

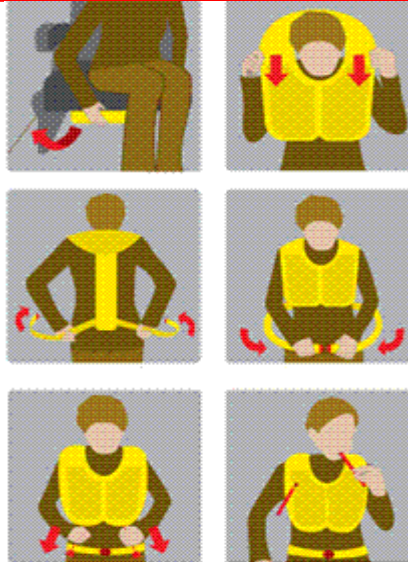
Exception Handling

- What are Exceptions
- try .. catch .. finally blocks
- Runtime Exceptions
 - Unchecked exceptions
 - Checked exceptions
- throws and throw keywords
- User-defined exceptions

Scenario



Meera is flying to NewYork



What are these?

This an air hostess giving a demonstration of steps that we have to take as passengers, in case of emergency.
Why this demonstration is important?
Why the air line staff insists on fastening our seat belts?

Exception Handling

Similarly, when we write programs as part of an application, we may have to visualize the challenges that can disrupt the normal flow of execution of the code.

Once we know what are the different situations that can disrupt the flow of execution, we can take preventive measures to overcome these disruptions.

In java, this mechanism comes in the form of Exception Handling.

What is an Exception?

- In procedural programming, it is the responsibility of the programmer to ensure that the programs are error-free in all aspects
- Errors have to be checked and handled manually by using some error codes
- But this kind of programming was very cumbersome and led to **spaghetti code**
- Java provides an excellent mechanism for handling runtime errors



What is an Exception? (Contd.).

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- The ability of a program to intercept run-time errors, take corrective measures and continue execution is referred to as exception handling

What is an Exception? (Contd.).

- There are various situations when an exception could occur:
 - Attempting to access a file that does not exist
 - Inserting an element into an array at a position that is not in its bounds
 - Performing some mathematical operation that is not permitted
 - Declaring an array using negative values

Uncaught Exceptions

```
class Demo {  
    public static void main(String args[]) {  
        int x = 0;  
        int y = 50/x;  
        System.out.println("y = " + y);  
    }  
}
```

Although this program will compile, but when you execute it, the Java run-time-system will generate an exception and displays the following output on the console :

```
java.lang.ArithmeticException: / by zero  
at Demo.main(Demo.java:4)
```

Uncaught Exceptions

What happens when you run the program without passing any command line arguments?

```
class ExceptionDemo {  
    public static void main (String args[]) {  
        System.out.println(args[0]);  
        System.out.println(args[1]);  
    }  
}
```

`java.lang.ArrayIndexOutOfBoundsException` exception is thrown at run time

Exception Handling Keywords

Exception-handling in Java

Java's exception handling is managed using the following keywords: **try**, **catch**, **throw**, **throws** and **finally**.

```
try {  
    statement(s)  
}  
catch (ExceptionType name) {  
    statement(s)  
}  
finally {  
    statement(s)  
}
```

- *A program can catch exceptions by using a combination of the try, catch, and finally blocks.*
- Program statements that you want to monitor for exceptions are contained within a try block .
- The catch block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- A single try block can have more than one catch blocks.
- The *optional* finally block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the try block.

How to Handle exceptions

```
class ExceptionDemo{
    public static void main(String args[]){
        int x, a;
        try{
            x = 0;
            a = 22 / x;

            System.out.println("This will be bypassed.");
        }
        catch (ArithmeticException e){
            System.out.println("Division by zero.");
        }
        System.out.println("After catch statement.");
    }
}
```

Handling Runtime Exceptions

- Whenever an exception occurs in a program, an object representing that exception is created and thrown in the method in which the exception occurred
- Either you can handle the exception, or ignore it
- In the latter case, the exception is handled by the Java runtime-system and the program terminates
- However, handling the exceptions will allow you to fix it, and prevent the program from terminating abnormally

