

Error, Exception Handling , Logging & Debugging

What is topic all about

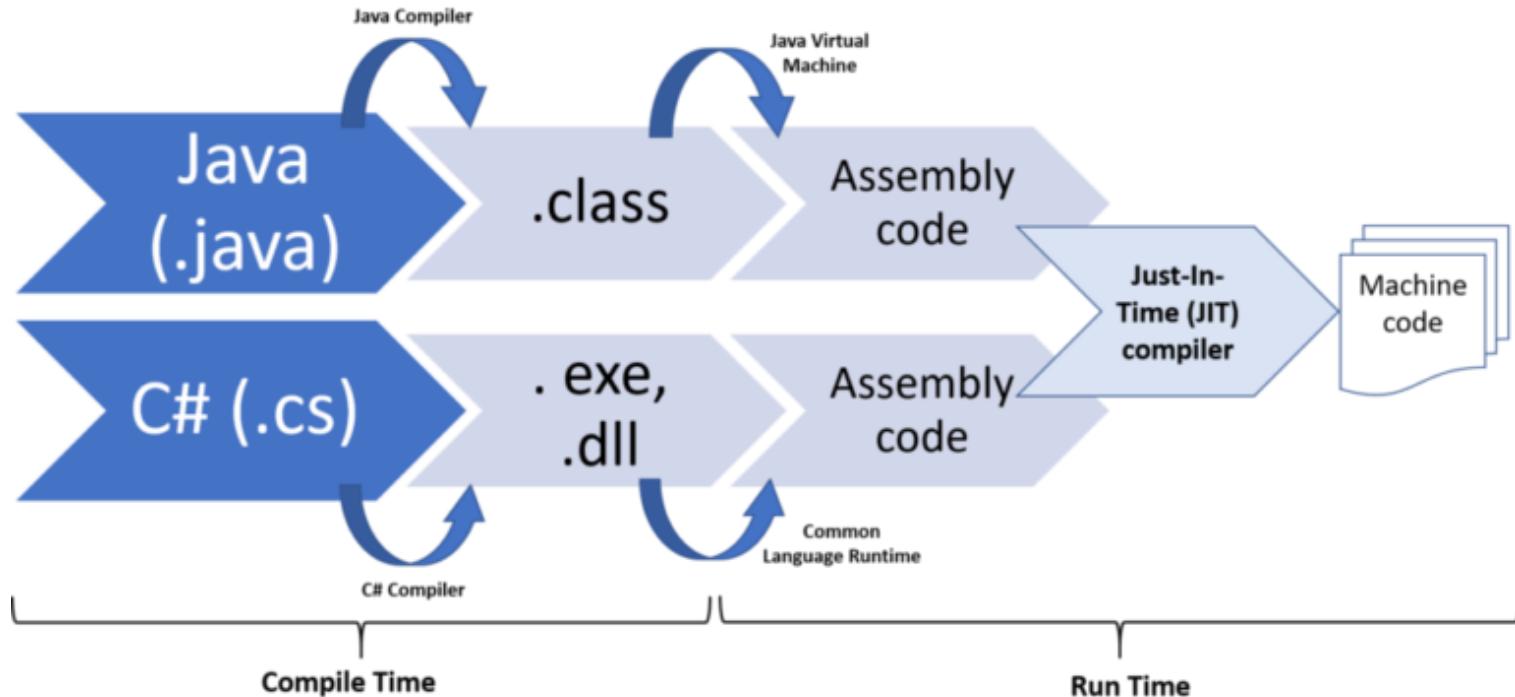
- We want to find out what is wrong with the program and finish the program in a good way.
- We want to know what and how the program doing.
- We want to find out why the program is doing in a wrong way.



When could error happen in java program

- Run Time
- Compile Time

Compile Time vs Run Time



Compile Time

Compile-time is the stage when the Java source code is translated into bytecode by the Java compiler , it will do:

- **Syntax checking:** Checks the source code for syntax errors or violations of the Java language rules.
- **Type checking:** Verifies that the data types used in the program are compatible with each other and that variables, method calls, and other expressions are used correctly.
- **Compilation:** Translated into bytecode (.Class), an intermediate representation that is platform-independent.
- **Binding:** Resolves references to classes, methods, and variables, ensuring they exist and are accessible.
- **Handling checked exceptions:** Enforces handling or declaring checked exceptions in the code.

Compile-time error (example)

```
1  import java.io.*;
2
3  public class CompileTimeError {
4      public static void main(String[] args) {
5
6          System.out.println("start the program!")
7          String file_name = 5;
8          // Create a File object
9          File file = new File(file_name);
10         // Create a BufferedReader object
11         BufferedReader br = new BufferedReader(new FileReader(file));
12
13         // Read the file
14         while ((line = br.readLine()) != null) {
15             System.out.println(line);
16         }
17
18         // Close the BufferedReader
19         br.close();
20         // Handle the exception
21         e.printStackTrace();
22     }
23 }
```

