OBJECT ORIENTED TECHNIQUES USING JAVA(ACSE0302)

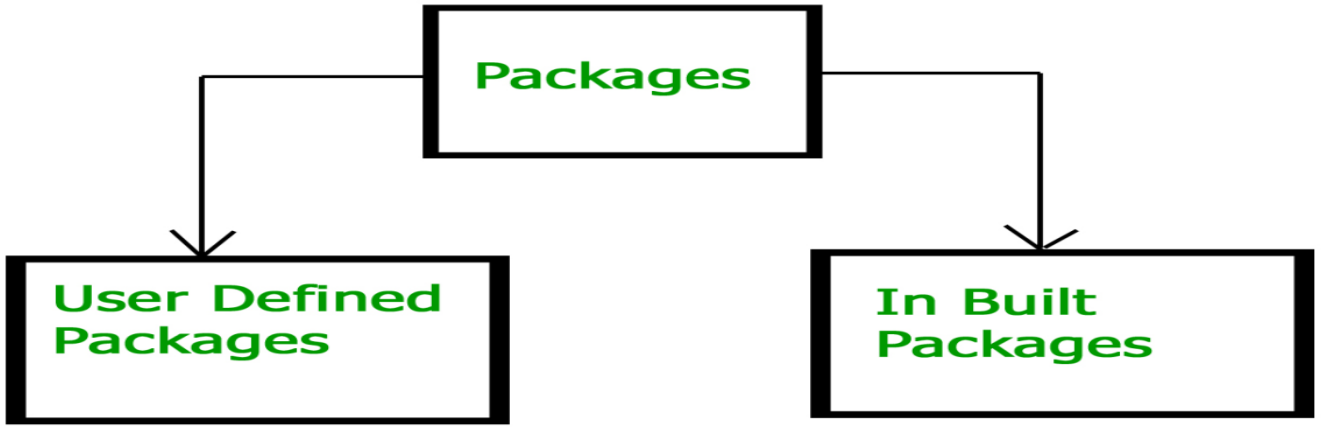
Unit: 3

Packages, Exception Handling and String Handling

Course Details
(B.Tech 3rd Sem /2nd Year)

