OBJECT ORIENTED PROGRAMMING USING JAVA

OUTLINE

- Introduction
- Default Exceptional handler
- try
- catch
- finally
- throw
- throws

WHAT IS EXCEPTION??

- **Dictionary Meaning:** Exception is an abnormal condition.
- An unwanted / unexpected event that disturbs, normal flow of the program (execution) is called exception.
- In java, exception is an **event that disrupts the normal flow of the program**.
- It is an object which is thrown at runtime.

Example

1. Read data from remote file, locating at Bennett University (file not found exception)
2. Online class exception- scenario, where during class your internet stop working.

TYPES OF PROGRAM ERRORS

We distinguish between the following types of errors:

1. **Syntax errors**: errors due to the fact that the syntax of the language is not respected.

2. **Semantic errors**: errors due to an improper use of program statements.

3. **Logical errors**: errors due to the fact that the specification is not respected.

From the point of view of when errors are detected, we distinguish:

1. **Compile time errors**: syntax errors and static semantic errors indicated by the compiler.

2. **Runtime errors**: dynamic semantic errors, and logical errors, that cannot be detected by the compiler(debugging).

WHAT IS THE PURPOSE OF EXCEPTIONAL HANDLING??

Example:

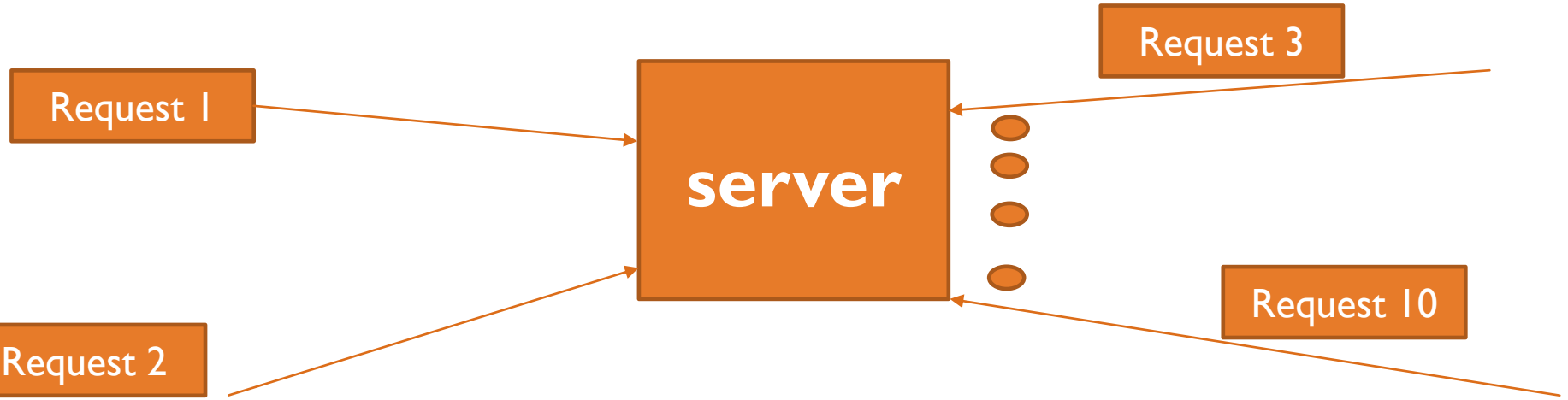
Opend db connetion

Open connection

Read the data

Sql exception

Close connection



Problem: Because of exception we should not block or miss something.

Solution: Exceptional Handling

Example: Problem: Writing code on code zinger, suddenly power cut occur then it may lead to loss of data.

Solution: Have power back up.

EXCEPTION HANDLING IN JAVA

- The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- Exceptional handling does not mean **repairing an exception.**
- Here we have to provide **alternative way to continue** the **rest of the program** normally is the concept of exceptional handling.

- **Example: Our programming requirement is to read data from remote file locating at Bennett university. At run time, if Bennett university file is not available. If we want that our program should not terminated abnormally, then we have to provide some local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.**

ADVANTAGE OF EXCEPTION HANDLING

- **The core advantage of exception handling is to maintain the normal flow of the application.**
- **Exception normally disrupts the normal flow of the application that is why we use exception handling.**
- **Suppose there is 6 statements in your program and there occurs an exception at statement 3, rest of the code will not be executed i.e. statement 4 to 6 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.**

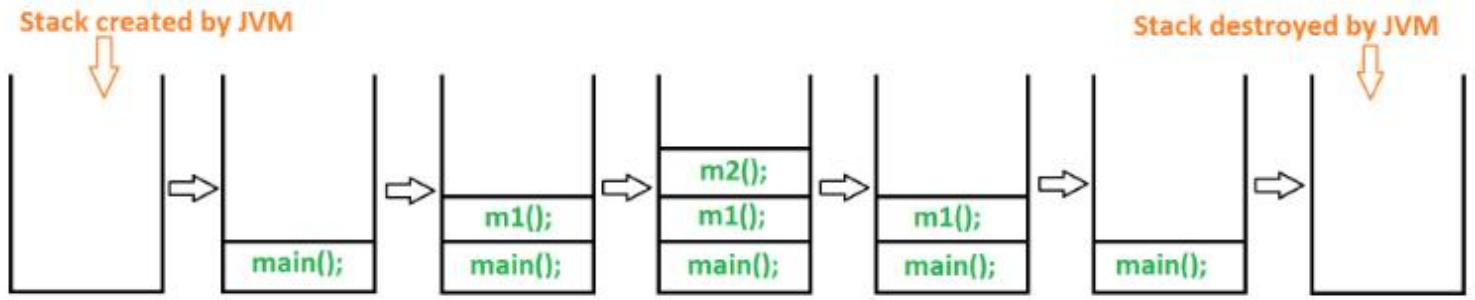
- **Let's take a scenario:**
- statement 1;
- statement 2;
- statement 3; //exception occurs
- statement 4;
- statement 5;
- statement 6;