Run Time

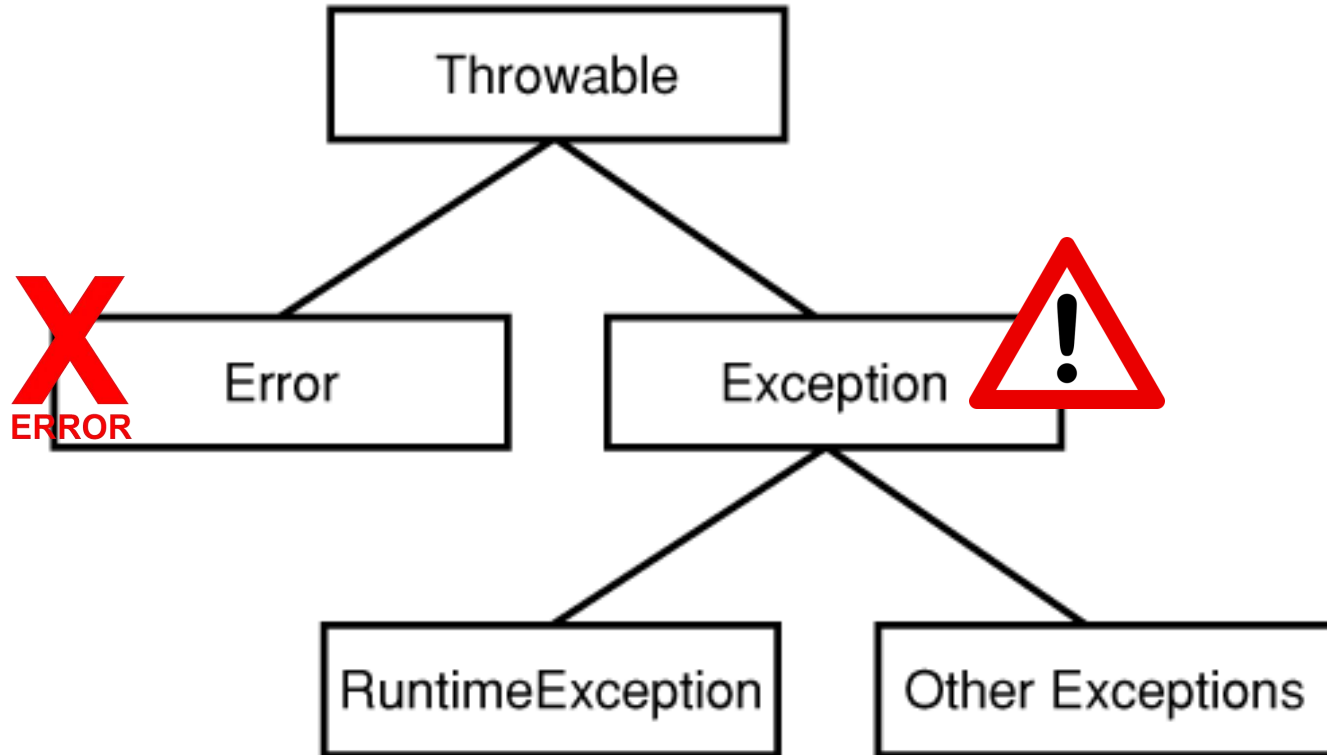
Runtime is the stage when the Java bytecode is executed by the Java Virtual Machine (JVM). During this stage, the following actions are performed:

- **Loading classes:** The JVM loads the required classes and resources into memory.
- **Instantiation:** Objects are created, and memory is allocated for them.
- **Execution:** The JVM interprets and executes the bytecode, following the instructions and performing the necessary actions (e.g., calculations, method calls, etc.).
- **Garbage collection:** The JVM manages memory by cleaning up and deallocating objects that are no longer in use.
- **Handling runtime exceptions:** Unchecked exceptions can occur during runtime, and the program may handle them, or they may cause the program to terminate abnormally.

Run-time Exception(example)

```
1 public class RunTimeErrors {  
2     public static void main(String[] args) {  
3  
4         String str = null;  
5         System.out.println(str.length());  
6  
7         int[] array = new int[5];  
8         System.out.println(array[5]);  
9  
10        int a = 5;  
11        int b = 0;  
12        System.out.println(a / b);  
13    }  
14 }
```


Java Error Handling (Class Hierarchy Overview)



Error

- Error represents a serious problem that the application shouldn't try to catch or handle. It usually indicates a critical system-level issue that the application can't recover from, such as running out of memory, thread death, or class loading problems.
- Errors are derived from the `java.lang.Error` class.
- Developers are not expected to catch and handle Errors in their code, as they often indicate irrecoverable situations.



- java.lang.**Error**
 - java.lang.annotation.**AnnotationFormatError**
 - java.lang.**AssertionError**
 - java.awt.**AWTError**
 - java.nio.charset.**CoderMalfunctionError**
 - javax.xml.parsers.**FactoryConfigurationError**
 - javax.xml.stream.**FactoryConfigurationError**
 - java.io.**IOException**
 - java.lang.**LinkageError**
 - java.lang.**BootstrapMethodError**
 - java.lang.**ClassCircularityError**
 - java.lang.**ClassFormatError**
 - java.lang.reflect.**GenericSignatureFormatError**
 - java.lang.**UnsupportedClassVersionError**
 - java.lang.**ExceptionInInitializerError**
 - java.lang.**IncompatibleClassChangeError**
 - java.lang.**AbstractMethodError**
 - java.lang.**IllegalAccessError**
 - java.lang.**InstantiationError**
 - java.lang.**NoSuchFieldError**
 - java.lang.**NoSuchMethodError**
 - java.lang.**NoClassDefFoundError**
 - java.lang.**UnsatisfiedLinkError**
 - java.lang.**VerifyError**
 - javax.xml.validation.**SchemaFactoryConfigurationError**
 - java.util.**ServiceConfigurationError**
 - java.lang.**ThreadDeath**
 - javax.xml.transform.**TransformerFactoryConfigurationError**
 - java.lang.**VirtualMachineError**
 - java.lang.**InternalError**
 - java.util.zip.**ZipError**
 - java.lang.**OutOfMemoryError**
 - java.lang.**StackOverflowError**
 - java.lang.**UnknownError**