Dr. Mohammad Shahid
Associate Professor
CSE Department

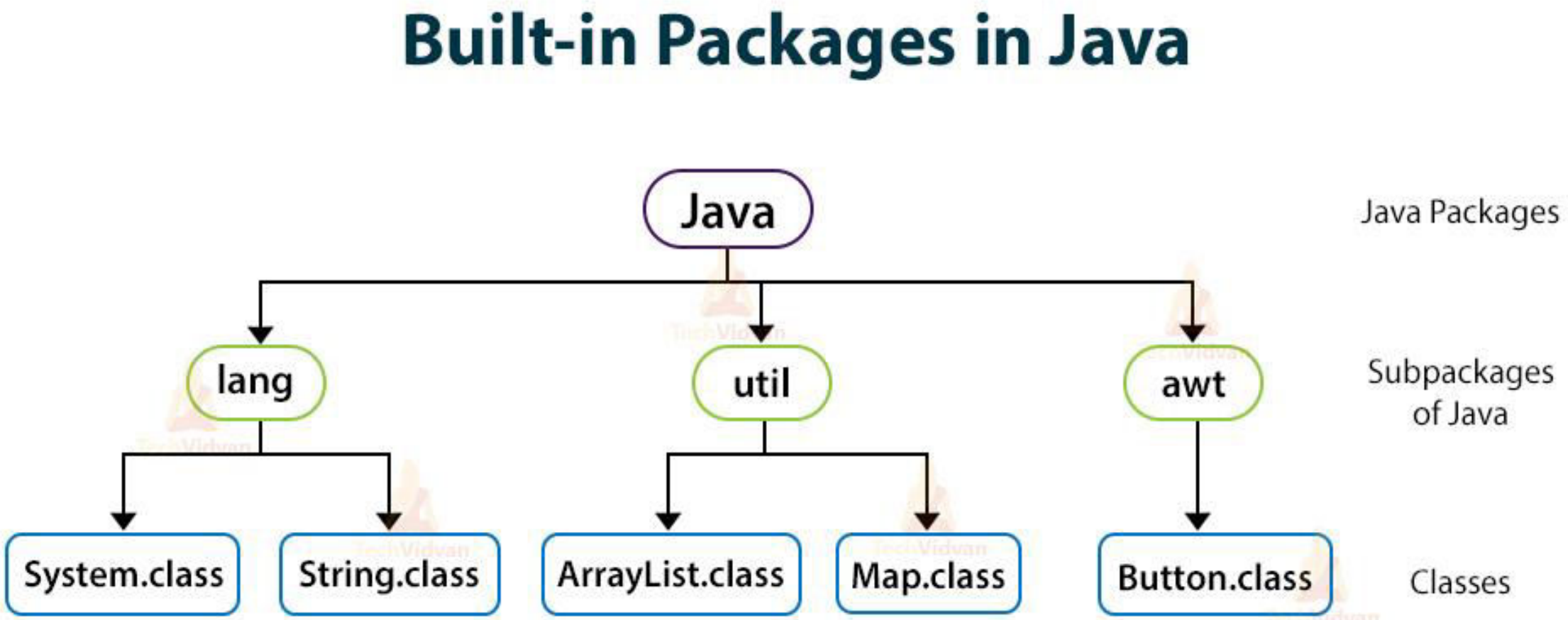
Types of Packages



1. User-defined Package

- The package which is defined by the user is called a User-defined package.
- It contains user-defined classes, interfaces and subpackages.
- We create a directory whose name should be the same as the name of the package. Then we create a class inside the directory.

Types of Packages



Types of Packages

2. Java API packages or built-in packages

Java provides a large number of classes grouped into different packages based on a particular functionality.

Examples:

- **java.lang:** It contains classes for primitive types, strings, math functions, threads, and exceptions.
- **java.util:** It contains classes such as vectors, hash tables, dates, Calendars, etc.
- **java.io:** It has stream classes for Input/Output.
- **java.awt:** Classes for implementing Graphical User Interface – windows, buttons, menus, etc.
- **java.net:** Classes for networking
- **java. Applet:** Classes for creating and implementing applets

Example of Java Package

- We can create a Java class inside a package using a **package** keyword.

```
package mypackage //package

class Example

{
    public static void main(String args[])
    {
        System.out.println("Welcome to OOT using Java Class");
    }
}
```

Command to create package:

c:\> javac –d . Example.java

Output:

Welcome to OOT using Java Class

How do Packages in Java work?

Access protection in java packages

- Package is a container of classes, sub-classes, interfaces, and sub-packages.
- Accessibility of the members of a class or interface depends on its access specifiers.
- Java has four access modifiers,
  1. **Default**
  2. **private**
  3. **protected**
  4. **public**

NIET Greater Noida
GET FUTURE READY
Estd.2001

Access control in java Packages

Accessibility / Access Specifier	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

Exceptions in Java

- It is a condition that is caused by a run-time error in the program.
- When the java interpreter encounters an error it create an exception object and throws it.(i.e. inform us that an error has occurred).

