JAVA @17

Exception Handling

Objective

After completing this session you will be able to understand,

- Introduction to Exception Handling
- Exception Hierarchy
- Try-Catch-Finally
- Multiple catch block
- Nested try block
- Throws keyword
- Throw keyword
- User Defined Exception

Exception

What is an Exception?

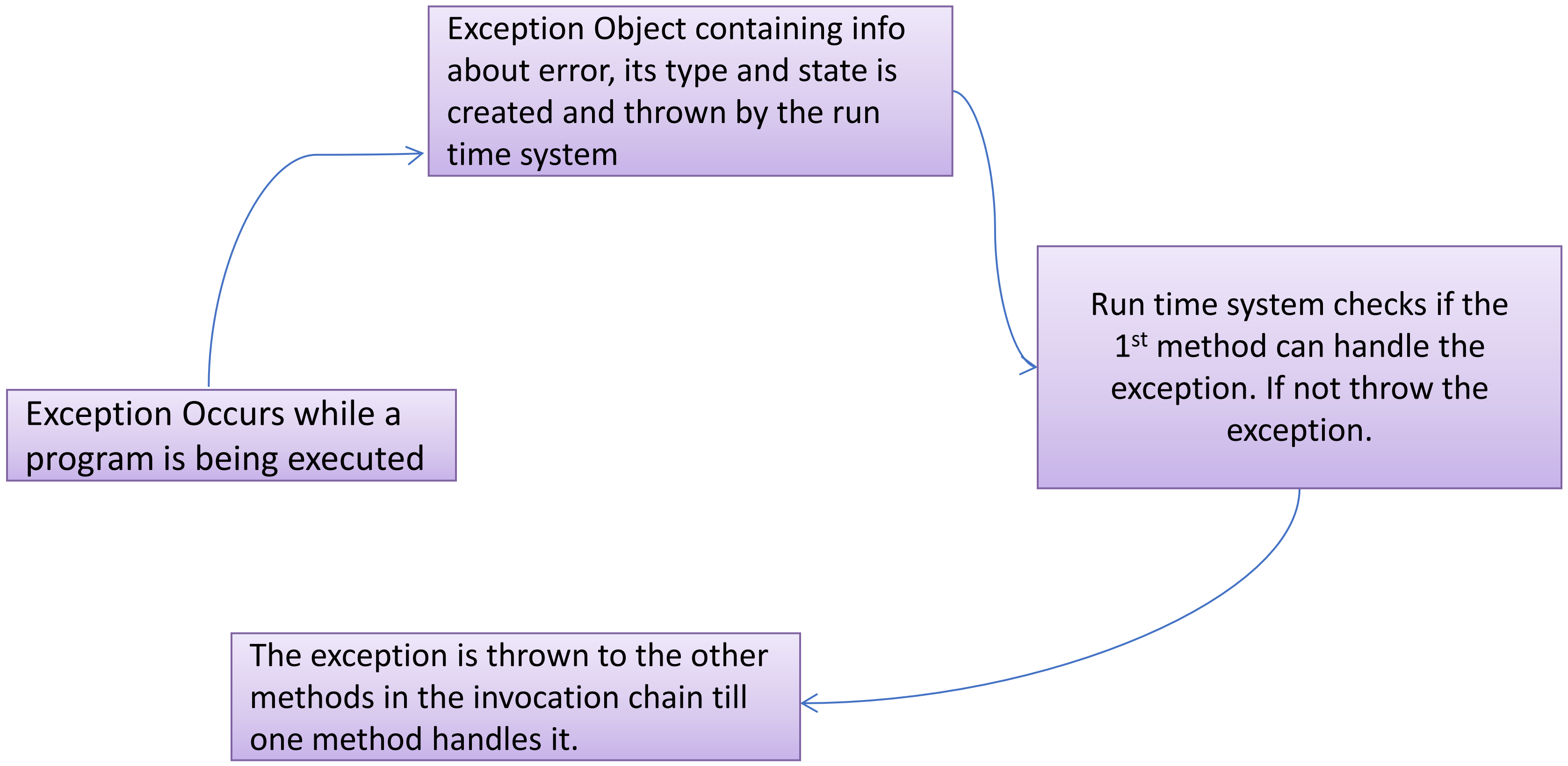
Exception refers to any abnormality or an **error** that occurs during **run time**

- An exception in Java is a signal that indicates the occurrence of some important or unexpected condition during execution of a program at runtime.
- Exception causes normal program flow to be disrupted.

Examples :

- **int num=5/0 – Divide by Zero Error –Arithmetic Exception**
- **Out of Memory Error.**
- **Trying to open a file that has been deleted.**

Exception Handling



Exception Occurs while a program is being executed

Exception Object containing info about error, its type and state is created and thrown by the run time system

Run time system checks if the 1st method can handle the exception. If not throw the exception.

The exception is thrown to the other methods in the invocation chain till one method handles it.

In Java Handlers are used for handling exceptions.