RUNTIME STACK MECHANISM

- Runtime Stack Mechanism as its name indicates is a Stack that have been created by **JVM** for each Thread at the time of Thread creation, **JVM** store every methods calls performed by that Thread in the Stack.
- Each entry or method call called "**activation record**" / "**stack frame**".

```java
public class Main{

  public static void main(String ... args){
    m1();
  }

  public static void m1(){
    m2();
  }

  public static void m2(){
    System.out.println("Bennett university");
  }
}
```

Output :
Bennett University



DEFAULT EXCEPTIONAL HANDLING IN JAVA

```java
public class Main{

  public static void main(String ... args){
      m1();
  }

  public static void m1(){
      m2();
  }

  public static void m2(){
      System.out.println(10/0);
  }
}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.m2(Main.java:12)
        at Main.m1(Main.java:8)
        at Main.main(Main.java:4)
```

DEFAULT EXCEPTIONAL HANDLING IN JAVA (PROCESS)

If an exception raised, the method in which it's raised is responsible for the creation of Exceptions object by including the following information:

- Name of the Exception
- Description of the Exception
- Stack Trace

- After creating Exception object the method handover it to the JVM.
- JVM checks for Exception Handling code in that method.

If the method doesn't contain any Exception handling code then JVM terminates the method abnormally and removes the corresponding entry from the stack.

- JVM identify the caller method and checks for Exception Handling code in that method. If the caller doesn't contain any exception handling code then JVM terminates that method abnormally and removes the corresponding entry from the stack.
- This process will be continue until main() method.
- If the main() method also doesn't contain exception handling code the JVM terminates that main() method and removes the corresponding entry from the stack.
- Just before terminating the program abnormally JVM handovers the responsibility of exception handling to the Default Exception Handler which is the component of JVM.
- Default Exception Handler just print exception information to the console in the following format:

Name of Exception: Description

Stack Trace (Location of the Exception)

EXAMPLE:

```
public class Main{

  public static void main(String ... args){
     m1();
    System.out.print("Bennett");

  }

  public static void m1(){
     System.out.print(" Univeristy");
     m2();
  }

  public static void m2(){
     System.out.print(" Greater Noida ");
     System.out.print(10/0);
    System.out.print("India");

  }
}
```

EXAMPLE:

```java
public class Main{

  public static void main(String ... args){
     m1();
    System.out.print("Bennett");

  }

  public static void m1(){
     System.out.print(" Univeristy");
     m2();
  }

  public static void m2(){
     System.out.print(" Greater Noida ");
     System.out.print(10/0);
    System.out.print("India");

  }
}
```

```
Univeristy Greater Noida Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.m2(Main.java:16)
        at Main.m1(Main.java:11)
        at Main.main(Main.java:4)
```

PREDICT THE OUTPUT??

```
public class Main{

  public static void main(String ... args){
     m1();
     System.out.print(" Bennett ");
    System.out.print(10/0);


  }

  public static void m1(){
     System.out.print(" Univeristy");
     m2();
  }

  public static void m2(){
     System.out.print(" Greater Noida ");
    System.out.print("India");

  }
}
```

PREDICT THE OUTPUT??