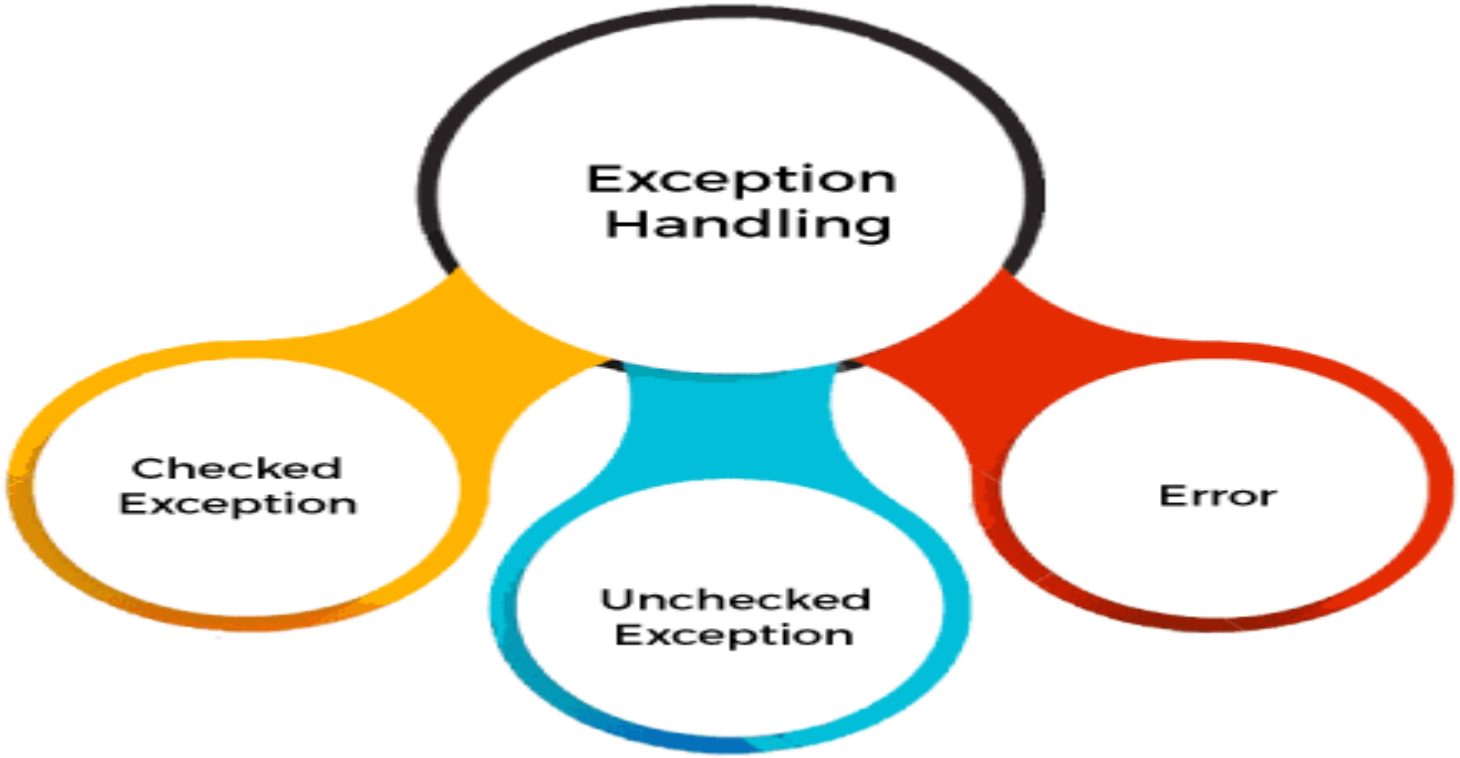
EXCEPTION HANDLING

- If the exception object is not caught and handled properly, the java interpreter will display a error message and will terminate the program.
- So we try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective action. This task is known as exception handling.

Types of Java Exceptions

- There are mainly **two** types of exceptions: checked and unchecked.
- An error is considered as the unchecked exception.
- However, according to Oracle, there are **three** types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



Difference Between Checked and Unchecked Exceptions

1. Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2. Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions.