Exception

- Exception is the superclass of all the exceptions from which ordinary programs **may wish to recover**.
- The class **RuntimeException** is a direct subclass of Exception. RuntimeException is the superclass of all the exceptions which may be thrown for many reasons during expression evaluation, but from which recovery may still be possible.
- RuntimeException and all its subclasses are, collectively, the ***run-time exception classes***.



Types of Exception - Checked

Checked Exceptions

- Checked by the compiler during the **compile-time**. They must be either caught using a try-catch block or declared in the method signature using the 'throws' keyword. Examples include IOException, and FileNotFoundException.
- The checked exception classes are Throwable and all its subclasses other than RuntimeException and its subclasses and Error and its subclasses.

Types of Exception - Checked

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class CheckedExceptionExample {

    public static void main(String[] args) {
        try {
            readFile("non_existent_file.txt");
        } catch (FileNotFoundException e) {
            System.err.println("Caught FileNotFoundException: " + e.getMessage());
        }
    }

    public static void readFile(String fileName) throws FileNotFoundException {
        FileInputStream fileInputStream = new FileInputStream(new File(fileName));
    }
}
```

Types of Exception - UnChecked

Unchecked Exceptions: The unchecked exception classes are the **run-time** exception classes and the **error classes**. ***RuntimeException*** classes are unchecked exceptions. Examples include `NullPointerException`, and `ArithmeticException`.

Types of Exception - UnChecked

```
public class UncheckedExceptionExample {  
  
    private static String teststring;  
  
    public static void main(String[] args) {  
        processString(teststring);  
    }  
  
    public static void processString(String input) {  
        int length = input.length();  
        System.out.println("The length of the input string is: " + length);  
    }  
}
```


Custom Checked Exception

We can create custom checked exception by extending **Exception** class

```
1 public class CustomCheckedException extends Exception{  
2  
3     public CustomCheckedException(String message) {  
4         super(message);  
5     }  
6  
7     public CustomCheckedException(String message, Throwable cause) {  
8         super(message, cause);  
9     }  
10 }
```

Custom Checked Exception (Cont.)

```
3 public class CustomCheckedExceptionTest {
4
5     public static void main(String[] args) {
6         try {
7             throwCustomCheckedException(null);
8         } catch (CustomCheckedException e) {
9             e.printStackTrace();
10        }
11        //throwCustomCheckedException();
12    }
13
14    public static void throwCustomCheckedException(String test) throws CustomCheckedException {
15        if (test == null) {
16            throw new CustomCheckedException("test is null");
17        }
18        else {
19            System.out.println("test is not null");
20        }
21    }
22 }
23 }
```

Custom Unchecked Exception

We can create custom unchecked exception by extending **RuntimeException** class

```
public class CustomUncheckedException extends RuntimeException {  
    public CustomUncheckedException(String message) {  
        super(message);  
    }  
  
    public CustomUncheckedException(String message, Throwable cause) {  
        super(message, cause);  
    }  
}
```

Most Common Ones

- **NullPointerException (Unchecked)**: Thrown when trying to use a **null** object where an object is required.
- **ArrayIndexOutOfBoundsException (Unchecked)**: Thrown when an array is accessed with an index that's negative or too large.
- **ClassCastException (Unchecked)**: Thrown when trying to cast an object to a class of which it's not an instance.
- **NumberFormatException (Unchecked)**: Thrown when trying to convert a string to a numeric type but the string doesn't have the correct format.

Most Common Ones (Cont.)

- **IOException(Checked)**: Thrown when an I/O operation fails or is interrupted.
- **FileNotFoundException(Checked)**: Thrown when trying to access a file that doesn't exist.
- **IndexOutOfBoundsException(Unchecked)**: Thrown when trying to access an index that's out of range (not just for arrays, but for any indexed collection).
- **ArithmeticException(Unchecked)**: Thrown when an exceptional arithmetic condition occurs, such as dividing by zero.

Most Common Ones (Cont.)

- **IllegalArgumentException**: Thrown when a method is passed an illegal or inappropriate argument.
- **NoClassDefFoundError**: Thrown when the JVM tries to load a class that was available at compile time but is missing at runtime.
- **OutOfMemoryError**: Thrown by the JVM when it runs out of heap memory, often due to too many objects or memory leaks.
- **StackOverflowError**: Thrown by the JVM when a program's call stack depth exceeds a limit, often due to uncontrolled recursion.

StackOverflowError:

```
1 public class StackOverFlow {  
2     public static void main(String[] args) {  
3         recursiveMethod();  
4     }  
5  
6     public static void recursiveMethod() {  
7         recursiveMethod();  
8     }  
9 }
```

OutOfMemoryError:

```
1  import java.util.*;
2
3  public class OutOfMemoryErrorExample {
4      public static void main(String[] args) {
5          List<long[]> list = new ArrayList<>();
6          while (true) {
7              list.add(new long[Integer.MAX_VALUE / 2]);
8          }
9      }
10 }
11
```


How to Handles Exceptions

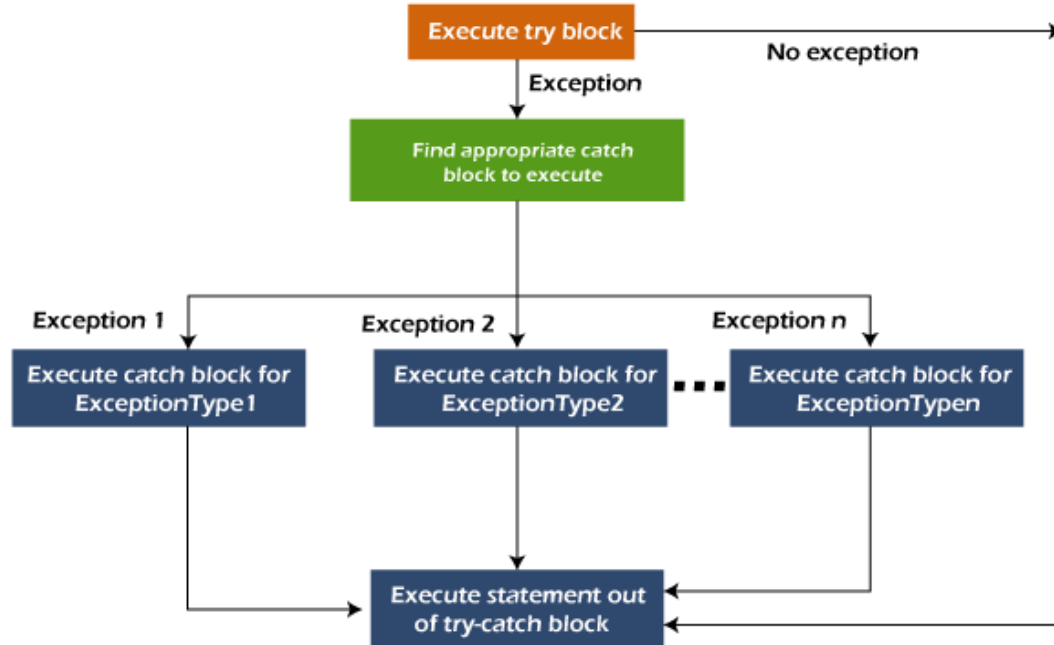
- Throw
- Throws
- Try
- Catch
- Finally

```
1 public class ExceptionHandlingKeyWords {
2     public static void main(String[] args) {
3         try {
4             System.out.println("About to call a function...");
5             doSomethingDangerous();
6             System.out.println("Function executed successfully!");
7         } catch (Exception e) {
8             System.out.println("An error occurred: " + e.getMessage());
9         } finally {
10            System.out.println("This is the finally block, it always gets executed!");
11        }
12    }
13
14    public static void doSomethingDangerous() throws Exception {
15        System.out.println("Inside the function...");
16
17        if (Math.random() < 0.5) {
18            throw new Exception("Dangerous function failed!");
19        }
20
21        System.out.println("Exiting the function...");
22    }
23 }
```

Catch Multiple Exceptions

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

Catch Multiple Exceptions (Cont.)



Catch Multiple Exceptions(Cont.)

```
3 public class MultipleTryCatch {
4     public static void main(String[] args) {
5         try {
6             // This block of code can throw multiple types of exceptions
7             if (Math.random() < 0.5) {
8                 int[] arr = new int[5];
9                 System.out.println(arr[10]); // Potential ArrayIndexOutOfBoundsException
10            } else {
11                Object obj = null;
12                obj.toString(); // Potential NullPointerException
13            }
14        } catch (ArrayIndexOutOfBoundsException e) {
15            // Handle an ArrayIndexOutOfBoundsException
16            System.out.println("Caught an ArrayIndexOutOfBoundsException: " + e.getMessage());
17        } catch (NullPointerException e) {
18            // Handle a NullPointerException
19            System.out.println("Caught a NullPointerException: " + e.getMessage());
20        }
21    }
22 }
```

Catch Multiple Exceptions(Cont.)

```
3 public class MultipleTryCatch_J7 {  
4     public static void main(String[] args) {  
5         try {  
6             // This block of code can throw multiple types of exceptions  
7             if (Math.random() < 0.5) {  
8                 int[] arr = new int[5];  
9                 System.out.println(arr[10]); // Potential ArrayIndexOutOfBoundsException  
10            } else {  
11                Object obj = null;  
12                obj.toString(); // Potential NullPointerException  
13            }  
14        } catch (ArrayIndexOutOfBoundsException | NullPointerException e) {  
15            // Handle either an ArrayIndexOutOfBoundsException or a NullPointerException  
16            System.out.println("Caught an exception: " + e.getMessage());  
17        }  
18    }  
19 }
```

Catch Multiple Exceptions(Cont.)