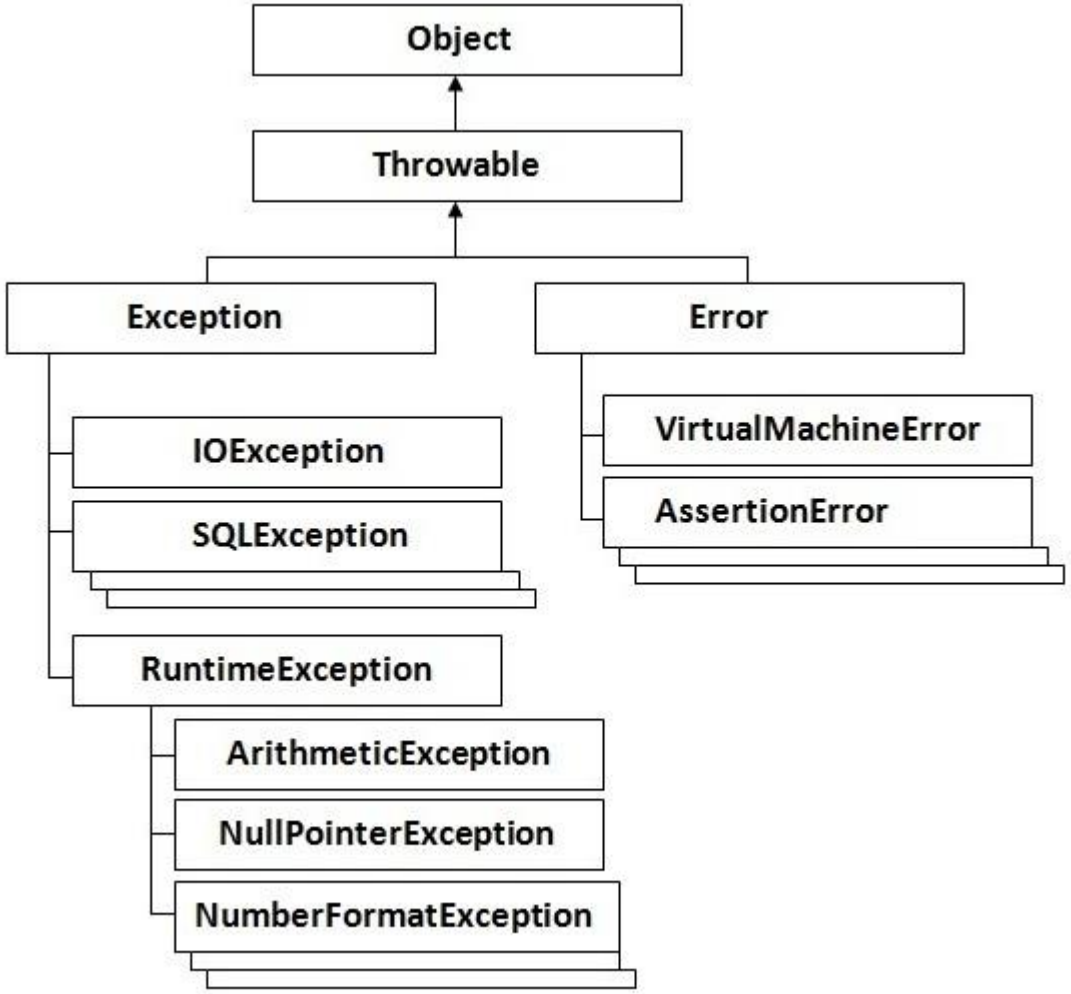
```java
public class Main{

  public static void main(String ... args){
     m1();
     System.out.print(" Bennett ");
    System.out.print(10/0);


  }

  public static void m1(){
     System.out.print(" Univeristy");
     m2();
  }

  public static void m2(){
     System.out.print(" Greater Noida ");
    System.out.print("India");

  }
}
```

```
Univeristy Greater Noida IndiaBennettException in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:6)
```

NOTE:

- If at least one method terminates abnormally then termination of a program is **abnormal**.
- If we are not satisfied with the default then we will go for **customised exceptional handling**

HIERARCHY OF JAVA EXCEPTION CLASSES



- Object class is the parent class of all the classes in java
- In Java, all errors and exceptions are represented with **Throwable** class.
- When an error occurs within a method, the method creates an object (of any subtype of Throwable) and hands it off to the runtime system.
- The object, called an exception object.
- Exception object contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called **throwing an exception.**

EXCEPTION VS ERROR

- **Most of the time Exceptions** are caused by programs and **are recoverable.**
- Most of the time errors are not caused by our program , and these **are due to lack of system resources. Errors are irrecoverable.**

Error Example

```
public class Main {
    public static void main(String[] args){
        recursiveMethod(10);
    }
    public static void recursiveMethod(int i){
        while(i!=0){
            i=i+1;
          //  System.out.println(i);
            recursiveMethod(i);

        }
    }
}
```

```
Exception in thread "main" java.lang.StackOverflowError
        at Main.recursiveMethod(Main.java:6)
```

Exception Example

```
public class Main {
    public static void main(String[] args){
        int x = 100;
        int y = 0;
        int z = x / y;
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:5)
```

TYPES OF EXCEPTION

There are mainly two types of exceptions:

- checked and unchecked
- where error is considered as unchecked exception.

The sun microsystem says there are three types of exceptions:

- Checked Exception
- Unchecked Exception
- Error

- **Note: All exception occur at runtime**

1) CHECKED EXCEPTION

- The exception checked by the compiler for the smooth execution of the program during runtime.
- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions
- e.g.IOException, SQLException etc.
- Checked exceptions are checked at compile-time.

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\bennett\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

1) CHECKED EXCEPTION

- The exception checked by the compiler for the smooth execution of the program during runtime.
- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions
- e.g.IOException, SQLException etc.
- Checked exceptions are checked at compile-time.

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\bennett\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

Output:

Exception in thread "main"
java.lang.RuntimeException: Uncompilable source code -
unreported exception
java.io.FileNotFoundException; must be caught or declared to be
thrown
    at Main.main(Main.java:5)

2) UNCHECKED EXCEPTION

- The classes that extend RuntimeException are known as unchecked exceptions.
- These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

```
public class Main {

    public static void main(String args[]) {
        int num[] = {1, 2, 3, 4};
        System.out.println(num[5]);
    }
}
```

2) UNCHECKED EXCEPTION

- The classes that extend RuntimeException are known as unchecked exceptions.
- These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

```java
public class Main {

    public static void main(String args[]) {
        int num[] = {1, 2, 3, 4};
        System.out.println(num[5]);
    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Main.main(Main.java:5)
```