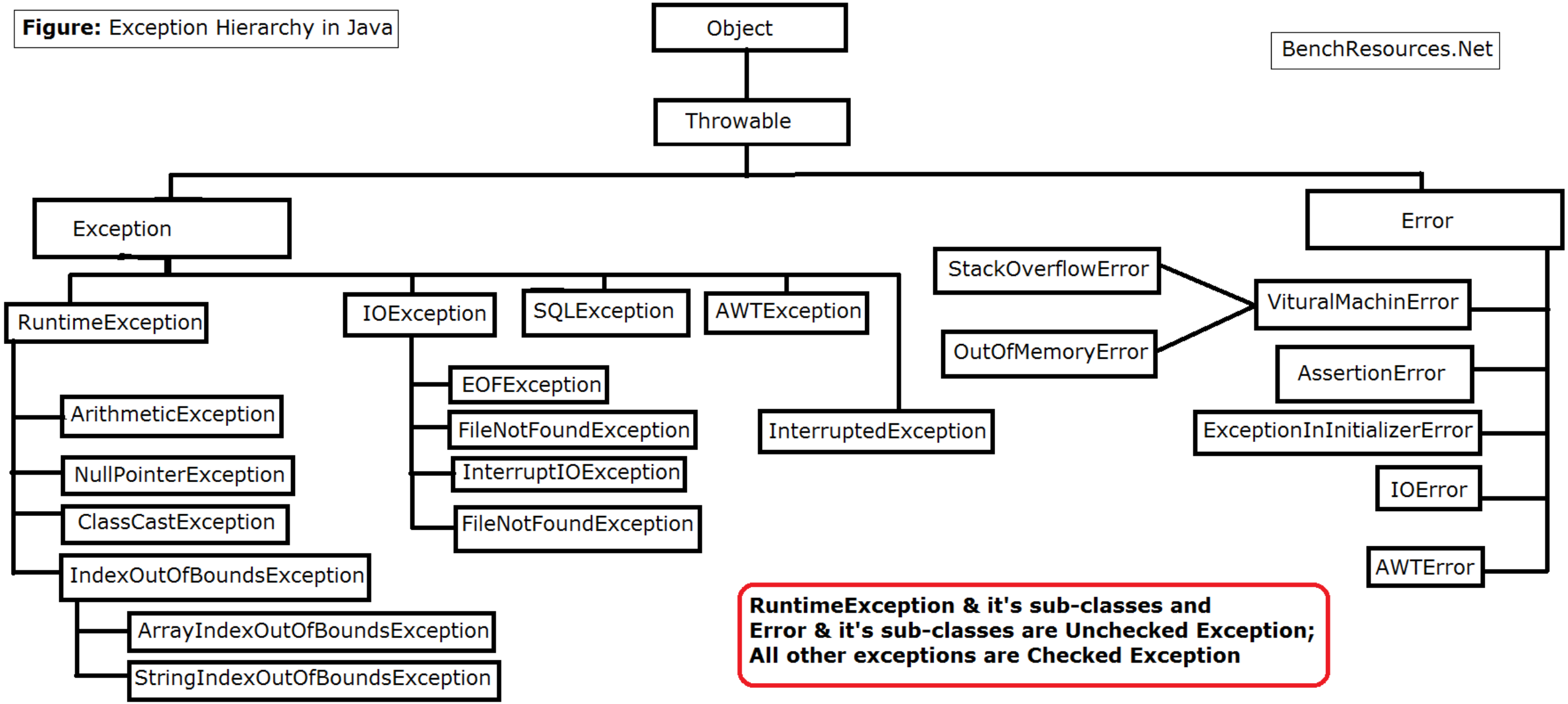
For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3. Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError etc.

Hierarchy of Java Exception Classes

Figure: Exception Hierarchy in Java



NIET Greater Noida
Estd.2001 GET FUTURE READY
AN AUTONOMOUS INSTITUTE

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Different structure of try-catch-finally block

```
try{
    ----
  }
catch(e){
        ------
  }
```

```
try{
    ----
  }
catch(e){
        ------
  }
finally{
    -------
  }
```

```
try{
    ----
  }
catch(e){
        ------
  }
catch(e){
        -----
  }
catch(e){
        ----
  }
```