```
1 package example3.errorHandling;
2
3 public class MultipleTryCatch_J7 {
4     public static void main(String[] args) {
5         try {
6             // This block of code can throw multiple types of exceptions
7             if (Math.random() < 0.5) {
8                 int[] arr = new int[5];
9                 System.out.println(arr[10]); // Potential ArrayIndexOutOfBoundsException
10            } else {
11                Object obj = null;
12                obj.toString(); // Potential NullPointerException
13            }
14        } catch (Exception | ArrayIndexOutOfBoundsException | NullPointerException e) {
15            // Handle either an ArrayIndexOutOfBoundsException or a NullPointerException
16            System.out.println("Caught an exception: " + e.getMessage());
17        }
18    }
19 }
```

try-with-resources Statement

This statement ensures that each resource opened in a try block is closed **automatically** when the try block exits, regardless of whether an exception was thrown. Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement **`java.io.Closeable`**, can be used as a resource. **`FileInputStream`, `FileOutputStream`, `PrintWriter`, `Scanner`, `Socket`**, etc.

try-with-resources Statement (Old)

```
1 package example3.errorHandling;
2
3 import java.io.BufferedReader;
4 import java.io.FileReader;
5 import java.io.IOException;
6
7 public class TryWithResourcesExampleOldWay {
8     public static void main(String[] args) {
9         BufferedReader br = null;
10        try {
11            br = new BufferedReader(new FileReader("test.txt"));
12            String line;
13            while ((line = br.readLine()) != null) {
14                System.out.println(line);
15            }
16        } catch (IOException e) {
17            e.printStackTrace();
18        } finally {
19            if (br != null) {
20                try {
21                    br.close();
22                } catch (IOException e) {
23                    e.printStackTrace();
24                }
25            }
26        }
27    }
28 }
29
```

try-with-resources Statement (New)

```
1 package example3.errorHandling;
2
3 import java.io.BufferedReader;
4 import java.io.FileReader;
5 import java.io.IOException;
6
7 public class TryWithResourcesExample {
8     public static void main(String[] args) {
9         try (BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {
10             String line;
11             while ((line = br.readLine()) != null) {
12                 System.out.println(line);
13             }
14         } catch (IOException e) {
15             e.printStackTrace();
16         }
17     }
18 }
19
20
21
```

Exception Handling Best Practices

- **Use appropriate exception types:** Select the most specific exception types to accurately represent the error, making it easier to understand the issue and handle the exception appropriately.
- **Throw specific exceptions use general for the rest:** Instead of generic exceptions like **Exception** or **RuntimeException**, throw specific exceptions that precisely describe the problem, aiding developers in understanding and handling.
- **Avoid using exceptions for control flow:** Don't use exceptions for control flow, as it can make code harder to understand and maintain. Prefer alternative methods for managing normal program flow.
- **Provide informative error messages:** When throwing exceptions, include meaningful error messages with context, helping developers diagnose issues more easily and providing better understanding of the problem.

Exception Handling Best Practices(Cont.)

- **Log exceptions:** Log exceptions and stack traces for easier debugging and troubleshooting, but be cautious about logging sensitive information to prevent security vulnerabilities.
- **Use checked exceptions for recoverable errors:** Employ checked exceptions when the exception is expected to be caught and handled by the caller, enforcing proper exception handling during compile-time.
- **Use unchecked exceptions for programming errors:** Utilize unchecked exceptions for programming errors or unrecoverable situations, reducing method signature clutter and improving overall code readability.

Go over questions

- When could an error happen in a Java program?
- What are the differences between Error and Exception in Java?
- What types of Exception does java have ?
- How to customize those type of Exceptions?
- What are the difference between those type of Exceptions?
- Which type of exceptions in Java does the RuntimeException and Error classes belong to?
- What does finally do?
- What is the best practice to handle multiple try catch?
- Is using exception for control flow a good idea? What should we do?
- Why try-with-resource statement is recommended?
- What are the difference between ClassNotFoundException and NoClassDefError ?
- What are the difference between Heap and Stack in Java ?

Logging – What is logging



Logging – It's Not Important, Until It Is

- How do we handle **production** issues?

Logging – Why Logging

- **Debugging and troubleshooting:** Logging helps developers identify and fix issues in the application by providing valuable insights into the application's behavior, errors, and performance.
- **Audit trail:** Logging creates a record of the application's activities, which can serve as an audit trail. This is particularly important in industries with strict compliance and regulatory requirements.
- **Documentation:** Logs can serve as a source of documentation for understanding how the application behaves under various conditions.

Logging – Why Logging (Cont.)

- **Security:** Analyzing log data can reveal security incidents or breaches, allowing developers to address vulnerabilities and take corrective actions.
- **Performance optimization:** Log data can help developers identify performance bottlenecks and optimize the application for better performance and resource utilization.
- **Root cause analysis:** In case of failures or issues, logs can provide crucial information for identifying the root cause and help developers implement long-term solutions.

Where to Log

- **Pretty much Everything:**
 - In your business layer - all important modules (almost everything)
 - Important operations in your data layer (saving / loading, DB failures, etc.)
 - Start with no logging in display layer, only add when absolutely necessary

Where to log(Cont.)

- **Log Exceptions and Errors** : In Catch Block
- **Log debugging information**: Log development/debugging insights, but avoid sensitive data and disable in production.
- **Log audit trails**: Record events for audit trails, especially in industries with strict compliance requirements.
- **Log security events**: Note security-related events, e.g., attempted breaches, security setting changes, detected vulnerabilities.

Where to log(Cont.)

- **Log user actions:** Record critical user actions like creating, updating, or deleting records in code.
- **Log access/authentication events:** Capture user access, authentication, and authorization details at appropriate code locations.
- **Log performance metrics:** Measure and log key metrics like response times and resource utilization strategically.
- **Log external service interactions:** Document interactions with external services, e.g., API calls, database queries.

Where to log(Cont.)

