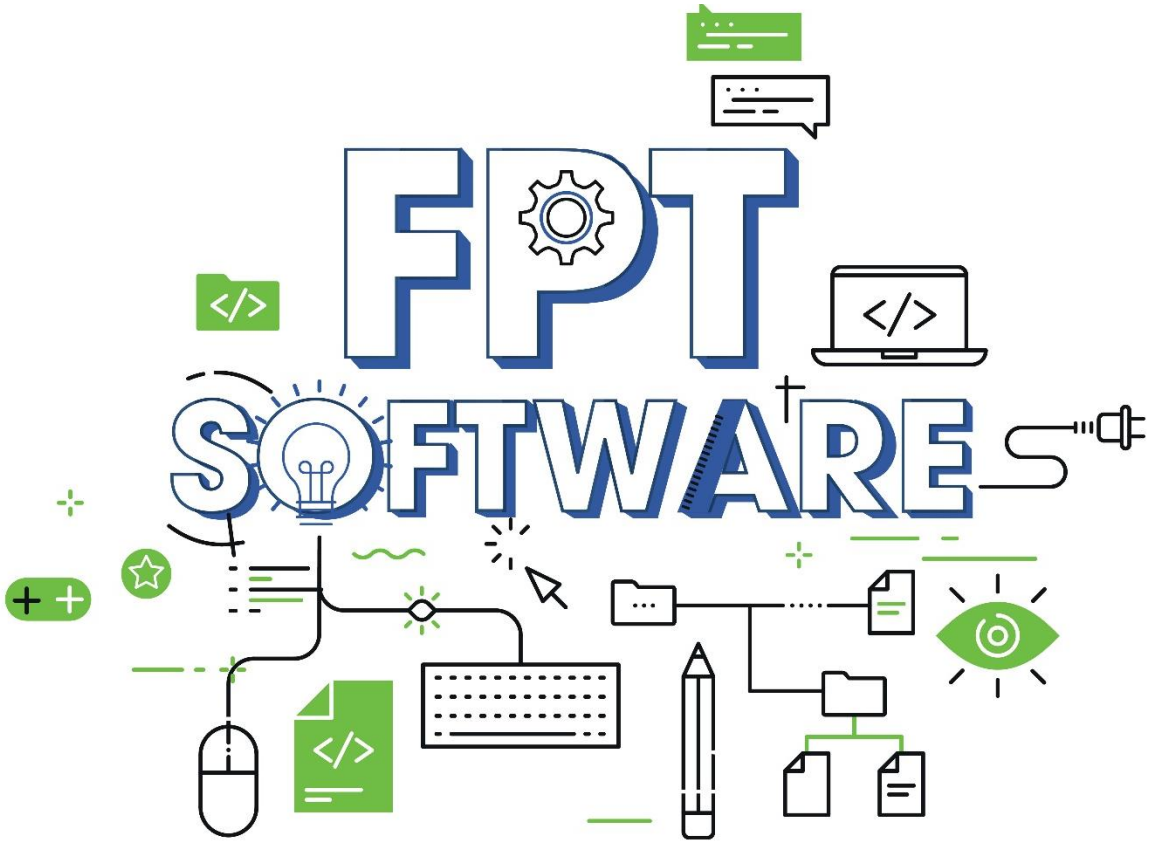
Java Exception Handling



Design by: DieuNT1

Agenda

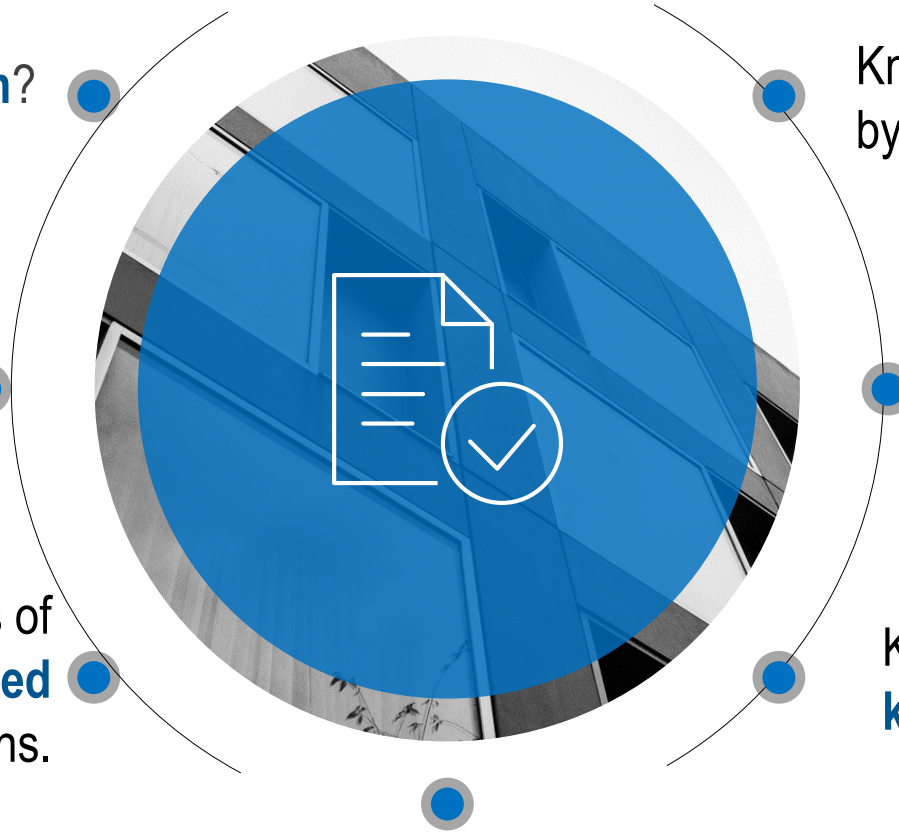
1. Exceptions in Java
2. Exception Handling
3. Checked vs Unchecked Exceptions
4. throw and throws in Java
5. Question and Answer

Lesson Objectives

Understanding **what is exception**?

