ERROR HANDLING

Errors default Exceptions
Errors indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

Exceptions are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions.

One way to handle exceptions is using try/catch blocks.

Try-catch-exceptions

The **try** block contains set of statements where an exception can occur. A try block is always followed by a **catch** block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or **finally** block or both.

If an exception occurs in try block then the control of execution is passed to the corresponding catch block. A single try block can have multiple catch blocks associated with it, you should place the catch blocks in such a way that the generic exception handler catch block is at the last(see in the example below).
The generic exception handler can handle all the exceptions but you should place is at the end, if you place it at the before all the catch blocks then it will display the generic message. You always want to give the user a meaningful message for each type of exception rather then a generic message.

Finally Block - No matter whether an exception is thrown or not inside the try or catch block the code inside the finally-block is executed.

Example of Multiple catch blocks

```
class Example2{
   public static void main(String args[]){
     try{
         int a[]=new int[7];
         a[4]=30/0;
         System.out.println("First print statement in try block");
     }
     catch(ArithmeticException e){
        System.out.println("Warning: ArithmeticException");
     }
     catch(ArrayIndexOutOfBoundsException e){
```

```
            System.out.println("Warning: ArrayIndexOutOfBoundsException");
        }
        catch(Exception e){
            System.out.println("Warning: Some Other exception");
        }
       System.out.println("Out of try-catch block...");
     }
}
```

Examples with my code

```
try {
            List<String> file1content = Files.readAllLines(filePath);

            Path filePath2 = Paths.get(file2);
            Files.write(filePath2, file1content);
            return true;
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
```

```
public static void divider(int x) {
    try {
        int result = 10 / x;
        System.out.println(result);
    } catch (ArithmeticException e) {
        System.out.println("Can't divide by 0");
    }
```

```
}
```

Best practices to handle exceptions:
1. Clean Up Resources in a Finally Block or Use a Try-With-Resource Statement
2. Prefer Specific Exceptions
3. Document the Exceptions You Specify
4. Throw Exceptions With Descriptive Messages
5. Catch the Most Specific Exception First
6. Don’t Catch Throwable
7. Don’t Ignore Exceptions
8. Don’t Log and Throw
9. Wrap the Exception Without Consuming it

Runtime / compile

Compile time
The program need not satisfy any invariants. In fact, it needn't be a well-formed program at all. You could feed this HTML to the compiler and watch it barf...
What can go wrong at compile time:

- Syntax errors
- Typechecking errors
- (Rarely) compiler crashes

If the compiler succeeds, what do we know?
The program was well formed---a meaningful program in whatever language.
It's possible to start running the program. (The program might fail immediately, but at least we can try.)

What are the inputs and outputs?
Input was the program being compiled, plus any header files, interfaces, libraries, or other voodoo that it needed to import in order to get compiled.
Output is hopefully assembly code or relocatable object code or even an executable program. Or if something goes wrong, output is a bunch of error messages.

Run time
We know nothing about the program's invariants---they are whatever the programmer put in. Run-time invariants are rarely enforced by the compiler alone; it needs help from the programmer.
What can go wrong are run-time errors:

- Division by zero
- Dereferencing a null pointer
- Running out of memory

Also there can be errors that are detected by the program itself:

- Trying to open a file that isn't there
- Trying find a web page and discovering that an alleged URL is not well formed

If run-time succeeds, the program finishes (or keeps going) without crashing.
Inputs and outputs are entirely up to the programmer. Files, windows on the screen, network packets, jobs sent to the printer, you name it. If the program launches missiles, that's an output, and it happens only at run time.

Stacktrace / debugging

A **stack trace** is a report of the active stack frames at a certain point in time during the execution of a program. A stack trace allows tracking the sequence of nested functions called - up to the point where the stack trace is generated.

Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.
Debugging tactics can involve interactive debugging, control flow analysis, unit testing, integration testing, log file analysis, monitoring at the application or system level, memory dumps, and profiling.

Validation

When receiving input that needs to be validated before it can be used, validate all input before using any of it. You should not change any state in the application or attached systems until all input data has been validated. That way you avoid leaving the application in a half valid state.

Be aware that any JavaScript input validation performed on the client can be bypassed by an attacker that disables JavaScript or uses a Web Proxy. Ensure that any input validation performed on the client is also performed on the server.

Input validation should be applied on both **syntactical** and **semantic** level. Syntactic validation should enforce correct syntax of structured fields (e.g. SSN, date, currency symbol) while semantic validation should enforce correctness of their values in the specific business context (e.g. start date is before end date, price is within expected range).

It is always recommended to prevent attacks as early as possible in the processing of the user's (attacker's) request. Input validation can be used to detect unauthorized input before it is processed by the application.

Implementing input validation
Input validation can be implemented using any programming technique that allows effective enforcement of syntactic and semantic correctness, for example:

- Data type validators available natively in web application frameworks (such as Django Validators, Apache Commons Validators etc)

- Validation against JSON Schema and XML Schema (XSD) for input in these formats
- Type conversion (e.g. Integer.parseInt() in Java, int() in Python) with strict exception handling
- Minimum and maximum value range check for numerical parameters and dates, minimum and maximum length check for strings
- Array of allowed values for small sets of string parameters (e.g. days of week)
- Regular expressions for any other structured data covering the whole input string (^...$) and not using "any character" wildcard (such as "." or "\S")

https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

https://dzone.com/articles/validation-in-java-applications

Expected skills (3/5)

- **Able to identify the relevant parts of a stack trace**

A stack trace is a list of the method calls that the application was in the middle of when an Exception was thrown. But a stack trace can also be accessed by the programmer even if no Exception was thrown.
One of the most important concepts of correctly understanding a stack trace is to recognize that it lists the execution path in reverse chronological order from most recent operation to earliest operation.

- **Able to explain the difference between runtime and compile time errors**

Compile time errors are errors occurred due to typing mistakes, if we do not follow the proper syntax and semantics.
Runtime errors are the errors that are generated when the program is in running state. These types of errors will cause your program to behave unexpectedly or may even kill your program. They are often referred as **Exceptions**.

- **Able to show how errors are handled in the application**

An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. A method is not required to declare in its throws clause any subclasses of Error that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur. That is, Error and its subclasses are regarded as unchecked exceptions for the purposes of compile-time checking of exceptions.

- **Able to explain where and why to use data/input validation**

(See above)

- **Able to demonstrate the debugging process**

IntelliJ - break points