Module-4:
Exception Handling and Errors
All the issues ignored during the Compilation Process

PRAMOD NAIK

Module-4: EXCEPTION HANDLING (5hrs)

Exception Handling: Benefits of exception handling, the classification of exceptions, exception hierarchy, checked exceptions and unchecked exceptions, usage of try, catch, throw, throws and finally. rethrowing exceptions, exception specification, built in exceptions, creating own exception sub classes.

What is Exception?:

In Java, Exception is an **unwanted** or **unexpected event**, which occurs during the **execution of a program**, i.e. at run time, that **disrupts the normal flow** of the program's instructions.

It is an object that is thrown at runtime when an unexpected situation (error) occurs.

Exception Handling:

Exception handling in Java is a mechanism to **manage runtime errors**, ensuring that the **normal flow** of the application is maintained. It allows a program to deal with unexpected events or errors without crashing.

What is Error:

Exceptional conditions that the application cannot normally **predict** or **recover** from. They are also ignored throughout the compilation process.

Errors represent **irrecoverable conditions** such as **Java virtual machine (JVM) running out of memory**, **memory leaks**, **stack overflow errors**, **library incompatibility**, **infinite recursion**, etc. Errors are usually **beyond the control of the programmer**, and we should not try to handle errors.

Some examples of errors:

- OutOfMemoryError
- VirtualMachineErrors
- AssertionError

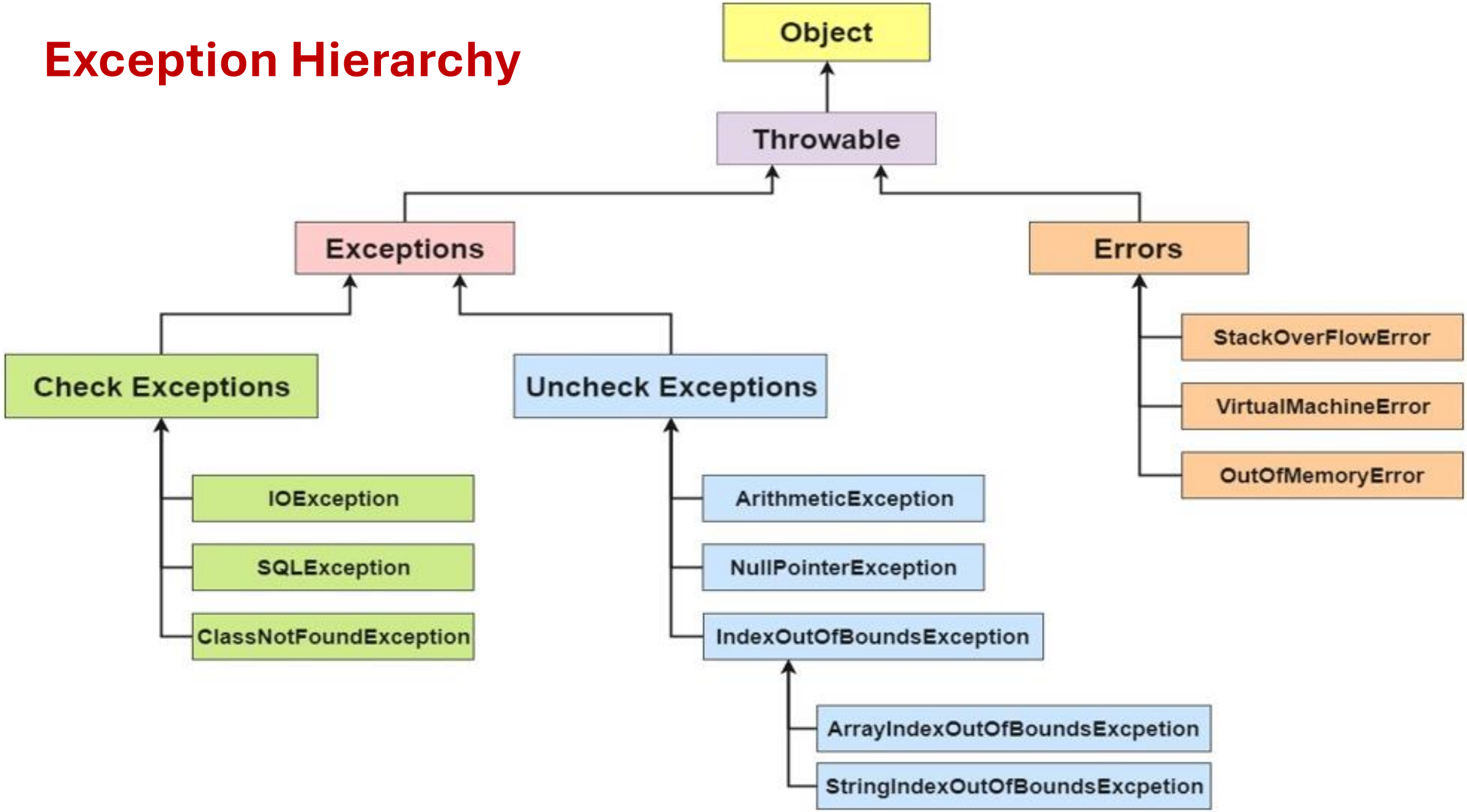
Diff between Error and Exception:

1. **Error:** Represents serious, unrecoverable issues that occur in the runtime environment. Errors are usually external to the application and indicate a problem that the application **cannot handle** or **recover** from (e.g., memory exhaustion, system crashes).
2. **Exception:** Represents conditions that a program might want to catch and handle. These are usually conditions that arise because of **logical** or **runtime** issues in the application (e.g., invalid user input, I/O problems, divide by zero).

Exception Hierarchy:

All exception and error types are **subclasses** of the class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Error is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE).

Exception Hierarchy



1. Checked or compiled-time exceptions:

Arise when something goes wrong in your code but is **potentially recoverable**. Checked at **compilation time**.

Some checked Exceptions include:

- **SQLException:** It occurs due to various database-related issues
- **IOException:** Occurs when an input-output operation fails or is interrupted.
- **ClassNotFoundException:** When The class is not available in the classpath or the class file is missing.
- **FileNotfound Exception:** When The specified file does not exist

Handling:

- Must be **caught** or **declared** because it's a checked exception.
- Usually handled in a **try-catch block** where database queries are executed.

2. Unchecked or Runtime Exceptions in Java:

Unchecked exceptions are classes that inherit from **RuntimeException**. They are not tested at compile time, but rather during runtime.

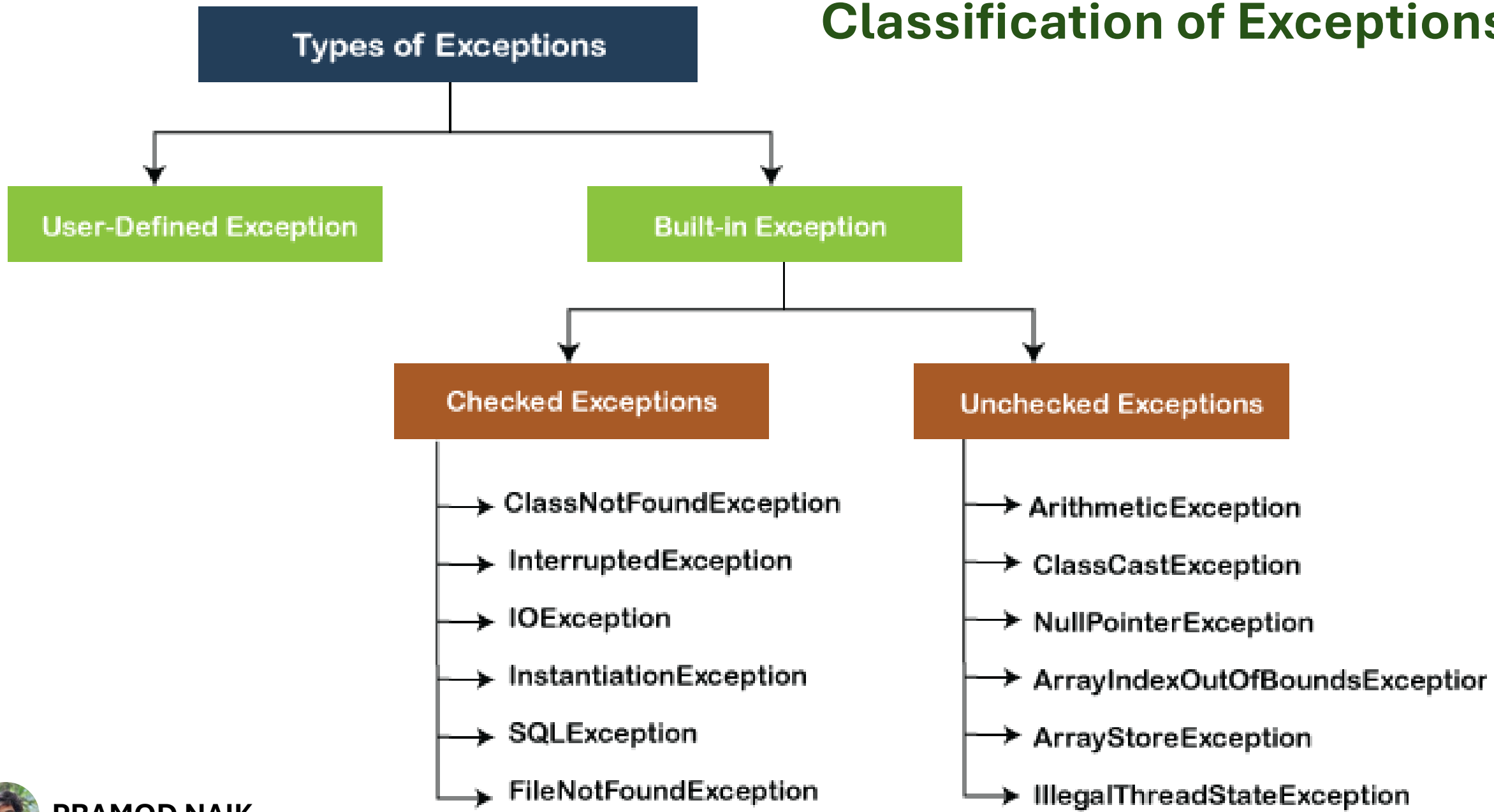
Some unchecked exceptions include:

- **ArithmeticException:** This exception is thrown when an illegal arithmetic operation is attempted, such as division by zero
- **NullPointerException:** This exception is thrown when an application tries to use a null object reference where an object is required.
- **ArrayIndexOutOfBoundsException:** This exception is thrown when you try to access an array with an invalid index, such as a negative index or an index greater than or equal to the array size.
- **NumberFormatException:** This exception is thrown when an attempt is made to convert a string to a number
- **InputMismatchException:** This exception is thrown when an attempt is made to read data of the wrong type using a scanner. For example, if you try to read an integer but provide a non-numeric input.

Classification of Exceptions:

In Java, exceptions are classified into two broad categories: **Checked Exceptions** and **Unchecked Exceptions**. These exceptions are further divided into different types based on their hierarchy and how they are handled.

