ArithmeticException e) {
        System.out.println("F");
    } finally {
        System.out.println("G");
    }
    System.out.println("H");
}

// Output: G B D E
```

- Example 2:

```
try {
    List<String> list = null;
    System.out.println(list.size()); // Throws NullPointerExcept
} catch (RuntimeException e) {
    try {
        System.out.println(5 / 0); // Throws ArithmeticExceptio
    } catch (ArithmeticException ae) {
        System.out.println("A");
    }
} catch (Exception e) {
    System.out.println("B");
} finally {
    System.out.println("C");
}
System.out.println("D");
```

```
// Output: A C D
```