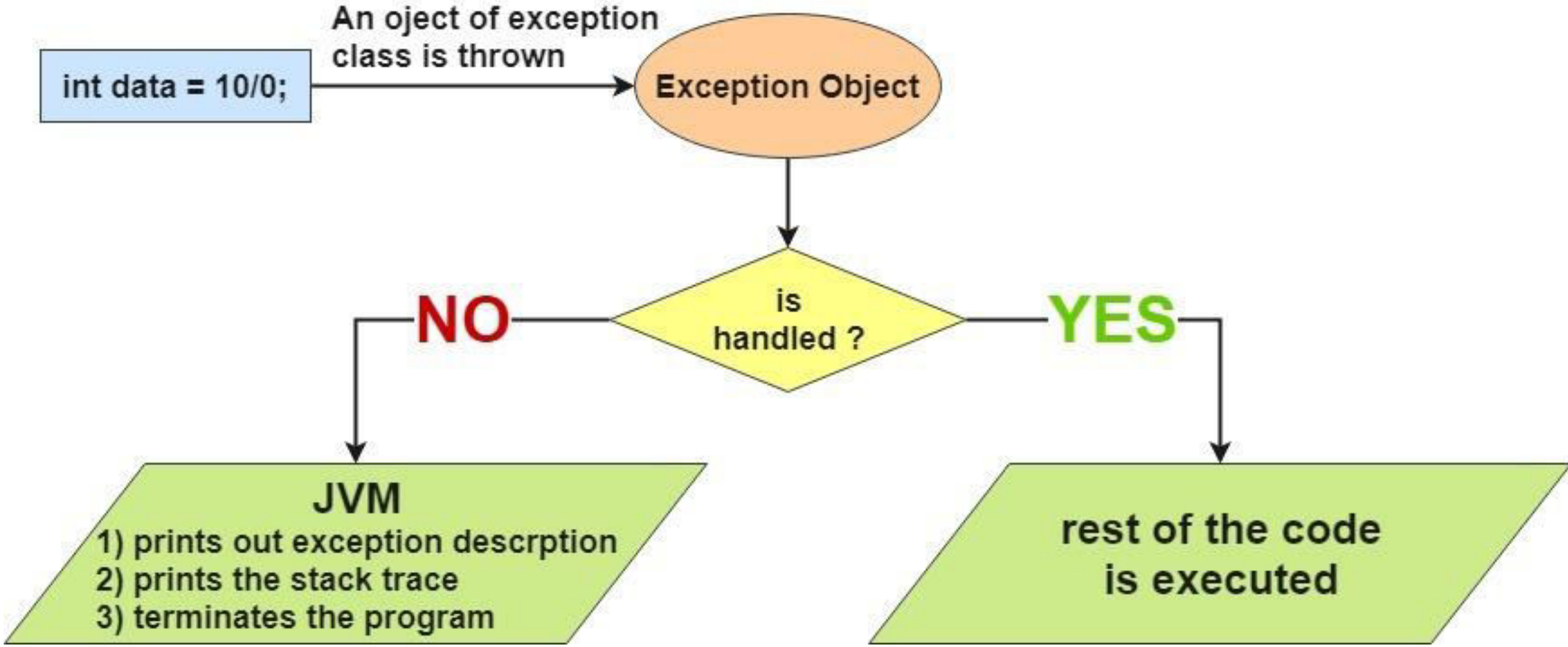
```
try{
    ----
  }
catch(e){
  }
catch(e){
  }
catch(e){
  }
finally{
    -------
  }
```

```
try{
    ----
  }
finally{
    -------
  }
```

try-catch block

- Java **try** block is used to enclose the code that might throw an exception.
- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception. Java try block must be followed by either catch or finally block.
- Java **catch block** is used to handle the Exception by declaring the type of exception within the parameter.
- The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal working of Java try-catch block



Internal working of Java try-catch block

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.
- But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

try-catch block

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;
  try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero
error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Multiple catch block

```
// Demonstrate multiple catch statements.
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }

    System.out.println("After try/catch blocks.");
  }
}
```

Output

```
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
a = 1
Array index oob:
java.lang.ArrayIndexOutOfBoundsException:42
 After try/catch blocks
```