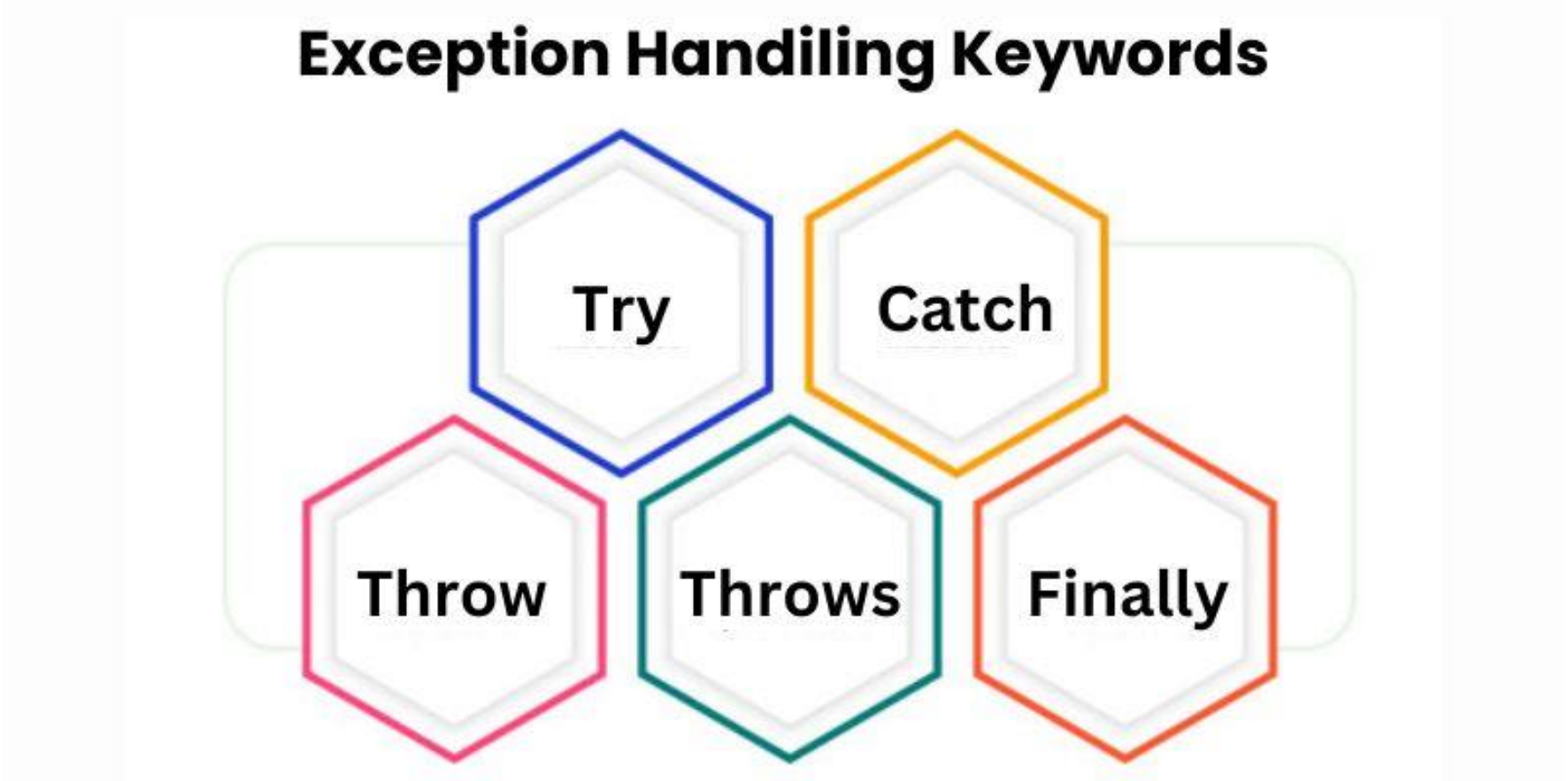
Classification of Exceptions:



PRAMOD NAIK

Important Keywords Related to Java Exception Handling:

In Java, exception handling is done using **try**, **catch**, **throw**, **throws**, and **finally**. These constructs allow you to handle runtime errors gracefully.



PRAMOD NAIK

1. **try: (Holds** Risky Code)

- The try block contains the code that **might throw an exception**. If an exception occurs, it **jumps to the catch block**.
- The try block is the block where the block of code that is needed to be **checked for exceptions** is placed. The try block is **followed by a catch or finally block, it cannot stand alone.**

Example:

```
try {
    // Code that might throw an exception
    int result = 10 / 0;  // This will throw ArithmeticException
}
```

2. catch:

- The catch block is used to **handle the exception thrown by the try block**. You can catch different types of exceptions by **chaining** multiple catch blocks. It is declared after the try block.