Knowing how to handle exceptions using **try-catch blocks**.

Understanding the different types of exceptions: **checked** and **unchecked** exceptions.



Knowing how to create **custom exceptions** by extending the Exception class.

Understanding how to use the **finally block**

Knowing how to use the **throw** and **throws keywords**

Understanding **best practices** for exception handling

Section 1

Exceptions in Java

Exceptions 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.

- When an exception occurs within a method, it creates an **exception object**.
- Exception object contains **information about the exception**:
  - ✓ The ***name*** and ***description*** of the exception
  - ✓ the ***state*** of the program when the exception occurred.

Errors

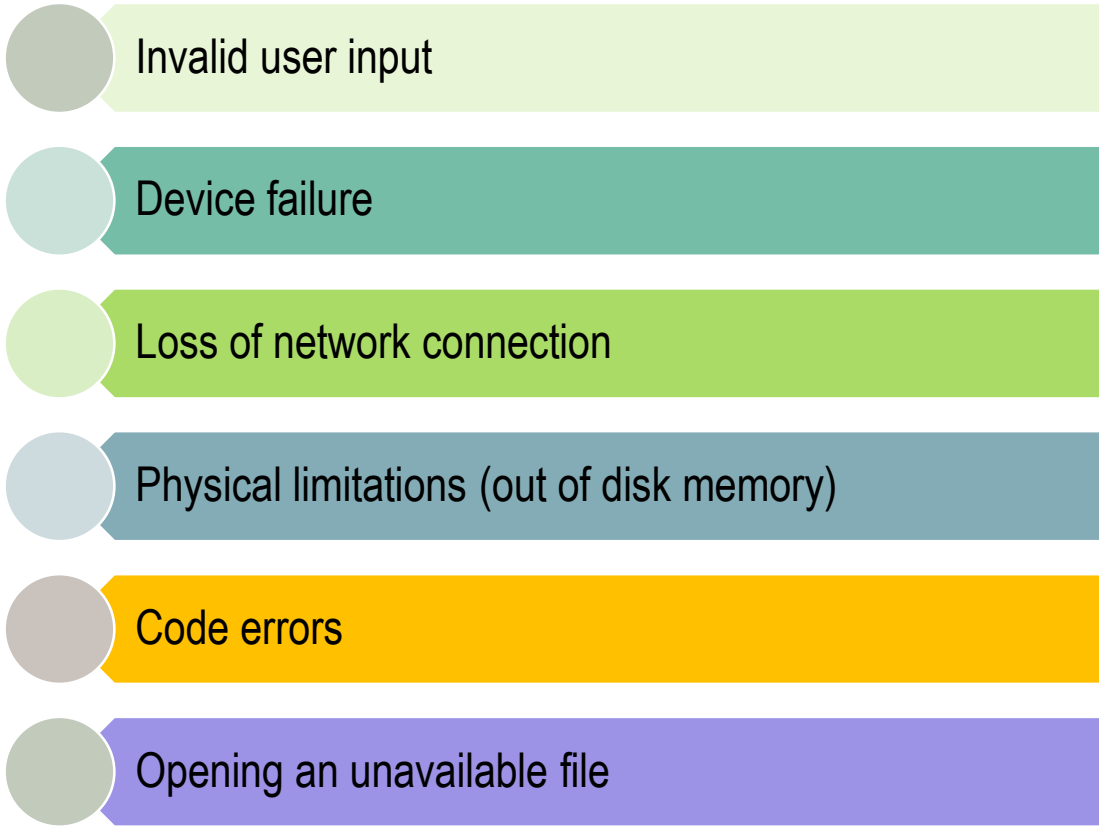
Errors represent ***irrecoverable conditions*** such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc.

- Errors are usually **beyond the control of the programmer**, and we should not try to handle errors.

Exception	Error
• Exception indicates conditions that a reasonable application might try to catch.	• An Error indicates a serious problem that a reasonable application should not try to catch.

Why an exception Occurs?

- **Major reasons why an exception Occurs**



Exception Handling

Exception Handling in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved.

- **Java Exception Handling** is a mechanism to handle runtime errors such as:
  - ✓ ClassNotFoundException,
  - ✓ IOException,
  - ✓ SQLException,
  - ✓ RemoteException, etc.