3) ERROR

- Errors are irrecoverable
- e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Checked Exception	Unchecked Exception
Checked exceptions occur at compile time.	Unchecked exceptions occur at runtime.
The compiler checks a checked exception.	The compiler does not check these types of exceptions.
These types of exceptions can be handled at the time of compilation.	These types of exceptions cannot be a catch or handle at the time of compilation, because they get generated by the mistakes in the program.
They are the sub-class of the exception class.	They are runtime exceptions and hence are not a part of the Exception class.
Here, the JVM needs the exception to catch and handle.	Here, the JVM does not require the exception to catch and handle.
Examples of Checked exceptions: • **File Not Found Exception** • **No Such Field Exception** • **Interrupted Exception** • **No Such Method Exception** • **Class Not Found Exception**	Examples of Unchecked Exceptions: • No Such Element Exception • Undeclared Throwable Exception • Empty Stack Exception • Arithmetic Exception • Null Pointer Exception • Array Index Out of Bounds Exception • Security Exception

COMMON SCENARIOS WHERE EXCEPTIONS MAY OCCUR

- There are given some scenarios where unchecked exceptions can occur. They are as follows:
- **1) Scenario where ArithmeticException occurs**
- If we divide any number by zero, there occurs an ArithmeticException.
- **int** a=50/0;//ArithmeticException

COMMON SCENARIOS WHERE EXCEPTIONS MAY OCCUR

- 2) Scenario where NullPointerException occurs

- If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.
- String s=null;
- System.out.println(s.length());//NullPointerException

COMMON SCENARIOS WHERE EXCEPTIONS MAY OCCUR

- 3) Scenario where NumberFormatException occurs

- The wrong formatting of any value, may occur NumberFormatException.
- Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.
- String s="abc";
- int i=Integer.parseInt(s);//NumberFormatException

COMMON SCENARIOS WHERE EXCEPTIONS MAY OCCUR

- 4) Scenario where ArrayIndexOutOfBoundsException occurs

- if you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

JAVA EXCEPTION HANDLING KEYWORDS

- There are 5 keywords used in java exception handling.
- try
- catch
- finally
- throw
- throws

CUSTOMIZED EXCEPTIONAL HANDLING USING TRY AND CATCH

try block

- Java try block is used to enclose the code that might throw an exception. It must be used within the method.

- Java try block must be followed by either catch or finally block.

SYNTAX OF JAVA TRY-CATCH

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

Syntax of try-finally block

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

CATCH BLOCK

- Java catch block is used to handle the Exception.
- It must be used after the try block only.
- You can use multiple catch block with a single try.

PROBLEM WITHOUT EXCEPTION HANDLING

```
public class Main
{
    public static void main(String[] args) {
        System.out.println("bennett");
        System.out.println(50/0);
        System.out.println("university");

    }
}
```

PROBLEM WITHOUT EXCEPTION HANDLING

```java
public class Main
{
    public static void main(String[] args) {
        System.out.println("bennett");
        System.out.println(50/0);
        System.out.println("university");

    }
}
```

output

```
bennett
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:13)
```

PROBLEM USING TRY-CATCH

```
public class Main
{
    public static void main(String[] args) {
        System.out.println("bennett");
        try {
        System.out.println(50/0);
        }
        catch(ArithmeticException e)
        {
            System.out.println(0);
        }
        System.out.println("university");

    }
}
```

PROBLEM USING TRY-CATCH

```java
public class Main
{
    public static void main(String[] args) {
        System.out.println("bennett");
        try {
        System.out.println(50/0);
        }
        catch(ArithmeticException e)
        {
            System.out.println(0);
        }
        System.out.println("university");

    }
}
```