Example:

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Cannot divide by zero");
}
```

3. finally:

- The finally block contains code that will **always execute** after the **try** and **catch** blocks, regardless of whether an **exception was thrown**. It is typically used for cleanup actions, like closing resources.
- Using the finally block we can **execute an important piece** of **code** because the finally block will be executed **regardless** of what the outcome is from the try block.

Example:

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    System.out.println("This block is always executed");
}
```

PRAMOD NAIK

4. throw:

- The throw keyword is used to explicitly throw an exception.
- Using the throw keyword we can **throw a predefined exception.**

Example:

```
public void validateAge(int age) {
    if (age < 18) {
        throw new ArithmeticException("Age must be at least 18");
    }
}
```

5. throws:

The throws keyword is used in **method signatures** to declare that a **method might throw one or more exceptions**. This is useful for checked exceptions.

Example:

```
public void readFile() throws IOException {
    // Code that might throw IOException
    FileReader file = new FileReader("file.txt");
}
```

Creating own Exception sub-classes:

In Java, you can create your own custom exception classes by **extending** the built-in **Exception class** (for checked exceptions) or **RuntimeException** class (for unchecked exceptions). Custom exceptions are useful when you want to **throw specific errors** related to your application's business logic.

Steps to Create a Custom Exception Class:

- **Extend** the Exception class (for checked exceptions) or RuntimeException class (for unchecked exceptions).
- Provide **constructors** to initialize the exception message and optionally pass other information like the cause of the exception.

1. Creating a Custom Checked Exception:

A checked exception must be declared in the method signature using the throws keyword and must be handled (caught) in the calling method.

Example:

```
// Custom checked exception class
class InvalidAgeException extends Exception {
    // Constructor that accepts a message
    public InvalidAgeException(String message) {
        super(message);
    }
}
```

```java
public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            checkAge(15);  // This will throw InvalidAgeException
        } catch (InvalidAgeException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }

    // Method that throws the custom checked exception
    public static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age is less than 18. Access denied.");
        } else {
            System.out.println("Access granted");
        }
    }
}
```

Explanation:

InvalidAgeException is a custom checked exception that must be declared in the method signature (throws InvalidAgeException).

When the age is less than 18, an InvalidAgeException is thrown with a custom message.

The calling method (main()) must catch or declare this exception.

2. Creating a Custom Unchecked Exception

An unchecked exception is created by extending RuntimeException. Unlike checked exceptions, these do not need to be declared in the method signature or caught.

Example: Creating a Custom Unchecked Exception

```
// Custom unchecked exception class
class InsufficientFundsException extends RuntimeException {
    public InsufficientFundsException(String message) {
        super(message);
    }
}
```

Explanation:
InsufficientFundsException is a custom unchecked exception (subclass of RuntimeException).
In this case, if the withdrawal amount exceeds the balance, an InsufficientFundsException is thrown.
This exception **does not need to be declared in the method signature,** nor does it need to be caught unless the developer chooses to.

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        withdraw(500, 300);  // This will throw InsufficientFundsException
    }

    // Method that throws the custom unchecked exception
    public static void withdraw(int balance, int amount) {
        if (amount > balance) {
            throw new InsufficientFundsException("Withdrawal amount exceeds balance.");
        } else {
            System.out.println("Withdrawal successful.");
        }
    }
}
```

Rethrowing Exceptions:

Rethrowing exceptions in Java refers to the practice of **catching an exception** in a **catch** block and then **throwing it again**, allowing it to propagate up the call stack. This is often done after logging or modifying the exception.

Two Ways to Achieve:

1. **Rethrow without modification**
2. **Rethrow with modification**

1. **Rethrow without modification:** The caught exception is simply thrown again, allowing it to propagate unchanged.

Example:

```
try {
    // Code that might throw an exception
} catch (Exception e) {
    // Log or handle the exception
    throw e; // Rethrowing the original exception
}
```

2. Rethrow with modification: You can throw a new exception or wrap the original one in a new exception, which adds **more context** or a **custom message**.

Example:

```
try {
    // Code that might throw an exception
} catch (Exception e) {
    // Log or handle the exception
    throw new CustomException("Additional context", e);
}
```

Exception Specification:

- In Java, exception specification refers to the **declaration of exceptions that a method might throw**.
- This is done using the **throws** keyword in the **method signature**, which informs the **caller** of the method about the exceptions that could be thrown and that the caller must **handle** or **propagate** these exceptions.

Types of Exceptions in Java:

1. Checked Exceptions:

- These are exceptions that are checked at **compile-time.**
- A method must either handle these exceptions using a **try-catch block** or **specify them in the method signature using throws**.
- **Example**: IOException, SQLException.

2. Unchecked Exceptions: (Exception Specification Not Required)

- These are exceptions that are not checked at compile-time, usually derived from RuntimeException.
- Methods are **not required to specify unchecked exceptions** in their throws clause as they are **propagated automatically.**
- **Example**: NullPointerException, ArithmeticException.

Syntax of Exception Specification:

A method can specify the exceptions it throws using the **throws** keyword.

Syntax:

```
public void someMethod() throws IOException, SQLException {
    // Code that might throw these exceptions
}
```

Example: In this example:

- The **readFile** method **specifies** that it **might throw** an **IOException**.
- The process method catches and handles the IOException.

```
public class FileProcessor {
    public void readFile(String filePath) throws IOException {
        // Code that might throw an IOException
        FileReader reader = new FileReader(filePath);
    }

    public void process() {
        try {
            readFile("example.txt");
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

PRAMO

Benefits of exception handling

Exception handling in Java provides several important benefits, making applications more robust, maintainable, and reliable. Here are the key benefits:

1. Improved Program Flow

- **Prevents Crashes**: By catching exceptions, you can handle errors gracefully without **crashing** the entire program. This enables the program to **continue executing** after encountering an exception.
- **Separation of Error-Handling Code**: Exception handling separates normal code from **error-handling code**, making the code cleaner and easier to maintain.

2. Grouped Error Handling:

Catching Multiple Exceptions: A single try-catch block can handle multiple types of exceptions. You can group related exceptions to be handled in one place, reducing redundancy and simplifying error handling.

Example:

```
try {
    // Code that may throw multiple exceptions
} catch (IOException | SQLException e) {
    // Handle both IOException and SQLException here
}
```

3. Cleaner Code Using finally Block:

- **Guaranteed Cleanup:** The finally block ensures that resources such as file streams, database connections, or network sockets are properly closed, regardless of whether an exception occurs. This prevents resource leaks and ensures that important cleanup tasks are always performed.

```
try {
    // Open resource
} finally {
    // Close resource
}
```

4. Custom Exceptions:

Create User-Defined Exceptions: Java allows you to create custom exceptions by extending the Exception class. This feature enables you to throw and catch meaningful exceptions specific to your application's domain (e.g., InvalidUserInputException, TransactionFailedException).

```
class InvalidUserInputException extends Exception {
    public InvalidUserInputException(String message) {
        super(message);
    }
}
```

PRAMOD NAIK

5. Code Maintainability

Modular and Readable Code: Exception handling makes your code modular, meaning you can focus on the core functionality in one section of the code and handle errors in another section. This leads to cleaner, more readable, and maintainable code.

6. Helps in Debugging

Exception Stack Traces: When an exception is thrown, Java provides a detailed stack trace that shows exactly where the error occurred and what methods were involved. This information can be crucial for debugging and identifying the root cause of issues.

7. Supports Legacy Code Integration

Handle Legacy Code Failures: Exception handling allows smooth integration with older, error-prone code by providing a structured way to manage its errors without disrupting modern applications.