What happen When an Exception occurs in a Program?

What happens when an Exception occurs?

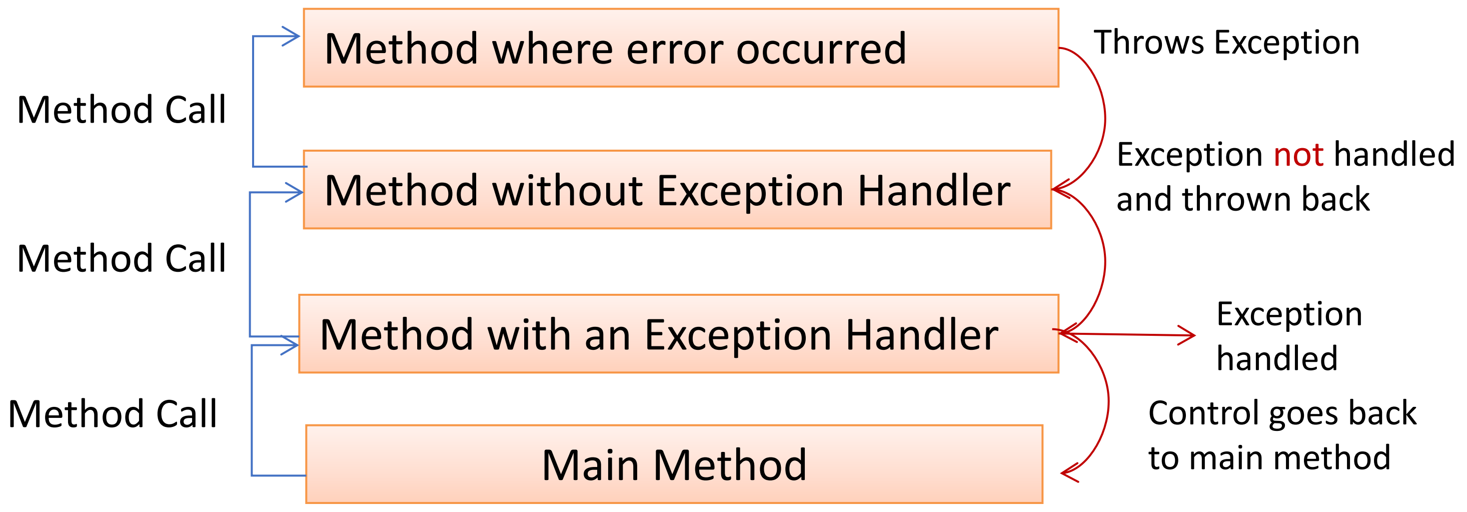
Step 1:

When an exception occurs within a method, the method creates an exception object and hands it off to the run-time system (called throwing an exception)

Exception object contains information about the error, including its type and the state of the program when the error occurred.

Step 2:

The run time system searches the call stack for a method that contains the method handler


Method where error occurred
Throws Exception
Method Call
Method without Exception Handler
Exception not handled and thrown back
Method Call
Method with an Exception Handler
Exception handled
Method Call
Main Method
Control goes back to main method

What happen when an Exception occurs in a Program?

Step 3:

When an appropriate handler is found, the run-time system passes the exception to the handler,

- The exception handler catches the exception and handles the exception.

If the run-time system cannot find an appropriate method to handle the exception, then the run-time system terminates and uses the default exception handler.

We will learn about how the handler handles the exception in the subsequent slides.

Try to run this program in your Eclipse IDE

```java
public class DivByZero {
    public static void main(String args[]) {
        System.out.println(3/0);
        System.out.println("Pls. print me");
        }
}
```