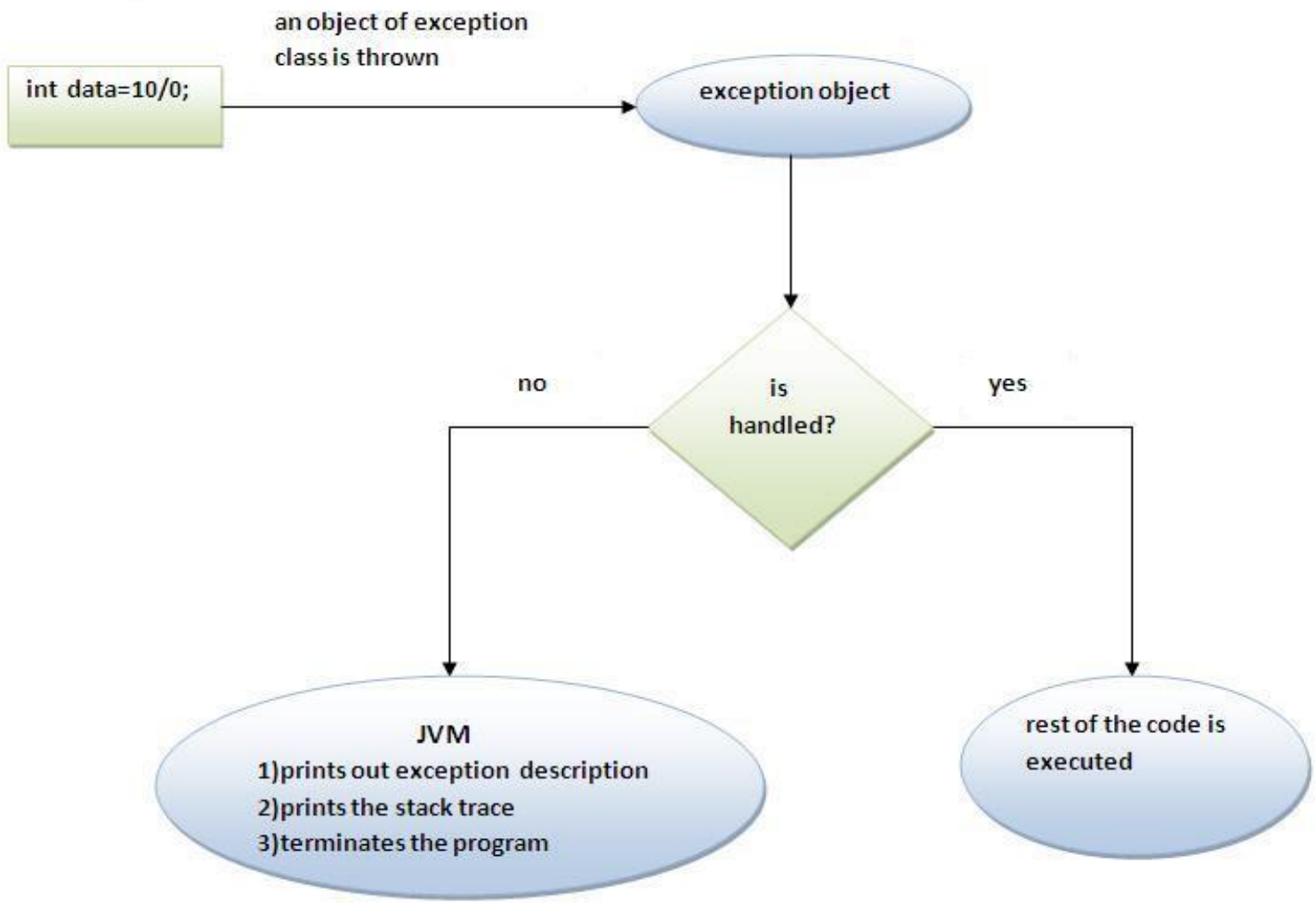
Risky code

The code which may rise an exception is called risky code and we have to define that code inside try block and corresponding handling code we have to define inside catch block.

output

```
bennett
0
university
```

INTERNAL WORKING OF JAVA TRY-CATCH BLOCK



EXPLANATION

- 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 exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

CONTROL FLOW IN TRY CATCH

Case 1: if there is no exception

```
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println("university");
        System.out.println();
        }
        catch(Exception e)
        {
            System.out.println(0);
        }
        System.out.println("Greater Noida");

    }
}
```

CONTROL FLOW IN TRY CATCH

Case 1: if there is no exception

```
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println("university");
        System.out.println();
        }
        catch(Exception e)
        {
            System.out.println(0);
        }
        System.out.println("Greater Noida");

    }
}
```

Order of execution output

```
bennett
university

Greater Noida
```

Termination: normal

CONTROL FLOW IN TRY CATCH

Case 2: if there is an exception in statement 2 of try

```
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println(10/0);
        System.out.println("university");

        }
        catch(Exception e)
        {
            System.out.println(0);
        }
        System.out.println("Greater Noida");

    }
}
```

CONTROL FLOW IN TRY CATCH

Case 2: if there is an exception in statement 2 of try

```
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println(10/0);
        System.out.println("university");

        }
        catch(Exception e)
        {
            System.out.println(0);
        }
        System.out.println("Greater Noida");

    }
}
```

Order of execution

```
bennett
0
Greater Noida
```

Termination: normal

Note: Hence length of try block should be as less as possible and take only risky code inside try block

Or go for multiple try block

CONTROL FLOW IN TRY CATCH

Case 3: if there is an exception in statement 2 and corresponding catch not matched

```
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println(10/0);
        System.out.println("university");

        }
        catch(NullPointerException e)
        {
            System.out.println(0);
        }
        System.out.println("Greater Noida");

    }
}
```

CONTROL FLOW IN TRY CATCH

Case 3: if there is an exception in statement 2 and corresponding catch not matched

```
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println(10/0);
        System.out.println("university");

        }
        catch(NullPointerException e)
        {
            System.out.println(0);
        }
        System.out.println("Greater Noida");

    }
}
```

Order of execution

output

```
bennett
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:14)
```

Termination: Abnormal

CONTROL FLOW IN TRY CATCH

Case 4: if there is an exception in try as well as in catch statement or outside try statement

```
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println(10/0);
        System.out.println("university");

        }
        catch(Exception e)
        {
            System.out.println(10/0);
        }
        System.out.println("Greater Noida");

    }
}
```

CONTROL FLOW IN TRY CATCH

Case 4: if there is an exception in try as well as in catch statement or outside try statement

```java
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println(10/0);
        System.out.println("university");

        }
        catch(Exception e)
        {
            System.out.println(10/0);
        }
        System.out.println("Greater Noida");

    }
}
```

Order of execution
output

```
bennett
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:14)
```

Termination: Abnormal

NOTE:

1. With in the try block if anywhere an exception occur then rest of the try wont be executed even though we handled that exception hence with in the **try block we have to take only risky code and length of try block should be as less as possible.**