Example 1

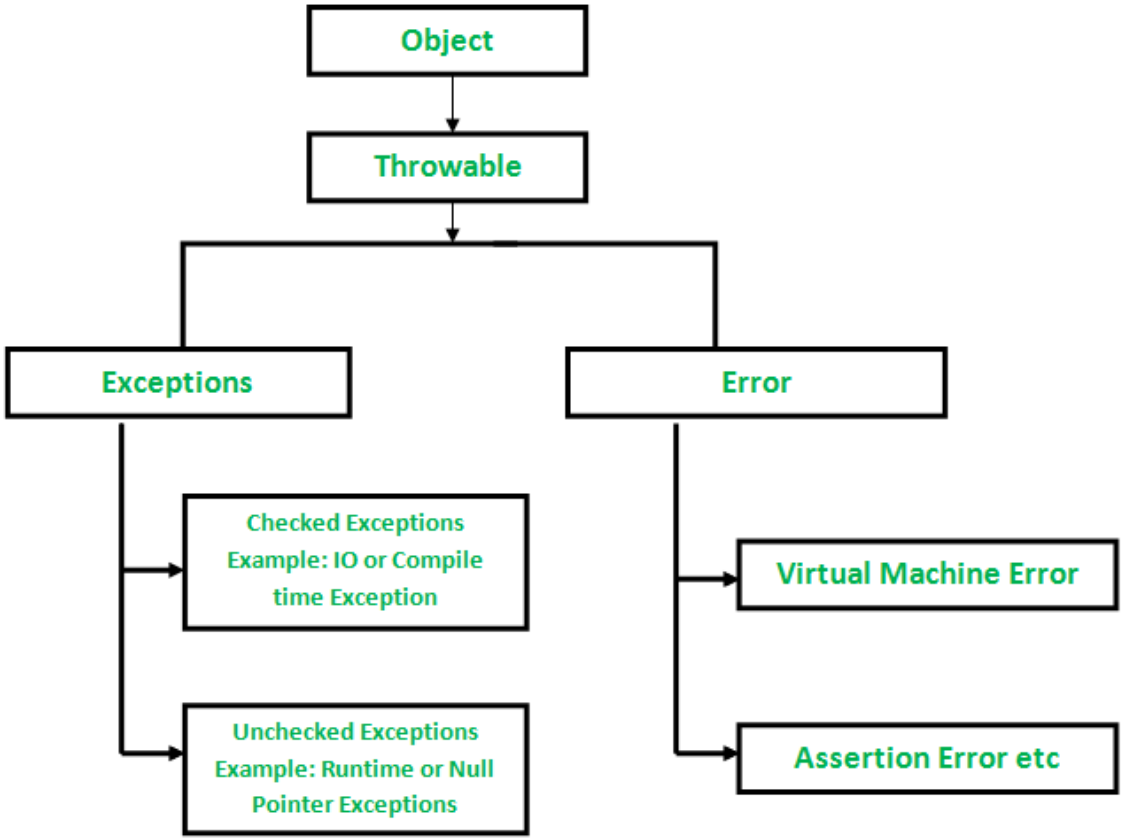
- A program that requests a number from the user:

```
public class Exercise {
    public static void main(String[] args) {
        double side;
        Scanner scnr = new Scanner(System.in);
        System.out.println("Square Processing");
        System.out.print("Enter Side: ");
        side = scnr.nextDouble();
        System.out.println("\nSquare Characteristics");
        System.out.printf("Side: %.2f\n", side);
        System.out.printf("Perimeter: %.2f\n", side * 4);
    }
}
```

```
Square Processing
Enter Side: dS
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextDouble(Unknown Source)
        at clc.btjb.unit04.les04.Exercise.main(Exercise.java:11)
```

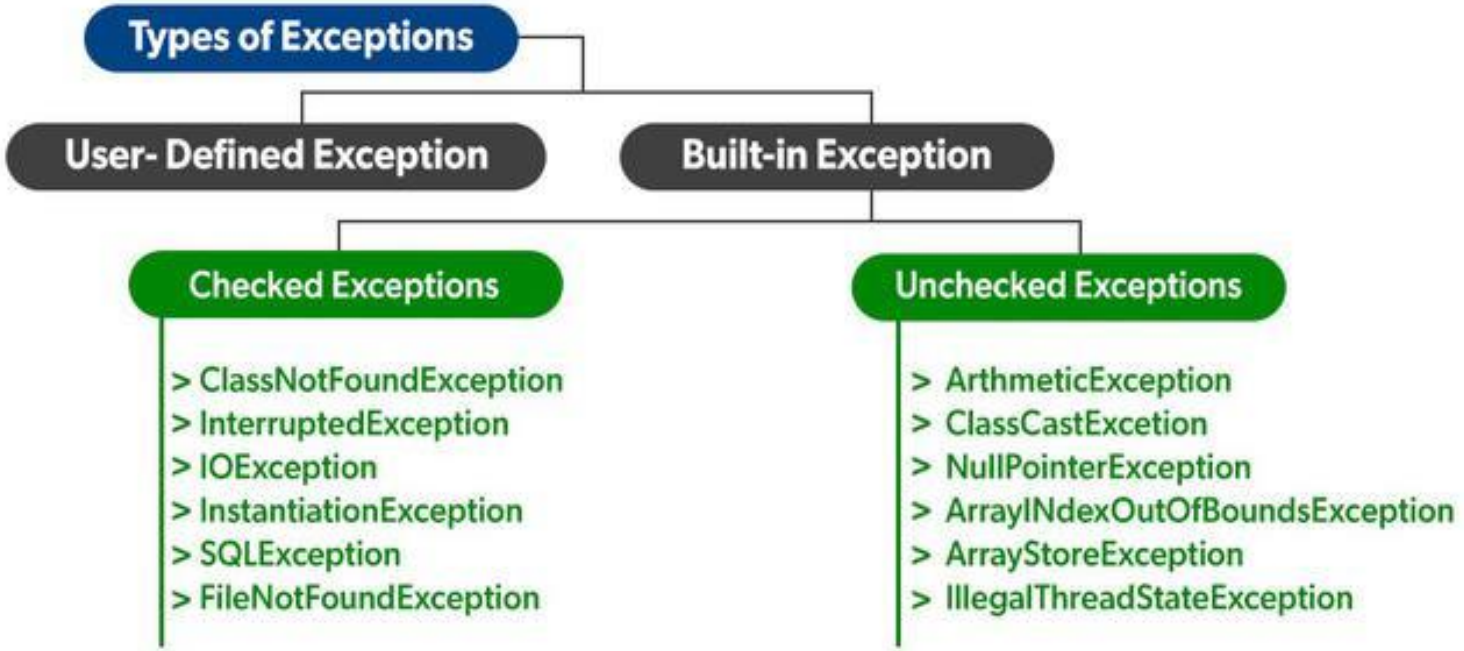
Hierarchy of Java Exception classes

- The **java.lang.Throwable** class is the root class of Java Exception hierarchy which is inherited by two subclasses:
  - ✓ Exception and
  - ✓ Error.