You can notice the following exception thrown by the run time system,

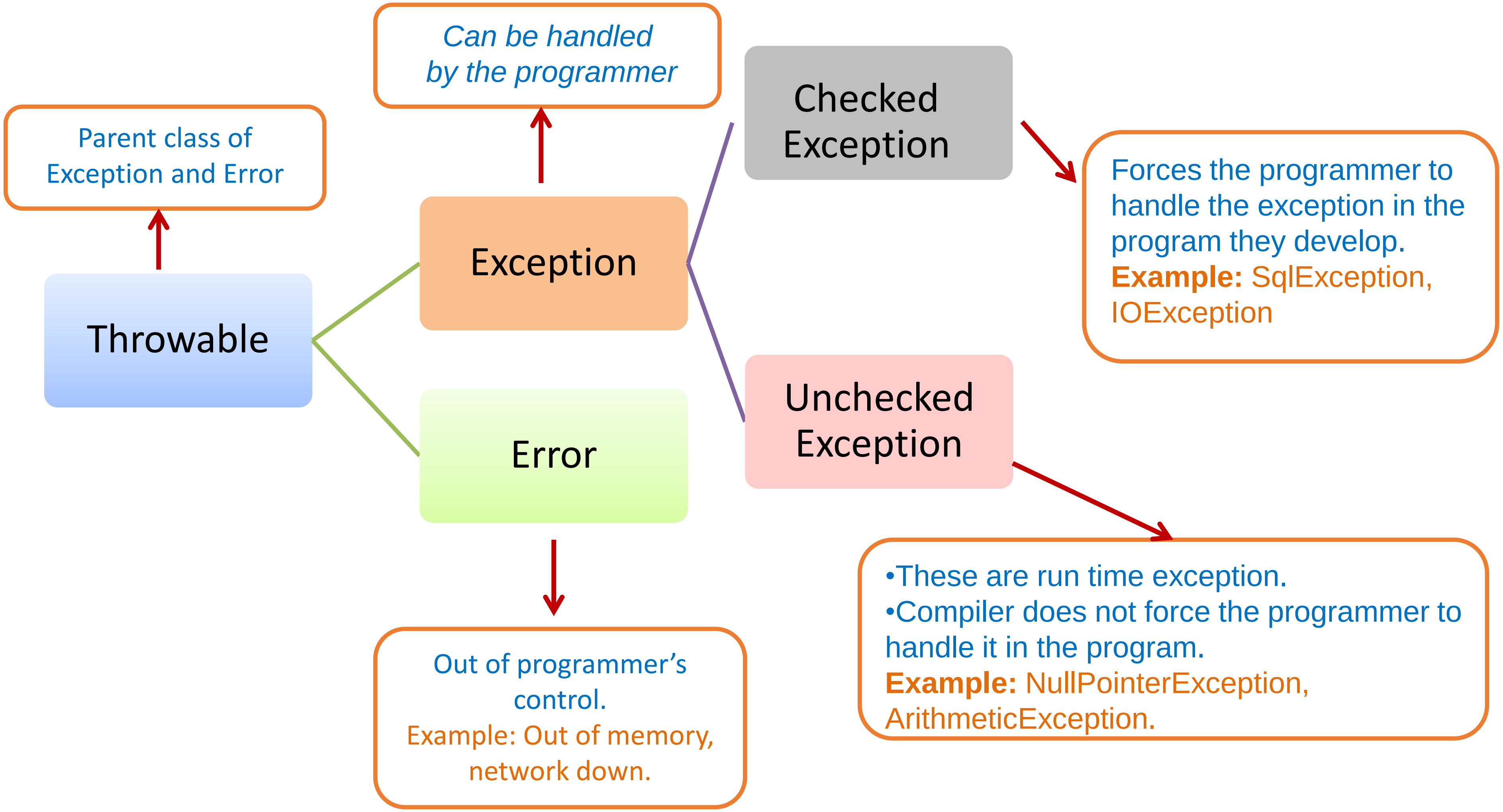
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at com.cognizant.academy.handson.DivByZero.main(DivByZero.java:3)
```

Benefits of Exception

Benefits of Java Exception Handling Framework:

- It separates Error-Handling code from “regular” business logic code.
- It can propagate errors up the call stack till a handler handles the exception.
- It can group and categorize the exception types.

Exception Hierarchy


Parent class of Exception and Error
Throwable
Can be handled by the programmer
Exception
Error
Checked Exception
Unchecked Exception
Forces the programmer to handle the exception in the program they develop.
Example: SqlException, IOException
Out of programmer's control.
Example: Out of memory, network down.
•These are run time exception.
•Compiler does not force the programmer to handle it in the program.
Example: NullPointerException, ArithmeticException.

Checked vs Unchecked

Checked Exception	Unchecked Exception
At compile time, the java compiler automatically checks that a program contains handlers for checked exceptions.	The compiler doesn't force them to be declared in the throws clause.
Checked exceptions must be explicitly caught or propagated using **try-catch-finally** blocks	Unchecked exceptions do not have this requirement. They don't have to be caught or declared thrown.
Checked exceptions in Java extend the *java.lang.**Exception*** class	Unchecked exceptions extend the *java.lang.**RuntimeException***.
Exception handling is mandated by JVM for these exceptions	It is not advisable to catch these exceptions since it might make the code unstable.
Example: **IOException**	Example: **NullPointerException**

Checked Exception

CheckedException	Description
IOException	Signals that an I/O exception of some sort has occurred.
InterruptedException	Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.
ParseException	Signals that an error has been reached unexpectedly while parsing.
SQLException	An exception that provides information on a database access error or other errors.

Uncheck Exception

UnCheckedException	Description
ArithmeticException	Thrown when an exceptional arithmetic condition has occurred.
ClassCastException	Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.
NullPointerException	Thrown when an application attempts to use null in a case where an object is required.
ArrayIndexOutOfBoundsException	Thrown to indicate that an array has been accessed with an illegal index.

Exception Handling

How can Exception be handled?

Exception can be handled using a exception handler.

What is a Exception Handler?

A set of code which can handle an error conditions in a program systematically by taking necessary action

Exception Handling Techniques:

- Option I: try-catch-finally
- Option II: throws.

Option I – try, catch & finally

try-catch

When a program performs an operation that causes an exception, an exception will be thrown. Exception can be handled by using the try and catch blocks.

How is it done?

- The suspected code is embraced in the try block, followed by the catch block in which the code to handle the exception is written.
- In the catch block, the programmer can also write the code to recover from the exception and can also print the exception.
- A catch block is executed only if it catches the exception coming from within the corresponding try block.

Option I – try, catch & finally

Syntax:

```
try{

        // The code that is prone to throw exception
}
catch(Exception exception){
        // What to do if this exception is thrown
}
finally{
        //to be done whether exception is thrown or not
        (Will be explained in next slide)
}
```

```
public void divide(int a,ir
        int quotient=0;
        try{
                quotient=a/b;
        }
        catch(ArithmeticExc
                System.out.pr
        }
        finally{
                System.out.pr
        }
}
```

finally block

Features of Finally block:

- The finally block will execute whether or not an exception is thrown.
- Finally block can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method.
- The finally block is optional.
- If a finally block is associated with a try, the finally block will be executed upon conclusion of the try
- A try can contain multiple catch block and only one finally block.

Execution flow in try, catch

```
public divide(int dividend, int divisor){
try{
   result = dividend/ divisor;
   // other statements..
}
Catch(ExceptionObject)
{
   // Exception handled
}
   // other statements..
}
```

If exception raised.

Exception Caught

Exception handled

Other statements till the end of the method will be triggered.

Execution flow try, catch & finally

```
public divide(int dividend, int divisor){
  try{
     result = dividend/ divisor;
     //other statements..
  }
  Catch(Exception Object)
  {
     //Exception handled
  }
  finally
  {
     //some logic
  }
     //other statements..
}
```

If exception raised.

Exception Caught

Exception handled

Finally block statements invoked

Other statements till the end of the method will be triggered.

Execution flow when NO exception raised in try, catch & finally block

```
public divide(int dividend, int divisor){
try{
    result = dividend/ divisor;
    //other statements..
}
Catch(ExceptionObject)
{
    //Exception handled
}
finally
{
    //some logic
}
    //other statements..
}
```

If NO exception raised in the method

Finally block statements invoked

Other statements till the end of the method will be triggered.

Multiple catch blocks

Multiple catch blocks:

One block of code can generate different types of exception.

Example:

```
try {
    int den = Integer.parseInt(args[0]);
    System.out.println(3/den);
}
```

This can be handled by having multiple catch blocks

Example of Multiple Catch blocks:

```
try{
        int den = Integer.parseInt(args[0]);
        System.out.println(3/den);
}catch(ArrayIndexOutOfBoundsException ab){
        // Exception a handled here
}catch(Arithmetic Exception ar){
        // Exception b handled here
}
```

Based on the exception thrown in the try block, the appropriate catch block will be executed

Multiple Exception Instance

The catch clause specifies the types of exceptions that the block can handle, and each exception type is separated with a vertical bar (|)

Example of Multiple exception instance,

```
try{
        int den = Integer.parseInt(args[0]);
        System.out.println(3/den);
}catch(ArrayIndexOutOfBoundsException | Arithmetic Exception e){
        // Exception a handled here
}
```

Nested try blocks

Nested Try Blocks:

Nested try can be used when we want to catch exceptions for specific lines of code

Example of Nested Try block:

```
try{
        int den = Integer.parseInt(args[0]);
        try{
                System.out.println(3/den);
        } catch(ArithmeticException ar){
                // Exception a handled here
        }
}catch(ArrayIndexOutOfBoundsException ab){
        // Exception b handled here
}
```

This is the nested TRY. Arithmetic Exception thrown here will be caught inside this block itself

NOTE: If the inner try does not have a matching catch statement for a particular exception, the control is transferred to the outer try statement's catch handlers where it searches for a matching catch statement

Rules for try, catch & finally

Rules for writing the try-catch-finally:

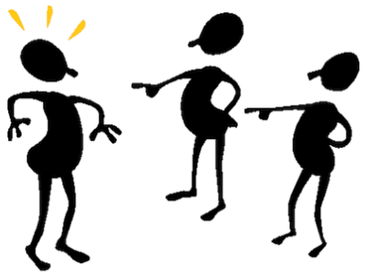
- The try block must be followed by either a catch block or a finally block, or both.
- The try block by itself is not complete.
- Any catch block must immediately follow a try block.
- The finally block must immediately follow the last catch block, or the try block if there is no catch block.

Option II – throws

Alternate way of handling exceptions:

Hope you remember how the guardian delegated

the responsibility back to the parents.



In exception handling, this delegation is done by throwing the exception.

Throws keyword is used to throw exception object from a method to the calling method.

The exception thrown by the method needs to be handled by the calling method.

Syntax - throws

Syntax:

```
<access specifier><return type><method name>() throws Exception-list
{
        //some code here which can throw
        //any type of exception specified in Exception-list
}
```

Exception-list is a comma-separated list of the exceptions that a method can throw.

The method invoking this method should handle the exceptions in the exception list using try catch block.

Example – try/catch, throws

Step 1: Create a class, *Demo* with a method, *division* with two int parameters

- Dividend
- Divisor

This method should divide the dividend by divisor and return the result.

This method should also throw an *ArithmeticException* to the calling method.

Step 2: Create a class, *ThrowsDemo* with a main method

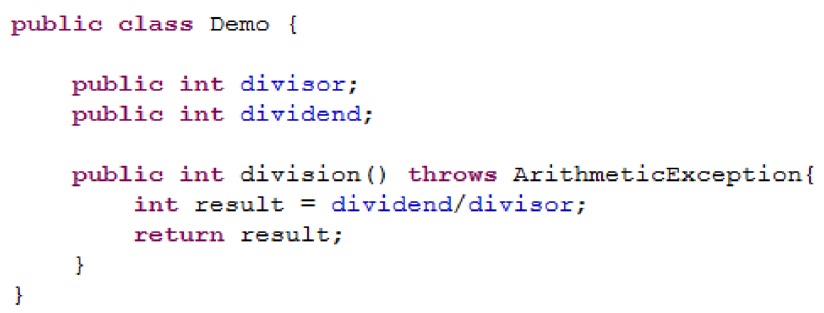
- The main method should invoke the division method in *Demo* class.
- The main method should also *catch* the *ArithmeticException* thrown by the division method and print the Exception "Arithmetic Exception is Thrown"
- The try/catch block should also have a finally block which prints a message "The result is" <Result>

Lets develop the program to demonstrate throws, try/catch.

Solution - try/catch, throws

Execute the divide method for two inputs

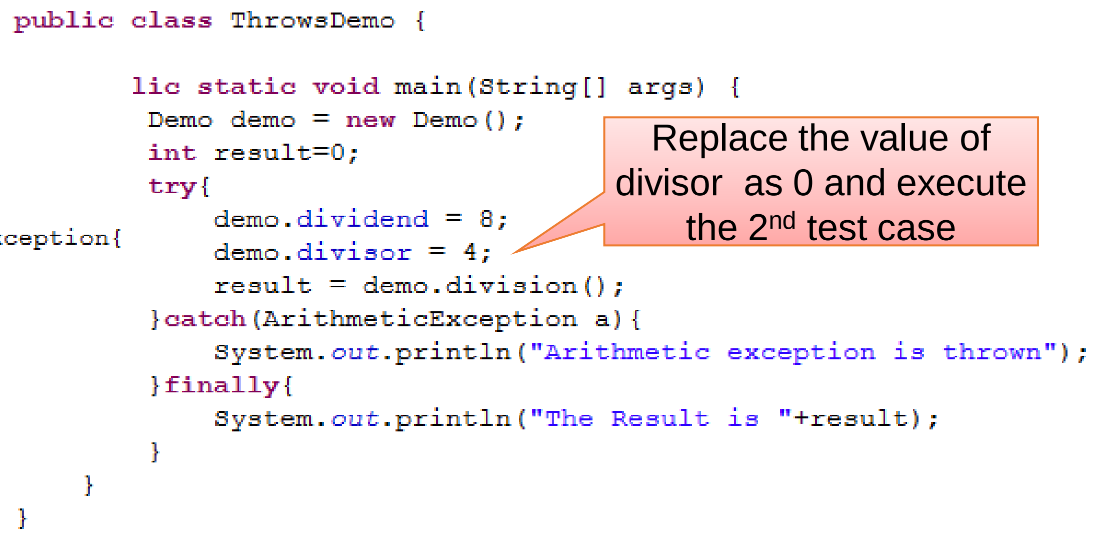
Dividend	Divisor	What did you notice?
12	3	Finally block executed and result printed as 4.
12	0	Arithmetic exception thrown, exception printed , finally block executed and result printed as ‘0’



```
public class Demo {

    public int divisor;
    public int dividend;

    public int division() throws ArithmeticException{
        int result = dividend/divisor;
        return result;
    }
}
```



```
public class ThrowsDemo {

    lic static void main(String[] args) {
     Demo demo = new Demo();
     int result=0;
     try{
         demo.dividend = 8;
         demo.divisor = 4;
         result = demo.division();
     }catch(ArithmeticException a){
         System.out.println("Arithmetic exception is thrown");
     }finally{
         System.out.println("The Result is "+result);
     }
    }
}
```

Replace the value of divisor as 0 and execute the 2nd test case

throw

Java allows you to **throw Exceptions** and generate Exceptions. An Exception you throw is an **Object**

Syntax:

throw <Exception Object>

Throw should be used in conjunction with ***throws*** clause. So if a method uses throw keyword it should either,

- Surround the ***throw*** statement with ***try/catch*** block and catch the exception thrown **(or)**
- Declare the ***throws*** clause in the methods signature for the exception thrown.

How to create a Exception Object?

