Lesosn-12 – Exception Handling

1. Checked Exception

```
BufferedReader ob = new BufferedReader(new InputStreamReader(System.in));
        String inp = ob.readLine();
```

2. Unchecked/ Runtime Exception

```
System.out.println(12/0); // DivisionByZeroException
```

3. Errors - StackOverFlowError

```
public static void getData()  {
    getData();
}
```

Inside main() – Cause OutOfMemeoryError

```
String t = "Hello";
  for(int i=0;i<50; i++) {
        t = fun(t);
    }

public static String fun(String s) {
     return s + s;
}
```

Examples of Uncheked/Runtime exceptions Scenario

```
// Unchecked Exceptions - Run time Exceptions
        // 1. Arithmetic Exception
        int x = 0;
        if(x==0) throw new ArithmeticException();
        //System.out.println(10/0);

        // 2. NullPointerException
        String s=null;
        //System.out.println(s.length());

        Float[] f = new Float[10];
    //  System.out.println(f[0].compareTo(20.5f));
```

```
        // 3. ClassCastException
        Object o = new Integer(30); // Not available from JDK 9
        // Replace like below
           Object o = Integer.valueOf(10);

//      String s1 = (String)o;

// 4. ArrayIndexOutOfBoundsException
        String[] col = new String[3];
//      System.out.println(col[3]);

 //5. NumberFormatException - Paring may face this situation
        String num = "20s";//"20" No issue
        //int a = Integer.parseInt(num);
        //int a = Integer.valueOf(num);
//6. InputMismatchException - Get the input the from the console
        Scanner input = new Scanner(System.in);
        System.out.println("Enter a int value");
        int b = input.nextInt();
```

Custom/ User defined Exception

```
int age = -10;
        if (age<0)
            throw new InValidAgeException();

        }
public class InValidAgeException extends Exception{

}
```

Slide – 21

Because exceptions of type RuntimeException are unchecked, they can also be used by developers to indicate a problem that needs to be corrected (useful during development, not for production code).

Explanation

- **Development**: RuntimeException is useful for quickly identifying and addressing issues, allowing developers to focus on core functionality without extensive error handling.

- **Production**: Proper exception handling, including catching and logging exceptions, is essential to ensure a good user experience, maintainability, and error recovery.

By understanding the appropriate use of unchecked exceptions like RuntimeException, developers can write more robust and reliable code that balances the needs of both development and production environments.

About Checked Exceptions

- **Checked exceptions** are enforced by the Java compiler to ensure proper error handling.
- Java enforces that methods must either handle a checked exception (using a try-catch block) or declare it (using the throws keyword).
- **Compile-time checking** helps catch errors early, leading to more reliable and robust code.
- **Method signature declarations** make the code self-documenting and easier to maintain.
- **Mandatory handling** encourages developers to address potential error conditions and prevents oversight.

By enforcing these rules, Java aims to promote better programming practices and enhance the stability and maintainability of applications.

Slide – 26

Point 2b, Explanation

partially handle the exception in a **catch** block, and then *re-throw* the exception to allow other methods in the call stack to handle it further.

Why should I rethrow?

Refer Demo: lesson12.checkedexceptions.soln2b

Separation of Concerns:

- The writeInfo() method is responsible for writing to a file. If it fails, it logs a warning and throws the exception to the caller.
- The caller (main method in Main class) is responsible for handling the failure in a way that makes sense for the application as a whole.

Handling at the Appropriate Level:

- By rethrowing the exception, the method allows higher-level methods (like main) to decide how to handle the error. This could include logging additional context, performing cleanup, or shutting down the application gracefully.
- To Maintain a Clear Flow and keeping the code organized.

Better Scenario – Real time

```
public class DataAccessLayer {
   private static final Logger LOG
    = Logger.getLogger(DataAccessLayer.class.getName());

   public void fetchData() throws SQLException {
      try {
         // Simulating database access
         throw new SQLException("Database connection error");
      } catch (SQLException e) {
         LOG.warning("Error fetching data from database");
         // Log and rethrow to allow higher layers to handle it
         throw e;
      }
   }
}

public class ApplicationLayer {
   private static final Logger LOG
    = Logger.getLogger(ApplicationLayer.class.getName());

   public void performOperation() {
      DataAccessLayer dal = new DataAccessLayer();
      try {
         dal.fetchData();
      } catch (SQLException e) {
         LOG.severe("Critical error: Unable to fetch data. Operation aborted.");
         // Perform application-specific handling, such as rollback or alerting
         System.out.println("Operation failed due to database error");
      }
   }

   public static void main(String[] args) {
      ApplicationLayer app = new ApplicationLayer();
      app.performOperation();
   }
}
```

Explanation

- **DataAccessLayer:** Handles the database operations. If an error occurs, it logs the error and rethrows the exception.

- **ApplicationLayer:** Contains the business logic. It calls the fetchData method and handles the exception appropriately, including logging and alerting.

By rethrowing the exception, you allow the ApplicationLayer to handle the error in a way that makes sense for the overall application, maintaining a clear separation of responsibilities and ensuring that errors are managed appropriately at all levels of the application.

Bad Practice of catching Exception e instead of Specific? Why?

```
try {
        . . .
}
catch(Exception e) {
        . . .
}
```

Answer:

Catching Exception can potentially mask programming errors and make debugging difficult. Always prefer catching specific exceptions to ensure that you handle errors appropriately and maintainable code.

Nested Try/Catch

Nested try-catch blocks are useful when different parts of your code need specific exception handling, especially when dealing with multiple resources or complex operations.