2. In addition other than try block there may be chance of rising an exception inside catch and finally blocks.

3. If any statement which is not part of try block and an exception occur then it is always abnormal termination.

METHODS TO PRINT EXCEPTION INFORMATION

```java
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println(10/0);
        System.out.println("university");

        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("Greater Noida");

    }
}
```

```
bennett
java.lang.ArithmeticException: / by zero
Greater Noida
```

toString()

```java
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println(10/0);
        System.out.println("university");

        }
        catch(ArithmeticException e)
        {
            System.out.println(e.getMessage());
        }
        System.out.println("Greater Noida");

    }
}
```

```
bennett
/ by zero
Greater Noida
```

getMessage()

```java
public class Main
{
    public static void main(String[] args) {
        try {
        System.out.println("bennett");
        System.out.println(10/0);
        System.out.println("university");

        }
        catch(ArithmeticException e)
        {
            e.printStackTrace();
            }
        System.out.println("Greater Noida");

    }
}
```

```
bennett
java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:14)
Greater Noida
```

printStackTrace()

TRY WITH MULTIPLE CATCH BLOCKS

```java
public class Main
{
    public static void main(String[] args) {
        try {

            System.out.println(100/0);
        }

        catch(ArithmeticException e) {
            System.out.println("bennett");

        }
        catch(Exception e)
        {
            System.out.println("university");
        }
    }
}
```

```
bennett
```

Case: 1

```java
public class Main
{
    public static void main(String[] args) {
        try {

            System.out.println(100/0);
        }
         catch(Exception e)
        {
            System.out.println("university");
        }
        catch(ArithmeticException e) {
            System.out.println("bennett");

        }
    }
}
```

```
Main.java:20: error: exception ArithmeticException has already been caught
                catch(ArithmeticException e) {
                ^
1 error
```

Case:2

```java
public class Main
{
    public static void main(String[] args) {
        try {

            System.out.println(100/0);
        }
          catch(ArithmeticException e) {
            System.out.println("bennett");

        }
        catch(ArithmeticException e) {
            System.out.println("bennett");

        }
    }
}
```

Case:3

TRY WITH MULTIPLE CATCH BLOCKS

- The way of handling an exception is varied from exception to exception, hence for every exception type it is highly recommended to take separate catch block i.e. try with multiple catch block is always possible and recommended to use.
- At a time only one Exception is occurred and at a time only one catch block is executed

Limitation

- Order of execution is very important when having multiple catch, because first we will take child first and the parent exception, if we take parent exception first and then child exception you will surely get a compilation error.

TRY WITH MULTIPLE CATCH BLOCKS

- **If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block**

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   catch(Exception e){System.out.println("common task completed");}

   System.out.println("rest of the code...");
 }
}
```

TRY WITH MULTIPLE CATCH BLOCKS

- **If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block**

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[4]=30/0;
   }
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   catch(Exception e){System.out.println("common task completed");}

   System.out.println("rest of the code...");
 }
}
```

Output:
task1 is completed
rest of the code

TRY WITH MULTIPLE CATCH BLOCKS

- If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/1;
   }
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   catch(Exception e){System.out.println("common task completed");}

   System.out.println("rest of the code...");
 }
}
```

TRY WITH MULTIPLE CATCH BLOCKS

- If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/1;
   }
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   catch(Exception e){System.out.println("common task completed");}

   System.out.println("rest of the code...");
 }
}
```

Output:
task2 is completed
Rest of the code

JAVA NESTED TRY BLOCK

- The try block within a try block is known as nested try block in java.

- 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.

JAVA NESTED TRY BLOCK: EXAMPLE

```
class Main{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
    int a[]=new int[5];
    a[5]=4;
    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

    System.out.println("other statement");
  }catch(Exception e){System.out.println("handeled");}

  System.out.println("normal flow..");
 }
}
```

output

```
going to divide
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 5
other statement
normal flow..
```