Types of Exceptions

- **Built-in Exception**: Java defines **several types of exceptions** that relate to its various class libraries.
- **User-Defined Exception**: Java also allows **users to define their own exceptio**



Types of Exceptions

- **Exceptions can be categorized in two ways:**

Built-in Exceptions	User-Defined Exceptions
• Checked Exception • Unchecked Exception	• Users to define their own exceptions.

Types of Exceptions

Built-in Exceptions

- Built-in exceptions are the exceptions that are available in Java libraries.

Checked Exception

✓ **Compile-time exceptions**

(these exceptions are checked at compile-time by the compiler)

VS

Unchecked Exception

✓ **Runtime exceptions**

(The compiler will not check these exceptions at compile time.)

Section 2

Exception Handling

Exception Handling

When an **exceptional condition arises**, an object representing that exception is created and thrown in the method that caused the error.

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

1. **try:** The try block contains a set of statements where an exception can occur.

```
try{
// statement(s) that might cause exception
}
```

Exception Handling

2. **catch:** The catch block is used to handle the uncertain condition of a try block.

```
catch (Exception_Class_Name ref) {
    // statement(s) that handle an exception
    // examples, closing a connection, closing
    // file, exiting the process after writing
    // details to a log file.
}
```

3. **throw**: The throw keyword is used to transfer control from the try block to the catch block.

Exception Handling

4. **throws**: The throws keyword is used for exception handling without try & catch block. It specifies **the exceptions that a method can throw to the caller** and does not handle itself.

5. **finally**: It is executed after the catch block.

We use it to put some common code (**to be executed irrespective of whether an exception has occurred or not**) when there are multiple catch blocks.

Exception Handling

- Syntax of Java try-catch

```
try {
        //code that may throw an exception
} catch(Exception_Class_Name ref){}
```

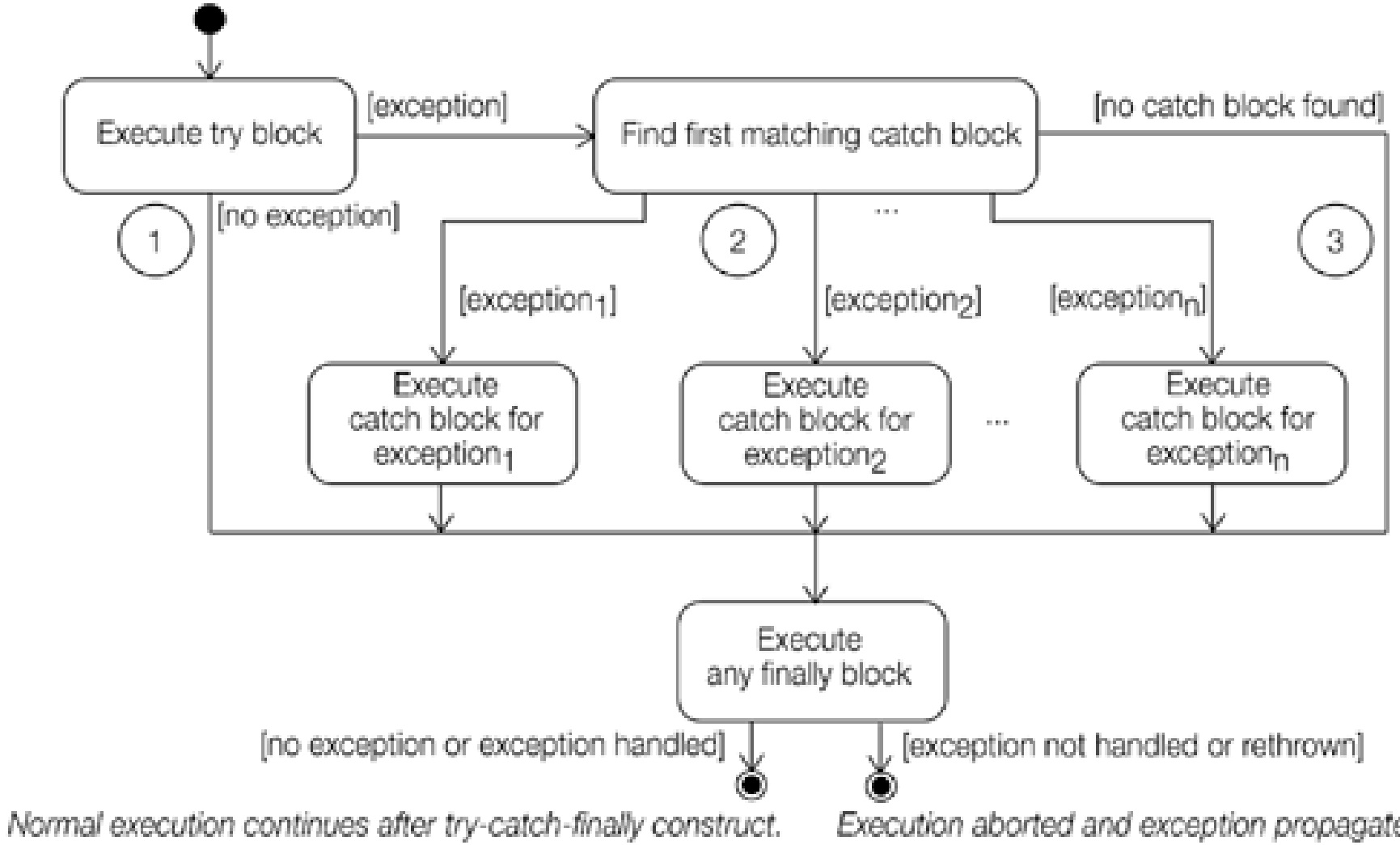
- Syntax of try-finally block

```
try {
        //code that may throw an exception
} finally {}
```