```
4 public class Log4j2Demo {
5     private static final Logger logger = LogManager.getLogger(Log4j2Demo.class);
6
7     public void userQueryDB() {
8         long startTime = System.currentTimeMillis();
9         logger.debug("userQueryDB Started at: {} ", startTime);
10
11         try {
12             User user = User.login();
13             logger.info("Logged in User is: {} ", user.getName());
14
15             boolean querySuccess = User.queryDB();
16             logger.info("Database query success is: {} ", querySuccess);
17
18         } catch (Exception e) {
19             logger.error("An error occurred during the database query: ", e);
20         }
21
22         long endTime = System.currentTimeMillis();
23         long timeTaken = endTime - startTime;
24         logger.debug("userQueryDB completed. Time taken: {} ms", timeTaken);
25     }
26 }
```

Logging Best Practices

- **Choose a logging framework:** Select a widely-used and robust logging framework, such as Log4j, SLF4J.
- **Use log levels appropriately:** Categorize log messages based on severity (TRACE, DEBUG, INFO, WARN, ERROR, FATAL) to filter log output according to your needs.
- **Include contextual information:** Add timestamps, class names, method names, and line numbers to log messages for easier issue diagnosis.

Logging Best Practices (Cont.)

- **Use parameterized logging:** Favor parameterized logging over string concatenation for better performance and readability.
- **Log file rotation:** Implement log file rotation to avoid large log files that could cause performance issues or disk space exhaustion.
- **Configure log filtering:** Filter log messages based on criteria like log level or

Logging Best Practices (Cont.)

- **Secure sensitive data:** Be cautious when logging sensitive information (e.g., personally identifiable information or passwords) as log files can be a target for attackers.
- **Use asynchronous logging:** Consider asynchronous logging to minimize logging's impact on application performance.
- **Maintain log format consistency:** Use a consistent log format throughout your application to simplify parsing and analysis.

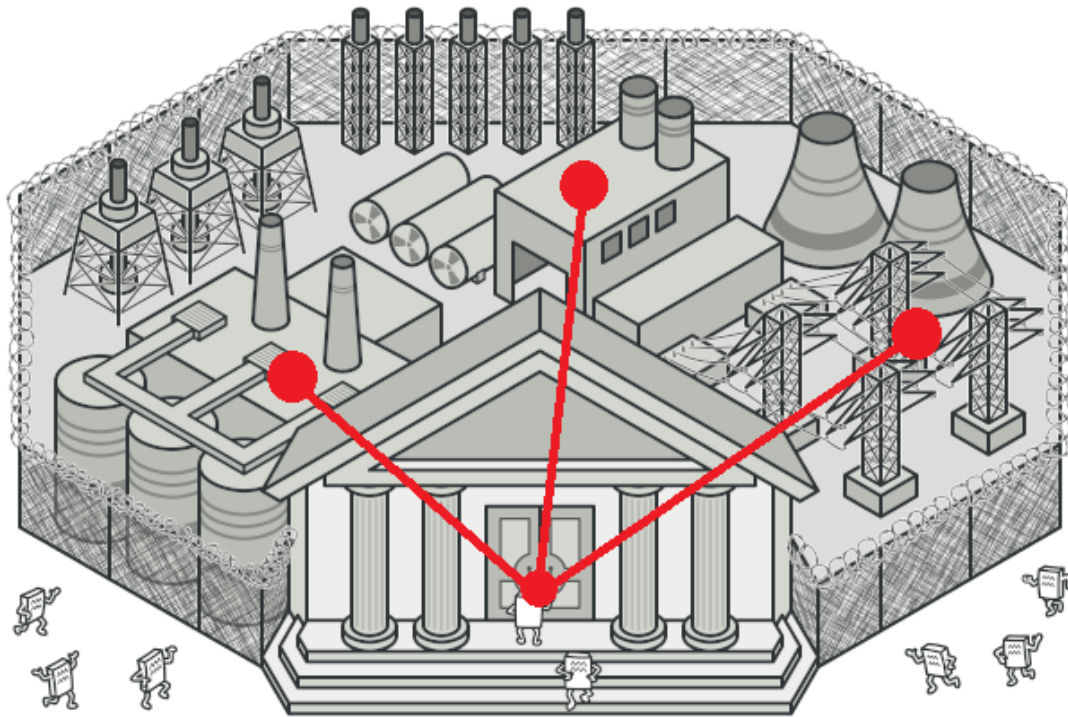
Java Logging Frameworks

- **java.util.logging (JUL)**: Java's built-in logging API, provided by the platform itself.
- **Log4j**: A powerful and flexible logging framework from Apache, capable of logging to multiple destinations with fine-grained control over log levels.
- **Logback**: The successor to Log4j, offering improvements such as increased speed and a more modern API. Designed to work seamlessly with SLF4J.
- **Log4j2**: The improved version of Log4j, with significant performance and flexibility improvements, especially in multi-threaded applications.
- **Commons Logging (JCL)**: A logging abstraction layer from Apache that works with various logging implementations, but has been somewhat superseded by SLF4J.

Special One - SLF4J

- **SLF4J (Simple Logging Facade for Java)**: Not a logging implementation itself, but provides a common interface for various logging frameworks, allowing for easy switching of underlying logging system.

SLF4J(Simple Logging Facade for Java)



SLF4J(Simple Logging Facade for Java)

SLF4J allows you to switch between different logging libraries without changing your application code. By using SLF4J, you can change the underlying logging implementation (e.g., from Log4j2 to Logback) just by updating the configuration and adding the appropriate library to your project, without modifying the actual logging calls in your code.

Let's talk about log4j2

- One of the major reason - **Performance** !

Asynchronous Logging allows the main application thread to continue executing without waiting for the log events to be written to their destination, such as a file or console.

Log4j2 – Set up to run