```
throw new ArithmeticException(“Id not found”);
```

When to use throw?

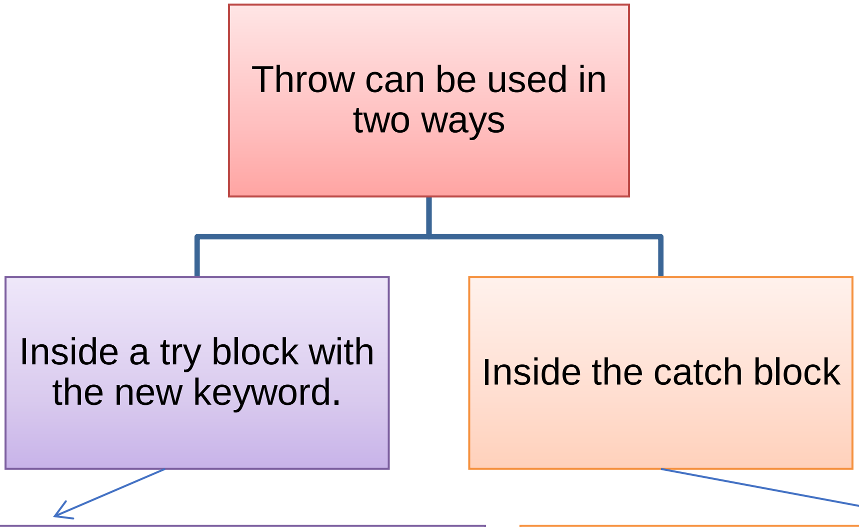
Used for throwing Throwable objects under undesired circumstances in a program and is used for throwing exception explicitly.

- User manually checks if the file is present, if the file is not present, user can throw a FileNotFoundException.
- Used to re-throw the caught exception object in catch block.

Example:

- User catches a FileNotFoundException
- Performs some logic and re-throws it as IOException (or)
- Performs some logic and throw it as FileNotFoundException itself with some additional information.

Usage of throw


Throw can be used in two ways
Inside a try block with the new keyword.
Inside the catch block

```
try{
        If(id ==0){
                throw new  <Throwable-Object>;
        }
        catch(<Throwable-Object>){
        // The thrown exception is caught here
        }
```

```
try{
        //some code here which throws an
        exception
}
catch(<Throwable-Object ){
        throw < Throwable-Object >;
}
```

Example – throw

Step 1: In the Validate class created the validate method that takes integer value as a parameter.

- If the age is less than 18, we are throwing the ArithmeticException with message “Not allowed” otherwise print a message **Welcome to vote**.
- This method should throw this ***ArithmeticException****.*

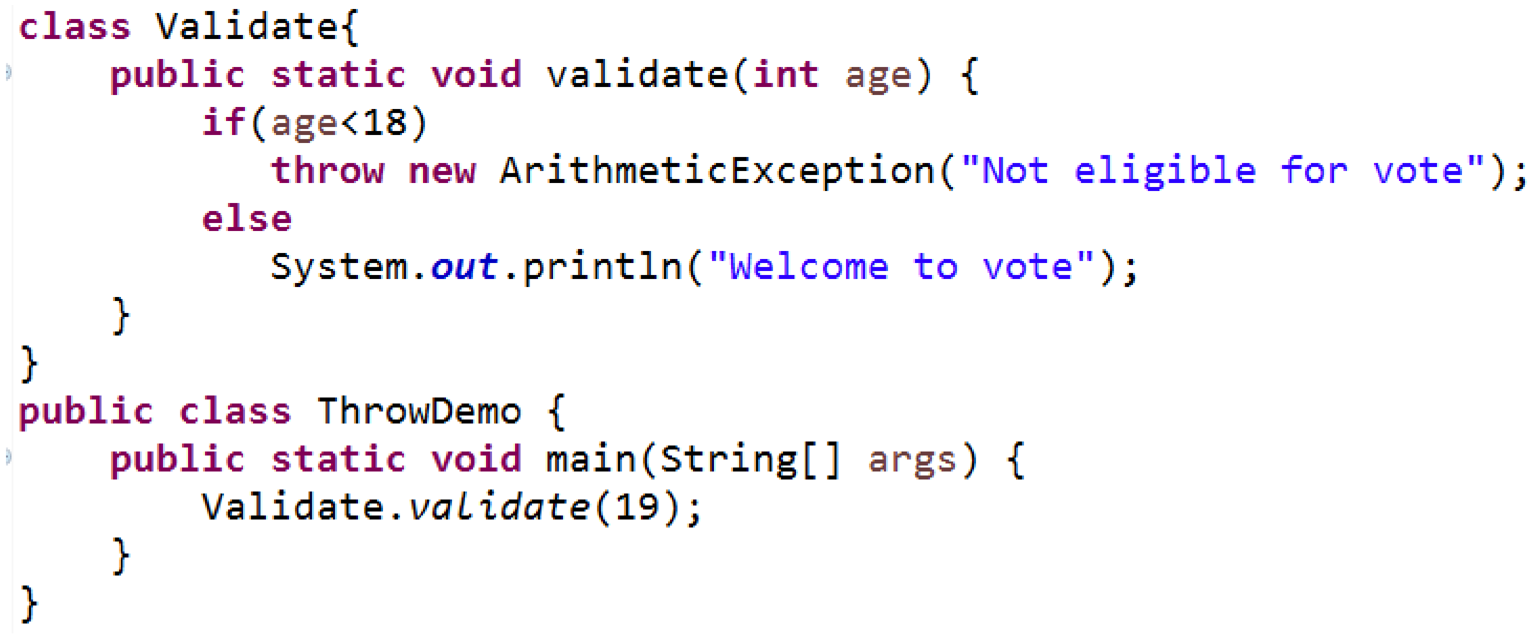
Step 2: The exception thrown needs to be handled in ***ThrowDemo***

- The main method should *catch* the ***ArithmeticException*** thrown by the validate method and print the Exception and print the message in the exception Object.
- The try/catch block should also have a finally block which prints a message “Message from validate” <Result>

Solution - throw

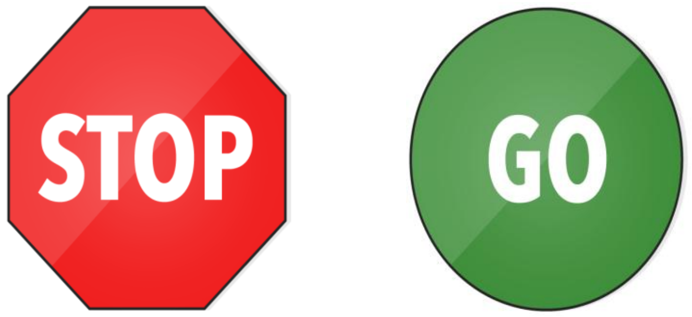
Execute the validate method for age input,

age	What did you notice?
19	Finally block executed and result printed as Welcome to vote
14	Arithmetic exception thrown, exception message ”Not eligible for vote”, finally block executed and result printed.