- Syntax of try-catch-finally block

```
try {
        //code that may throw an exception
} catch(Exception_Class_Name ref){

} finally {}
```

Exception Handling



Solution for Example 1

```
try {
    side = scnr.nextDouble();
    System.out.println("\nSquare Characteristics");
    System.out.printf("Side: %.2f\n", side);
    System.out.printf("Perimeter: %.2f\n", side * 4);
}
catch(InputMismatchException ex) {
  System.out.println(ex.getMessage());
  // method get message
}
```

Multi-catch block

- A try block can be followed by **one or more catch blocks.**
- Each catch block must contain a **different exception handler**.
- You have to **perform different tasks** at the occurrence of different exceptions, use java multi-catch block.

```
try {
        // block of code to monitor for errors
} catch (ExceptionType1 exOb) {
        // exception handler for ExceptionType1
} catch (ExceptionType2 exOb) {
        // exception handler for ExceptionType2
}
// ...
```

Multi-catch block

- **Points to remember:**

Note 1

- At a time **only one exception occurs** and at a time **only one catch block is executed**.

Note 2

- All catch blocks **must be ordered from most specific to most general**, i.e. catch for ArithmeticException must come before catch for Exception.

Multi-catch block

- In this example, try block contains two exceptions.
- But at a time only one exception occurs and its corresponding catch block is invoked.

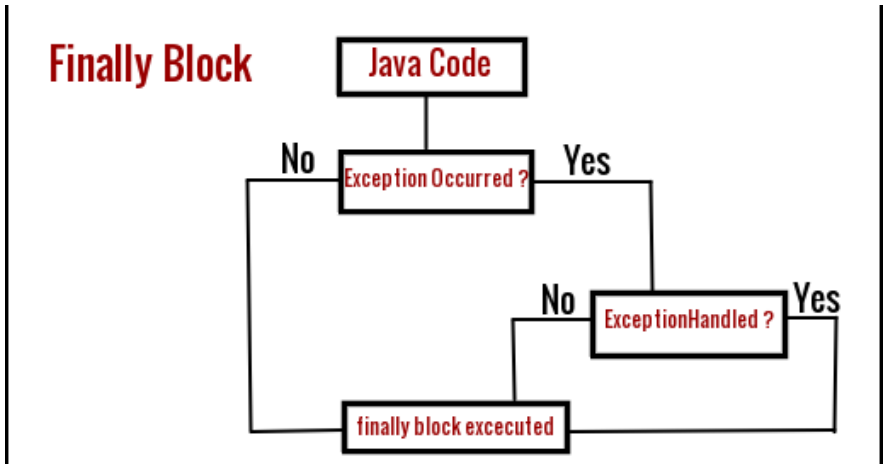
```
public class MultipleCatchBlock {
  public static void main(String[] args) {

    try {
      int a[] = new int[5];
      a[5] = 30 / 0;
      System.out.println(a[10]);

    } catch (ArithmeticException e) {
      System.out.println("Arithmetic Exception occurs");
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("ArrayIndexOutOfBounds Exception occurs");
    } catch (Exception e) {
      System.out.println("Parent Exception occurs");
    }
    System.out.println("Rest of the code");
  }
}
```

Output:

```
Arithmetic Exception occurs
Rest of the code
```

finally block

- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.



Example:

```
Connection conn= null;
try {
        conn= get the db conn;
        //do some DML/DDL
} catch(SQLException ex) {
} finally {
        conn.close();
}
```

Example 2

```
public class Division {
    public static void main(String[] args) {
        int a = 10, b = 5, c = 5, result;
        try {
                result = a / (b - c);
                System.out.println("result" + result);
        } catch (ArithmeticException e) {
                System.out.println("Exception caught:Division by zero");
        } finally {
                System.out.println("I am in final block");
        }
    }
}
```

Output:

```
Exception caught:Division by zero

I am in final block
```

Difference between final, finally and finalize

final	finally	finalize
✓ Final is used to apply restrictions on class, method and variable ✓ Final class can't be inherited ✓ Final method can't be overridden ✓ Final variable value can't be changed.	✓ Finally is used to place important code, it will be executed whether exception is handled or not.	✓ Finalize is used to perform clean up processing just before object is garbage collected.
final is a **keyword**.	finally is a **block**.	finalize is a **method**.