```
<dependencies>
  <!-- Log4j2 API -->
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.14.1</version>
  </dependency>
  <!-- Log4j2 Core -->
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.14.1</version>
  </dependency>
</dependencies>
```

Log4j2 – Core and API

- The **Log4j API** is a set of interfaces and classes that define the logging API used by application developers. It provides the basic functionality required for logging, such as logger creation, logging methods, and log levels
- The **Log4j Core** is the actual implementation of the logging framework, providing the functionality behind the Log4j API.
- The **Log4j Core** also responsible for handling the logging configuration (e.g., reading **log4j2.xml**), creating and managing logger instances, and directing log messages to the appropriate destinations based on the configuration

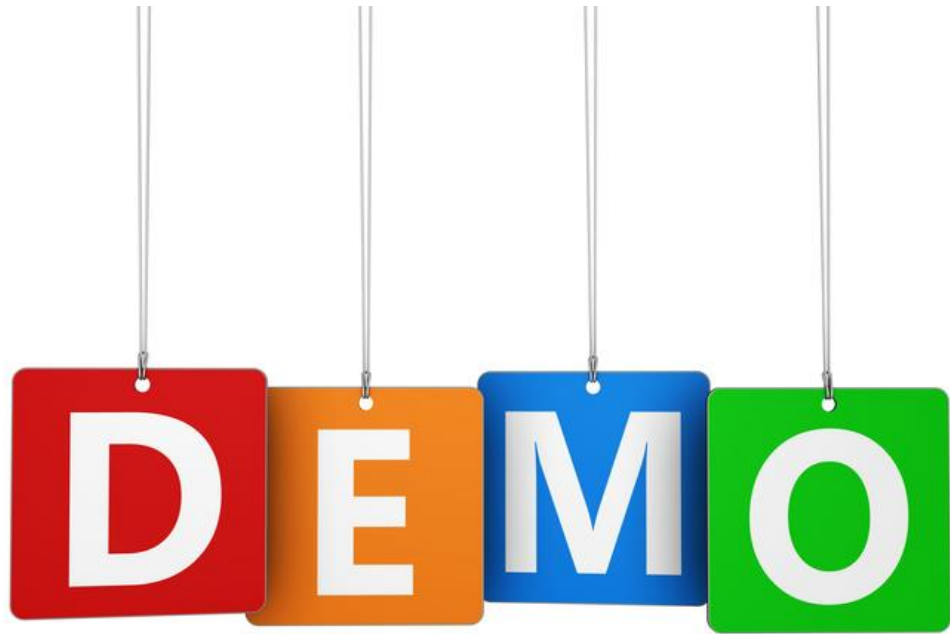
Log4j2 – log level (From Low to High)

- **TRACE:** This level is used for fine-grained, detailed debugging information that you usually don't need during normal operations. Trace level logging can be very verbose.
- **DEBUG:** Debug level is used for general debugging purposes, providing more information than the INFO level. It's helpful during development or when diagnosing issues.
- **INFO:** The INFO level is used for general, informative messages that indicate the normal operation of your application. These messages are not overly verbose and provide a high-level view of what your application is doing.

Log4j2 – log level (From Low to High Cont.)

- **WARN:** The WARN level is used for potentially problematic situations that don't necessarily cause an error but might be a sign of an issue. These messages indicate that something unexpected happened, or that the system is approaching a threshold or limit.
- **ERROR:** The ERROR level is used for serious issues that prevent the normal operation of your application. These messages usually indicate a failure, an exception, or an error in the application logic.
- **FATAL:** The FATAL level is used for critical issues that cause the application to terminate. These messages are very rare and usually indicate a severe problem that requires immediate attention.

Log4j2 – Demo



LogBack vs Log4J2

- **Performance:**

- Log4j2: Generally performs better, especially in asynchronous logging scenarios, due to its improved architecture and optimized design.
- Logback: Performs well in most use cases but may be slightly slower than Log4j2 in certain scenarios, particularly when using asynchronous logging.

- **Configuration:**

- Log4j2: Offers a more complex and flexible configuration system with support for XML, JSON, YAML, and properties files. Allows the use of plugins for extending functionality.
- Logback: Has a simpler and more straightforward XML-based configuration system. Easier for beginners to understand but may not be as flexible as Log4j2's configuration.

LogBack vs Log4J2 (Cont.)

- **SLF4J Support:**

- Log4j2: Can be used with SLF4J through the **log4j-slf4j-impl** adapter library.
- Logback: Is the direct implementation of the SLF4J API, ensuring seamless integration with SLF4J without the need for any additional adapters or bridges.

- **Filtering:**

- Log4j2: Provides a robust filtering system that can be extended through plugins. Allows complex filtering rules, including composite and dynamic filters.
- Logback: Supports filtering through built-in filter classes like **ch.qos.logback.core.filter.EvaluatorFilter** and **ch.qos.logback.classic.filter.ThresholdFilter**. Offers fewer filtering options compared to Log4j2.

SLF4J – Switching from Lgo4J2 To Logback

```
<dependencies>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.32</version>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-slf4j-impl</artifactId>
    <version>2.17.1</version>
  </dependency>
  ...
</dependencies>
```

SLF4J – Switching from Lgo4J2 To Logback(Cont.)