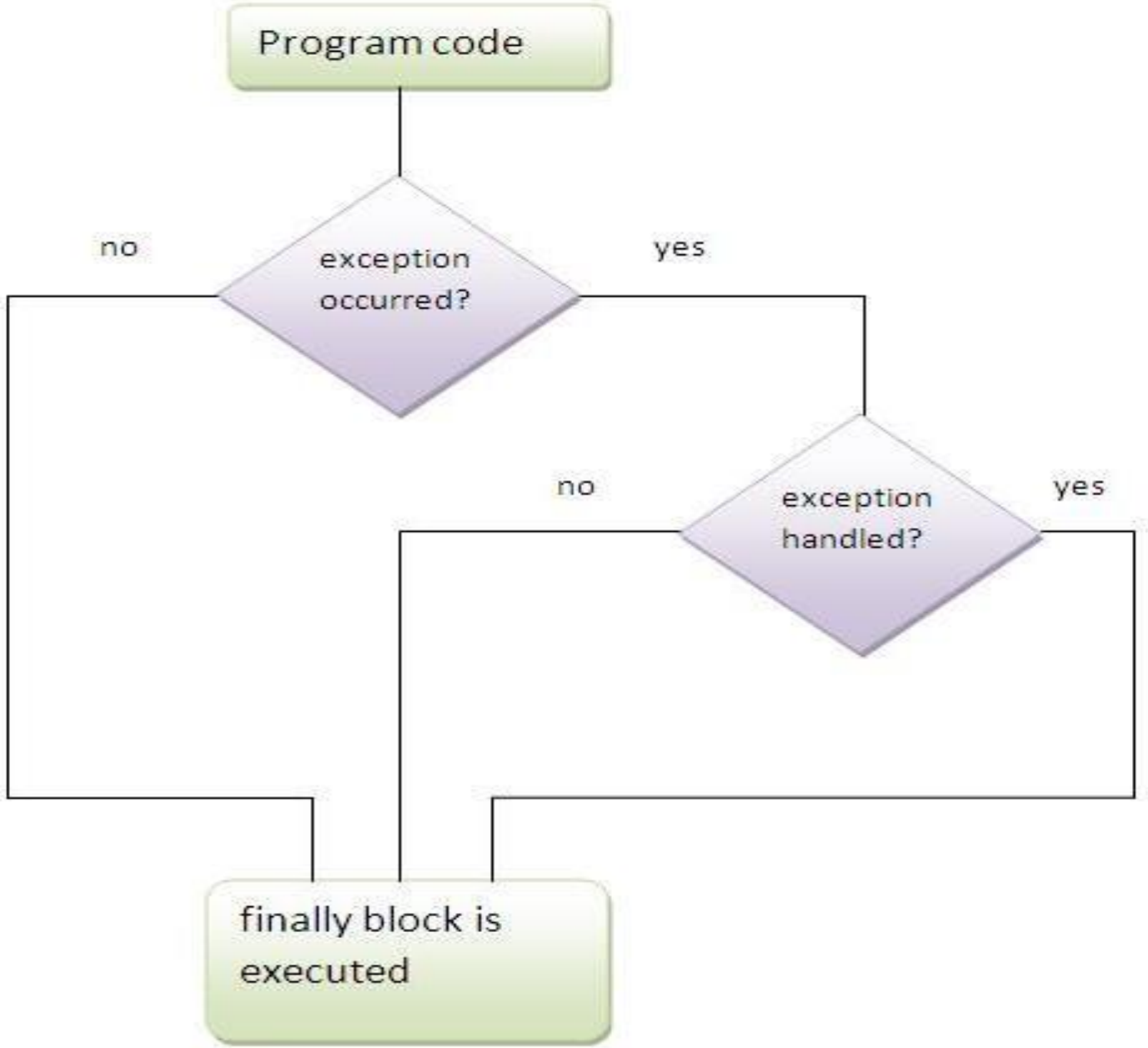
JAVA FINALLY BLOCK

- Finally block is used to perform **clean-up** activities that we related to try block , means what ever resources we open in the try block will be closed inside finally block.

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

- Java finally block **is always executed whether exception is handled or not**.

- Java finally block **follows try or catch block**.


Program code
exception occurred?
no
yes
exception handled?
no
yes
finally block is executed

CASE 1: FINALLY EXAMPLE WHERE EXCEPTION DOESN'T OCCUR

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

CASE 1: FINALLY EXAMPLE WHERE EXCEPTION DOESN'T OCCUR

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Output:

```
finally block is always executed
rest of the code...
```

CASE 2: FINALLY EXAMPLE WHERE EXCEPTION OCCURS AND NOT HANDLED.

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

CASE 2: FINALLY EXAMPLE WHERE EXCEPTION OCCURS AND NOT HANDLED.

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Output:
finally block is always executed
Exception in thread main

java.lang.ArithmeticException:/ by zero

CASE 3:FINALLY EXAMPLE WHERE EXCEPTION OCCURS AND HANDLED.

```
public class TestFinallyBlock2{

  public static void main(String args[]){

  try{

   int data=25/0;

   System.out.println(data);

  }

  catch(ArithmeticException e){System.out.println(e);}

  finally{System.out.println("finally block is always exe
cuted");}

  System.out.println("rest of the code...");

  }

}
```

- **Output**
- **Exception in thread main java.lang.ArithmeticException:/ by zero**
- **finally block is always executed**
- **rest of the code...**

- **Note:**
- **For each try block there can be zero or more catch blocks, but only one finally block.**

- **The finally block will not be executed if program exits(either by calling System.exit()**
- **or by causing a fatal error that causes the process to abort).**

THROW KEYWORD

- Some times we can create exception object explicitly and handover the object manually for which we use throw keyword.

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or un-checked exception in java by throw keyword.
- Hence the main objective of throw keyword is to handover the object manually to jvm.
- Note: throw keyword are used for custom/ user defined exception.

EXAMPLE

```
public class TestThrow1{
   static void validate(int age){
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

```
class Bankclass extends RuntimeException

{
    Bankclass (String msg)
    {
    super(msg);
    }

}

public class Main {
  public static void main(String[ ] args) {
    int amount =990;
    if (amount<1000)
    {
     throw new Bankclass("transaction unsucessful");
    }
     else
     {
     System.out.println("continue transaction");
     }