```
class Validate{
    public static void validate(int age) {
        if(age<18)
           throw new ArithmeticException("Not eligible for vote");
        else
           System.out.println("Welcome to vote");
    }
}
public class ThrowDemo {
    public static void main(String[] args) {
        Validate.validate(19);
    }
}
```

Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is an Exception?
- What are the types of Exceptions?
- What is an exception handler?
- What are techniques of Handling Exceptions?
- Can I have a try block alone?
- Where will I use finally block?
- What is the keyword used to manually throw the exception?

Exception – Case study

Mr. Hari with problem with development !!!

- Hari has three classes where each class calls the method in the next class.
- The third class needs to check if the employee name and employee designation is empty.
- If the fields are empty then appropriate exception needs to be thrown back to the first class.
- Hari is confused on which Exception object to throw back since there is no exception object that depicts empty values!!!
- How does Hari solve this problem?

Using custom Exceptions

User defined exception

User Defined Exceptions are custom exceptions which are created by programmers to handle the various application specific errors.

Example: In a banking application the developers can create the following exceptions in the specified scenario,

- ***InvalidAccountNumberException*** - Thrown when the account number entered by the user is wrong.
- ***AccountInactiveException*** – Exception thrown when User trying to operate an account which has become inactive.
- ***InsufficientFundException*** - Exception thrown when user trying to transfer amount with insufficient funds.

How to create user defined exception

Step 1: Create a Java class which extends the Exception class.

Step 2: Override the necessary constructors.

Scenario: Assume an application has a business logic of validating the age entered by the user, the programmer may create a user defined exception named “InvalidAgeException” and throw it when the validation fails.

Step 1: *Create a java class extending Exception*

```
public class AgeValidationException extends Exception {

    public AgeValidationException(final String message) {
        super(message);
    }
    public AgeValidationException(final Throwable exception) {
        super(exception);
    }
    public AgeValidationException(final String message,
            final Throwable exception) {
        super(message, exception);
    }
}
```

Step 2: *Override the necessary constructors.*

How can you throw the exception?

Assume in the application if the user age is < 18 he should not be allowed to register in the site.