```
<dependencies>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.32</version>
  </dependency>
  <dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.9</version>
  </dependency>
  ...
</dependencies>
```

SLF4J – Switching from Lgo4J2 To Logback(Cont.)

Log4J2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
    </Console>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="Console" />
    </Root>
  </Loggers>
</Configuration>
```

SLF4J – Switching from Lgo4J2 To Logback(Cont.)

Logback.xml

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
  </appender>
  <root level="info">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```


SLF4J – Switching from Lgo4J2 To Logback(Cont.)

Codes stay the SAME!

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyApp {
    private static final Logger logger = LoggerFactory.getLogger(MyApp.class);

    public static void main(String[] args) {
        logger.info("This is an info level log message.");
        logger.error("This is an error level log message.");
    }
}
```

Debugging – What causes the bug

There are all kinds of situations that you may create the bugs.

- When misunderstand the BL
- When it didn't cover 100% BL
- When failed to handle boundary conditions (zero, max , min)
- When data type mismatched
- When system and environment are different.
- When data is different (dev, prod)
- When your internet connection slow and fast (javascript)


Debugging – Log (Stack Trace)

When error happened in the prod, what should we do ? read the log!

A stack trace in Java is a representation of the call stack at a specific point in time, typically when an exception is thrown. It provides a series of method calls that led to the exception, helping you identify the cause of the problem

Debugging – Log (Stack Trace Cont.)

Simple Example (**topmost method call**):

Exception in thread "main" java.lang.NullPointerException
 **CALL** at com.example.myproject.Book.getTitle(Book.java:16)
at com.example.myproject.Author.getBookTitles(Author.java:25)
at com.example.myproject.Bootstrap.main(Bootstrap.java:14)

```
15    public String getTitle() {  
16        System.out.println(title.toString());  
17        return title;  
18    }
```

Debugging – Log (Stack Trace Cont.)

Chain Exception Example (Find Root cause):

Given code below:

```
34 public void getBookIds(int id) {  
35     try {  
36         book.getId(id);    // this method it throws a NullPointerException on line 22  
37     } catch (NullPointerException e) {  
38         throw new IllegalStateException("A book has a null property", e)  
39     }  
40 }
```

```
Exception in thread "main" java.lang.IllegalStateException: A book has a null property  
    at com.example.myproject.Author.getBookIds(Author.java:38)  
    at com.example.myproject.Bootstrap.main(Bootstrap.java:14)  
Caused by: java.lang.NullPointerException  
    at com.example.myproject.Book.getId(Book.java:22)  
    at com.example.myproject.Author.getBookIds(Author.java:36)  
    ... 1 more
```

Debugging - Best Practices

- Clearly understand the issue, gather information like error messages, stack traces, and reproduction steps.
- Ensure consistent issue reproduction for easier debugging and verifying resolution.
- Isolate the problem with the simplest test case that still reproduces the issue.
- Divide code into smaller units, test independently, and narrow down the bug location.

Which code is easier to debug ?

```
private void parallel(ServerRequest req, ServerResponse res) {  
    int count = count(req);  
  
    Multi.range(0, count)  
        .flatMap(i -> Single.create(CompletableFuture.supplyAsync(  
            () -> client().get().request(String.class), EXECUTOR))  
            .flatMap(Function.identity()))  
            .collectList()  
            .map(it -> "Combined results: " + it)  
            .onError(res::send)  
            .forSingle(res::send);  
}
```

```
private void parallel(ServerRequest req, ServerResponse res) throws Exception {  
    try (var exec = Executors.newVirtualThreadPerTaskExecutor()) {  
        int count = count(req);  
  
        var futures = new ArrayList<Future<String>>();  
        for (int i = 0; i < count; i++) {  
            futures.add(exec.submit(() -> callRemote(client)));  
        }  
  
        var responses = new ArrayList<String>();  
        for (var future : futures) {  
            responses.add(future.get());  
        }  
  
        res.send("Combined results: " + responses);  
    }  
}
```

Debugging - Best Practices

- Write unit tests to validate components, catch early issues, and ensure correct code behavior.
- Seek help from colleagues or online communities when stuck, as fresh perspectives can quickly identify issues.
- Add strategic logging to track execution flow and variable values, while avoiding sensitive information.
- Use an integrated debugger (e.g., IntelliJ IDEA, Eclipse) for breakpoint setting, code stepping, and variable inspection.

Debugging using IDE



Projects

- **Logging:**

- **You must use logging in your projects**, set up logging to your environment's database or to a file
- Use log4j or logback, that's the easiest way

- **Error handling:**

- **You must use error handling as appropriate in your projects**, if your code could throw an exception you should be catching it and handling it at some level.
- The minimum for exception handling is logging, but to get an A+ you should have error handling as part of your design.