         System.out.println("amount is" + amount);
    }
  }
```

```
Exception in thread "main" Bankclass: transaction unsucessful
        at Main.main(Main.java:16)
```

Output:
Exception in thread main java.lang.ArithmeticException:not valid

JAVA EXCEPTION PROPAGATION

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

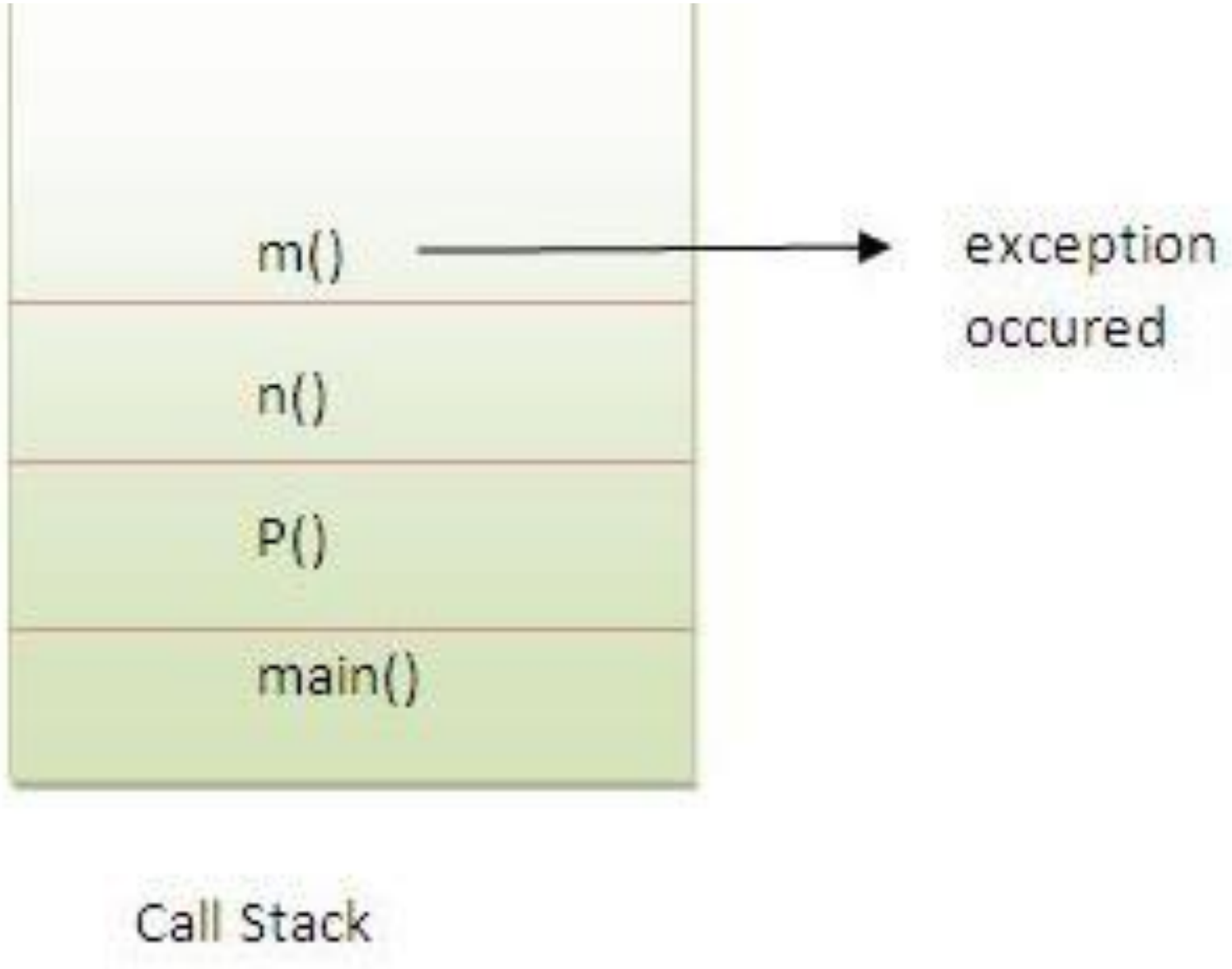
Note: the next line after throw exception will give us unreachable statement because compiler is aware about that statement that it is not going to execute.

JAVA EXCEPTION PROPAGATION: EXAMPLE

```
class TestExceptionPropagation1{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception h
andled");}
  }
  public static void main(String args[]){

    TestExceptionPropagation1 obj=new
  TestExceptionPropagation1();
    obj.p();
    System.out.println("normal flow...");
   }
}
```

OUTPUT

- exception handled
- normal flow...



CHECKED EXCEPTION

```
class TestExceptionPropagation2{
  void m(){
    throw new java.io.IOException("device error");//checked exception
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handeled");}
  }
  public static void main(String args[]){
    TestExceptionPropagation2 obj=new TestExceptionPropagation2();
    obj.p();
    System.out.println("normal flow");
  }
}
```

throws KEYWORD

- The **Java throws keyword** is used to declare an exception.
- To delegate the responsibility of exceptional handling to the caller method, caller method can be another method or jvm. Then caller method is responsible to handle the exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- **Note: It is generally used for checked exception.**
- **Throws keyword does not prevent us from abnormal termination**

Syntax of java throws

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

Which exception should be declared

checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Which exception should be declared

checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Example

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception  }
  void n()throws IOException{
    m();    }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}   }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  } }
```

Example

```
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception  }
  void n()throws IOException{
    m();    }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}   }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  } }
```

OUTPUT

exception handled

normal flow...

If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

Case1: You caught the exception i.e. handle the exception using try/catch.

Case2: You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");   }}
```

Case1: You handle the exception

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");   }}
```

OUTPUT:
exception handled
normal flow...

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.

- B)In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();
    System.out.println("normal flow...");
  }  }
```

OUTPUT:
device operation performed
normal flow...

B)Program if exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

B)Program if exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

Output:
Runtime Exception

Difference between throw and throws in Java

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

THANK YOU
?