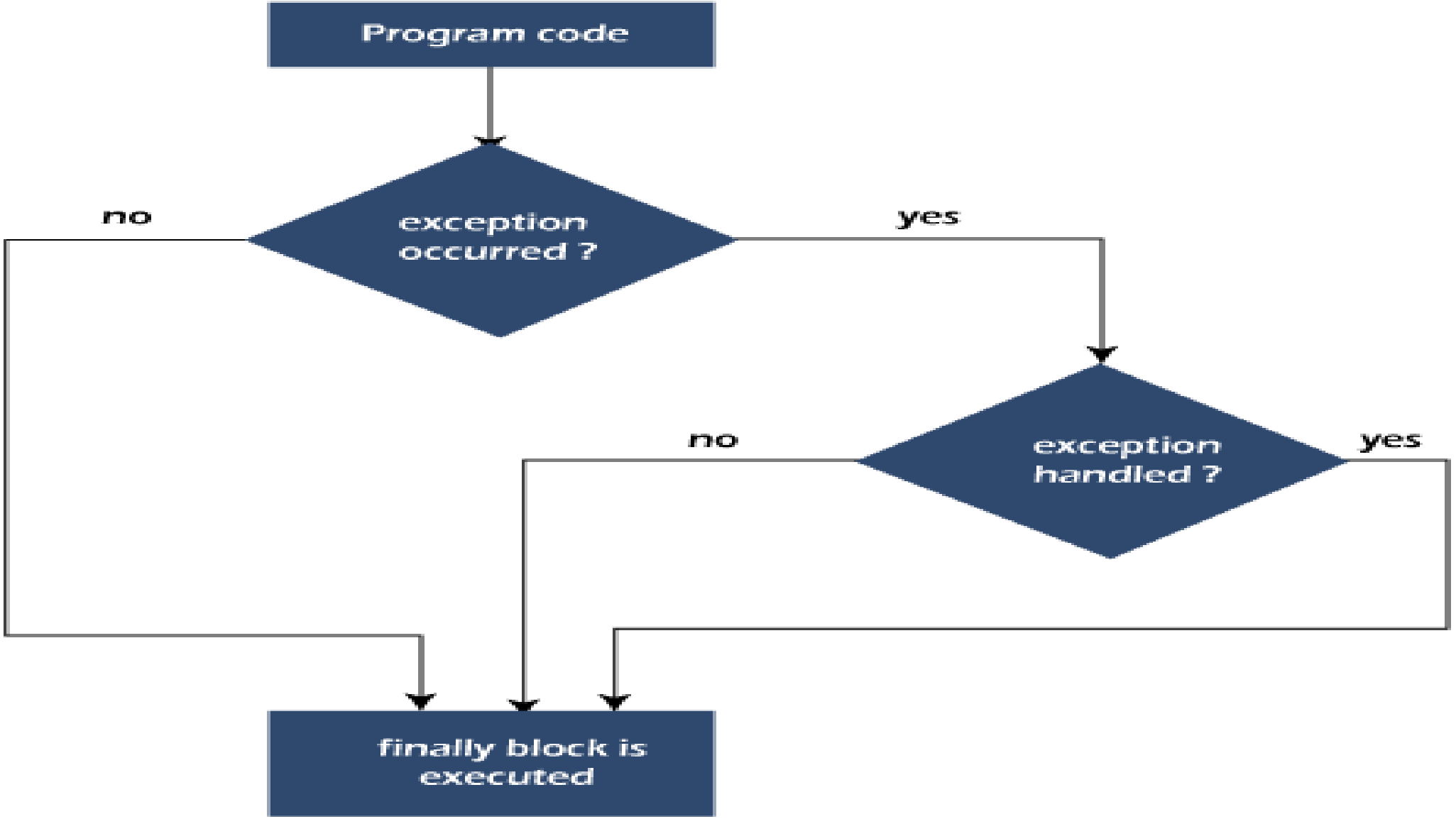
Finally block

- **finally block** is used to execute important code such as closing a file, closing the database connection, etc.

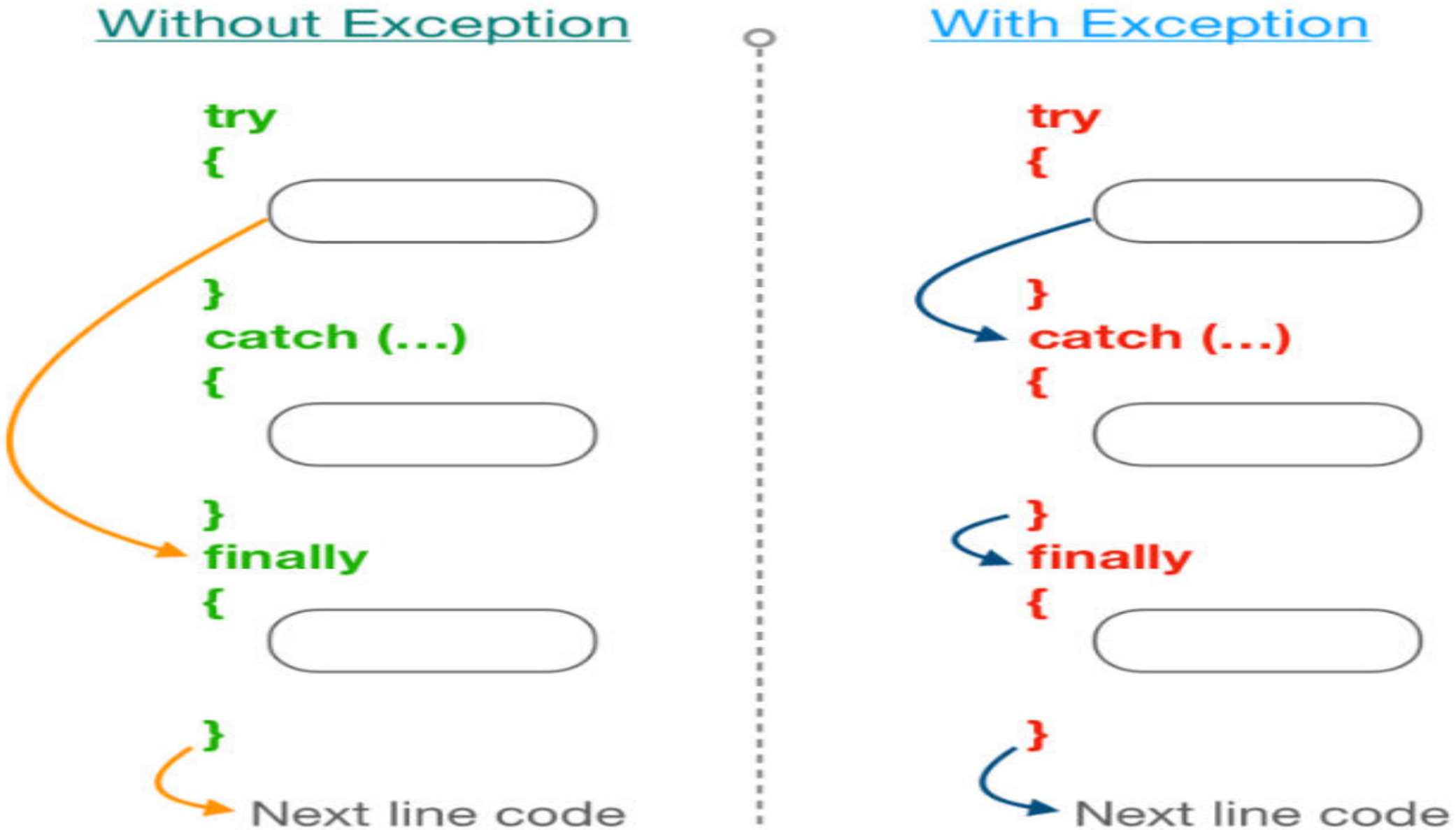
- It is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