```
public void registerProfile() throws AgeValidationException
{
    try{
        if(age< 18){
            throw new AgeValidationException("User Age is not eligible");
        }
    }
}
```

NOTE: The method throwing the exception should also declare the exception in ***throws*** clause.

Example – User Defined Exception

After this demo you will be able to create custom Exceptions and understand how to throw them.

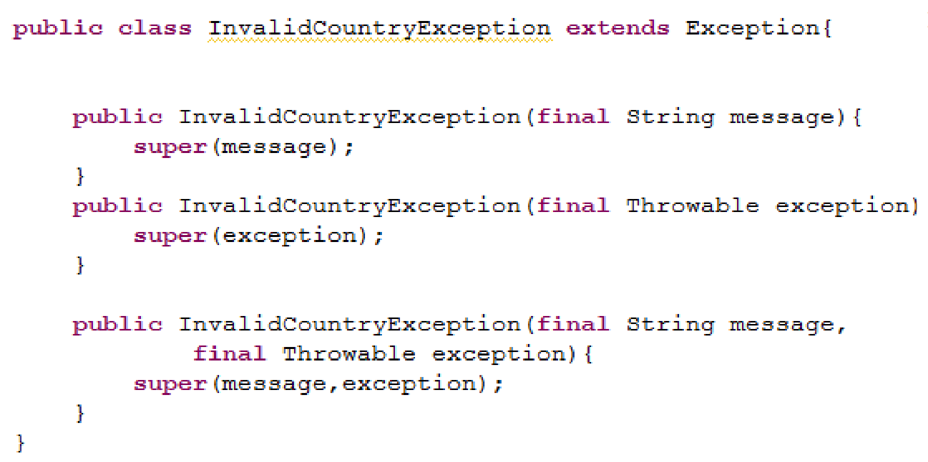
Scenario: A shopping portal provides users to register their profile. During registration the system needs to validate the user age above 18 and should be placed in India. If not the system should throw an appropriate error.

1. Create a user defined exception classes named "***InvalidCountryException***" & "***InvalidAgeException***"
2. Overload the respective constructors.
3. Create a main class "**UserRegistration**" , add the following method,

- ***registerProfile*** - The parameter are String userName , int age, String country. Add the following logic
- *if country is not equal to "India" throw a **invalidCountryException** with error message "User Outside India cannot be registered"*
- *If age < 18 throw a **InvalidAgeException** with error message "User is a Minor"*
- *Invoke the method **registerProfile*** from the main method with the data specified and see how the program behaves,

Solution – User Defined Exception

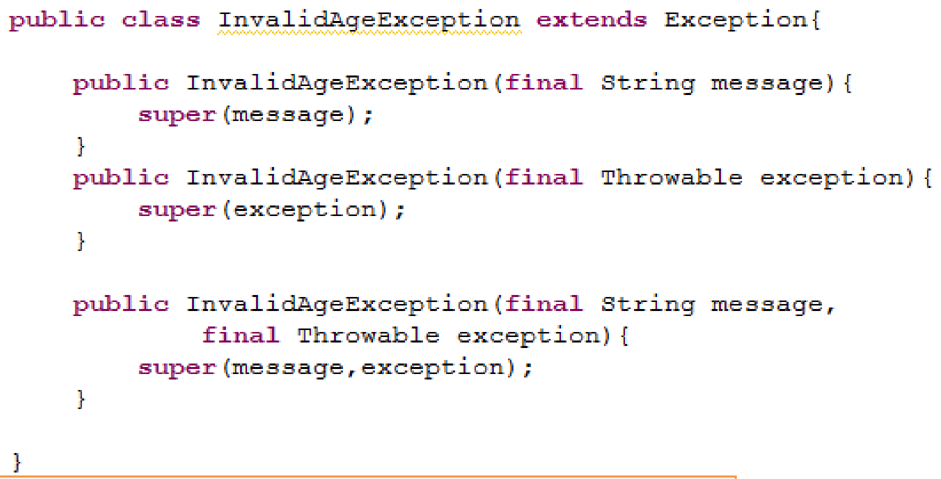
Create the exception class as mentioned below.



```
public class InvalidCountryException extends Exception{


    public InvalidCountryException(final String message){
        super(message);
    }
    public InvalidCountryException(final Throwable exception)
        super(exception);
    }

    public InvalidCountryException(final String message,
            final Throwable exception){
        super(message,exception);
    }
}
```



```
public class InvalidAgeException extends Exception{

    public InvalidAgeException(final String message){
        super(message);
    }
    public InvalidAgeException(final Throwable exception){
        super(exception);
    }

    public InvalidAgeException(final String message,
            final Throwable exception){
        super(message,exception);
    }

}
```

Execute the program for the following scenarios.

Name	Age	Country	Output
Hari	7	India	InvalidAgeException should be thrown. *The error message should be "User is a Minor"*
Saro	23	Australia	*InvalidCountryException should be thrown. The error message should be "User Outside India cannot be registered"*

Solution – User Defined Exception

Create the user registration program as mentioned below.

```java
public class UserRegistration {

    public void registerProfile(String userName,int age,String country)
    throws InvalidCountryException,InvalidAgeException{
        if(!"India".equals(country)){
            throw new InvalidCountryException("User Outside India cannot be registered");
        }
        if(age<18){
            throw new InvalidAgeException("User is a Minor");
        }

    }
    public static void main(String[] args) {
        UserRegistration userReg = new UserRegistration();
        try{
            userReg.registerProfile("John", 13, "India");
        }catch(InvalidCountryException ice){
            System.out.println(ice.getMessage());
        }catch(InvalidAgeException iae){
            System.out.println(iae.getMessage());
        }
    }
}
```

Thank you

You have successfully completed

Exception Handling