Section 3

Checked vs Unchecked Exceptions

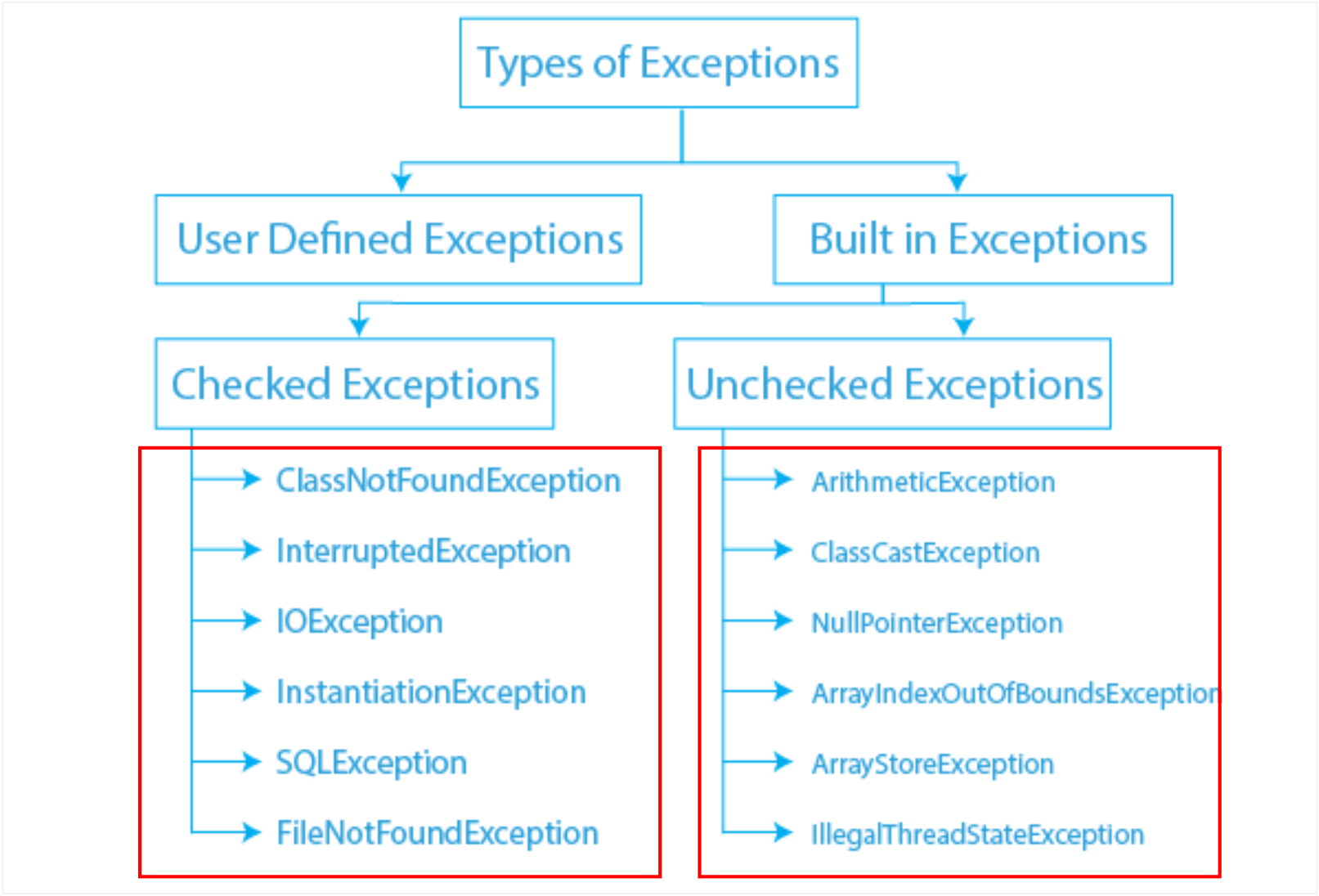
Checked vs Unchecked exceptions

- **Checked exceptions**
  - ✓ These are the exceptions that are **checked at compile time**.
  - ✓ The method must either handle the exception or it must specify the exception using the *throws* keyword.
- **Checked exceptions** have two types:

fully checked	partially checked

• A fully checked exception is a checked exception where **all its child classes are also checked:**
  • IOException, and
  • InterruptedException
  • …
	• A partially checked exception is a checked exception where **some of its child classes are unchecked**, like an Exception.

Checked vs Unchecked exceptions



Checked Exception: Example

>> try-catch

Output:

```
First three lines of file data.txt:
-----------------------------------
Welcome to FPT Software Academy!
Java Programming Language
Hello World
```

```java
public class CheckedException {
    public static void main(String args[]) {
        BufferedReader fileInput = null;
        try {
            // Creating a file and reading from local repository
            FileReader file = new FileReader("data.txt");
            // Reading content inside a file
            fileInput = new BufferedReader(file);
            // Printing first 3 lines of file "data.txt"
            System.out.println("First three lines of file data.txt:");
            for (int counter = 0; counter < 3; counter++)
                System.out.println(fileInput.readLine());
        } catch (IOException exception) {
                exception.printStackTrace();
        } finally {
        // Closing all file connections using close() method
        // Good practice to avoid any memory leakage
            if (fileInput != null) {
                try {
                        fileInput.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
            }
        }
    }
}
```

Checked Exception: Example

>> throws

```
public class CheckedException {
     public static void main(String args[]) throws IOException {
          BufferedReader fileInput = null;
          try {
               // Creating a file and reading from local repository
               FileReader file = new FileReader("data.txt");
               // Reading content inside a file
               fileInput = new BufferedReader(file);
               // Printing first 3 lines of file "data.txt"
               System.out.println("First three lines of file data.txt:");
               System.out.println("-----------------------------------");
               for (int counter = 0; counter < 3; counter++)
               System.out.println(fileInput.readLine());
          } finally {
               // Closing all file connections using close() method
               // Good practice to avoid any memory leakage
               if (fileInput != null) {
                    fileInput.close();
               }
          }
     }
}
```

Unchecked Exceptions

- **Runtime Exceptions** are also known as Unchecked Exceptions as the **compiler do not check whether** the programmer has handled them or not but it's the duty of the programmer to handle these exceptions and provide a safe exit.
- If these exceptions are not handled/declared in the program**, it will not give compilation error**.
- **Examples of UnChecked Exceptions:**
  - `ArithmeticException`
  - `ArrayIndexOutOfBoundsException`
  - `NullPointerException`
  - `NegativeArraySizeException, etc.`