- The finally block follows the try-catch block.

Flowchart of finally block



try-catch finally block



try-catch finally block

```
Class  Ex20{
    Public static void main(String str[]){
   Try{
        int x=20/0;
       }
    Catch(ArithmeticException e){
      System.out.println(e);
      }
    Finally{
       System.out.println("I am always here");
            }}}
```

Output:

ArithmeticException

I am always here

Nested try block

- Using a try block inside another try block is called as nested try block.
- Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.
- For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

Why use nested try block?

- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Nested try-catch block

```
class NestTry {
 public static void main(String args[]) {
  try {
        int a = args.length;
        int b = 42 / a;
        System.out.println("a = " + a);
        try {
            if(a==1) a = a/(a-a); // division by zero
            if(a==2) { int c[] = { 1 };
                       c[42] = 99; }// out-of-bounds
            }
        catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds:"+e);}
       }
   catch(ArithmeticException e) {
    System.out.println("Divide by 0: " + e);}
  }
```

Nested try-catch block

Output

```
C:\\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

C:\\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

C:\\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException: 42
```

Throw Keyword

- The Java throw keyword is used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception(user defined exception).

The syntax of the Java throw keyword is given below.

```
throw new exception_class("error message");
```

Throw Keyword

```
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // rethrow the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);

    }
  }
}
```

Output:

Caught inside demoproc. Recaught:

java.lang.NullPointerException: demo

Throws Keyword

- If a method is capable of causing an exception that it does not handle, we can do this by including a throws clause in the method's declaration.
- A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

Syntax of Java throws

type method-name(parameter-list) **throws** exception-list {

// body of method }

Throws Keyword

```
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

Custom Exception

- Although Java's built-in exceptions handle most common errors, we can create our own exception types to handle situations specific to our applications.
- for this we define a subclass of Exception (which is, of course, a subclass of Throwable).
- The Exception class does not define any methods of its own. It inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable .
- We may also wish to override one or more of these methods in exception classes that you create.

Custom Exception

```
class MyException extends Exception {
   private int detail;

   MyException(int a) {
      detail = a;
   }

   public String toString() {
      return "MyException[" + detail + "]";
   }
}
```

Custom Exception

```
class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Called compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Normal exit");
  }

  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output:

Called compute(1)

Normal exit

Called compute(20)

Caught MyException[20]