Unchecked Exceptions: Example

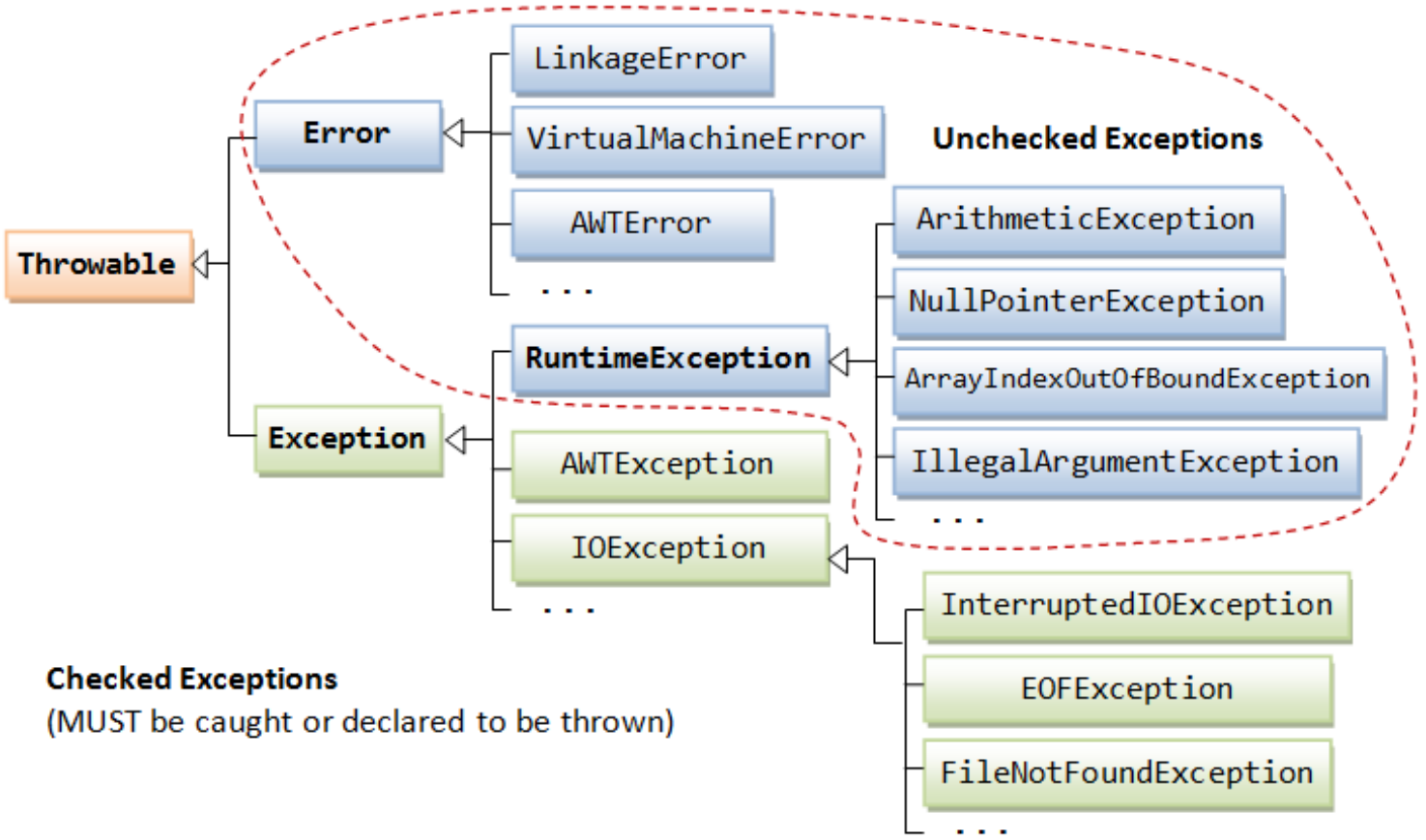
```
public class ArrayListSample {
     public static void main(String[] args) {
          List<String> list = new ArrayList<>();

          list.add("Apple");
          list.add("Samsung");
          list.add(1, "Oppo");
          // Inserts the specified element at index is 5
          list.add(5, "Nokia");
          // Get size of list
          int size = list.size();
          // Get element in list
          for (int i = 0; i < size; i++) {
                    System.out.print(list.get(i) + "\t");
          }
     }
}
```

Output:

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 5, Size: 3
at java.base/java.util.ArrayList.rangeCheckForAdd(ArrayList.java:756)
at java.base/java.util.ArrayList.add(ArrayList.java:481)
at fa.training.list.ArrayListSample.main(ArrayListSample.java:20)
```

Exception classes

- The figure below shows the hierarchy of the Exception classes.



Section 4

throw and throws in Java

throw keyword

- The Java throw keyword is used to **explicitly throw an exception.**
- We can throw either **checked** or **uncheked** exception in java by throw keyword.
- The throw keyword is mainly used to throw custom exception.
- **Syntax**:

```
throw exception;
```

```
throw new IOException("sorry device error);
```

throw keyword: Example

```
public class ThrowExcep {
    static void fun() {
        try {
                throw new NullPointerException("demo");
        } catch (NullPointerException e) {
            System.out.println("Caught inside fun().");
            // rethrowing the exception
            throw e;
        }
    }
    public static void main(String args[]) {
        try {
                fun();
        } catch (NullPointerException e) {
                System.out.println("Caught in main.");
        }
    }
}
```

Output:

```
Caught inside fun().
Caught in main.
```

throw keyword

- The Java throw keyword is used to **explicitly throw an exception.**
- We can throw either **checked** or **uncheked** exception in java by throw keyword.
- The throw keyword is mainly used to throw custom exception.
- **Syntax**:

```
throw exception;
```

```
throw new IOException("sorry device error);
```

- **Example**:

```
public static void isAge(int age) {
   if (age < 18) {
     throw new ArithmeticException("Not valid");
   } else {
     System.out.println("welcome to vote");
   }
 }
```

throw keyword

```
public class Main {

  public static void main(String[] args) {
    try {
      Validator.isAge(15);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Output:

```
java.lang.ArithmeticException: Not valid
at fa.training.utils.Validator.isAge(Validator.java:20)
at fa.training.jpe2.Main.main(Main.java:9)
```

throws keyword

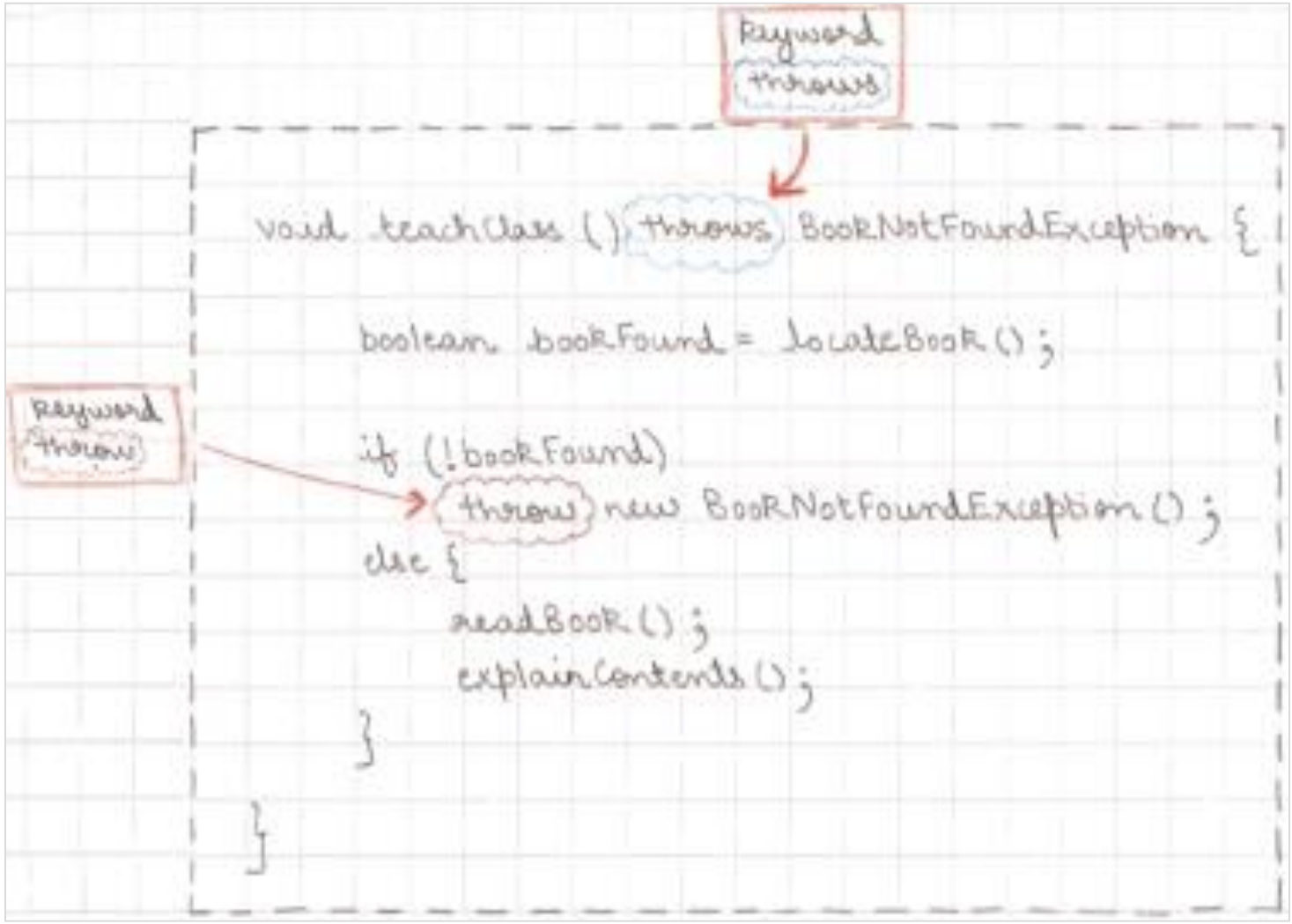
throws is a keyword in Java which is used in the signature of method to indicate that **this method might throw one of the listed type exceptions**.

The **caller to these methods has to handle the exception** using a try-catch block.

- Exception Handling is mainly used to handle the **checked exceptions**.
- **Syntax**:

```
return_type method_name() throws Exception_Class_Name {
//method code
}
```

Using throws and throw



Using throws and throw

- Enclose the code within a try block and catch the thrown exception:

```
void callTeachClass() {
    try {
        teachClass();
    } catch (BookNotFoundException e) {
        // code
    }
}
```

- Declare the exception to be rethrown in the method's signature:

```
void callTeachClass() throws BookNotFoundException {
        teachClass();
}
```

Using throws and throw

- Implement both the above together:

```
void callTeachClass() throws BookNotFoundException {
    try {
        teachClass();
    } catch (BookNotFoundException e) {
        // code
    }
}
```

Difference between throw and throws

throw	throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
throw is followed by an instance.	throws is followed by class.
throw is used within the method.	throws is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Summary

- Exceptions in Java
- Exception Handling
- Checked And Unchecked Exception
- throw and throws keywords
- Question and Answer

Questions

THANK YOU!