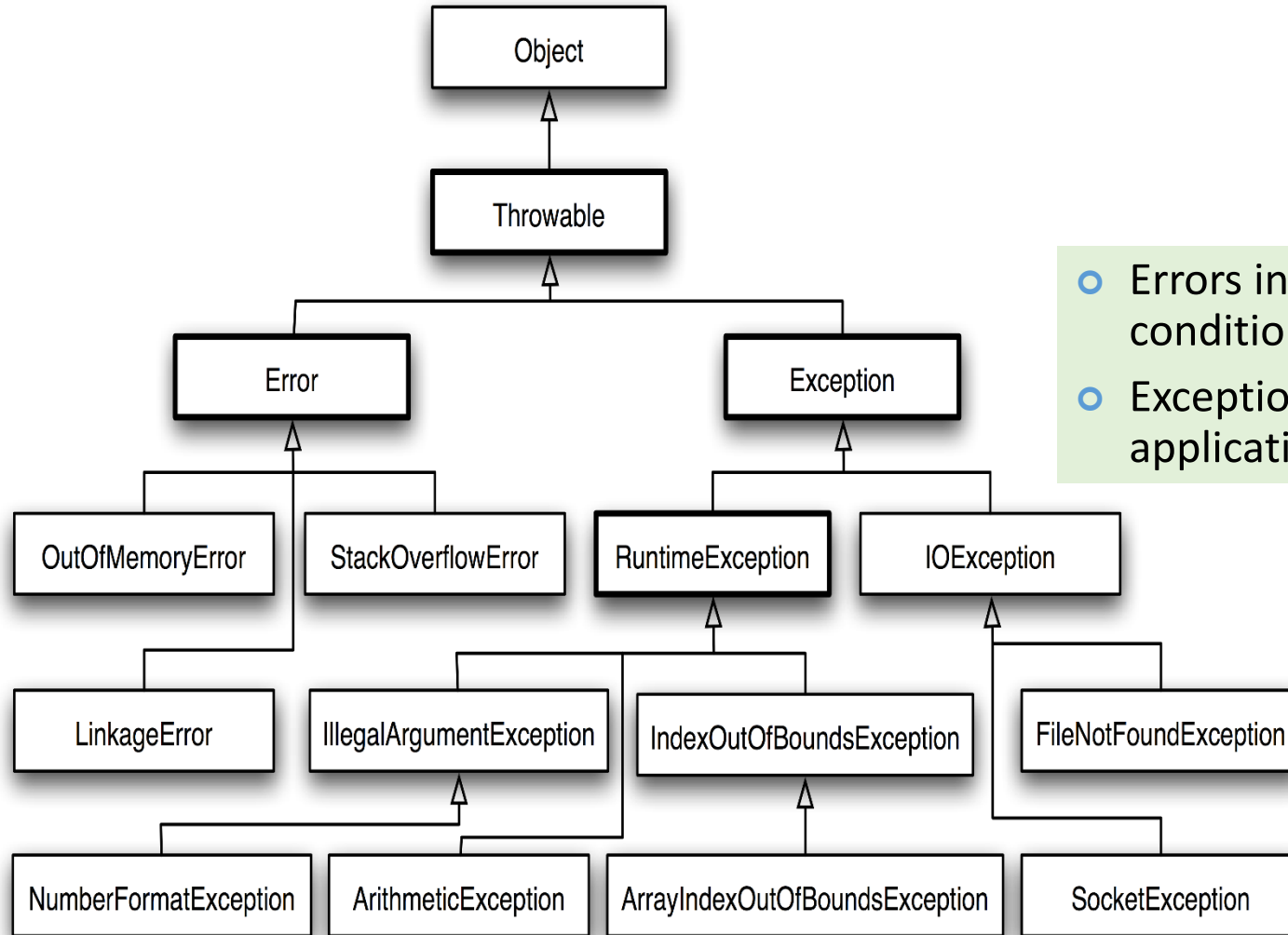
Handling Runtime Exceptions

- An **Exception** is a run-time error i.e. exception that occurs at the time of running the program.
- It is an event that occurs during the execution of a program that disrupts the normal flow of execution.
- **Exception Handler** is a set of instructions that handles an exception.
- The Java programming language provides a mechanism to help programs report and handle errors.
- When an exceptional condition arises, an ***object representing that exception*** is created and **thrown** in the method that caused the error.
- The method may choose to handle the exception ***itself, or pass it on***. Either way, at some point, the exception is **caught** and processed.

Exception Types

- There are several built-in exception classes that are used to handle the very fundamental errors that may occur in your programs
- You can create your own exceptions also by extending the **Exception** class
- These are called user-defined exceptions, and will be used in situations that are unique to your applications

The Exception Class Hierarchy



- Errors indicate serious problems and abnormal conditions that are out of programmer's reach.
- Exceptions are situations within the control of an application, that can be handled.

Exception types

Exceptions are of *two* types:

1. Unchecked Exceptions

- It is not mandatory to handle unchecked exceptions. **Java compiler does not check** to see if a method handles or throws these exceptions.
- **RuntimeException** is the super class for all the classes that come under unchecked exceptions.
- *ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException, NumberFormatException* etc. are some unchecked exceptions.

2. Checked Exceptions

- Checked exceptions are bound to be handled. Java compiler checks to see if a method handles or throws these exceptions. **If not handled, it leads to compilation error.**
- **Exception** is the super class for all the classes that come under checked exceptions.
- *ClassNotFoundException, IOException, FileNotFoundException, InterruptedException* etc. are some checked exceptions.

Execution Flow

- Once an exception object is thrown in a ***try*** block, control stops executing remaining statements of the ***try*** block and enters into its ***catch*** block.
- Once the ***catch*** statement is executed, program control executes the statements in finally block if present and then continues with the next line of the program **following** the ***try/catch*** block.
- A try block is always followed by a catch block, which handles the exception **or** a ***try*** block can be followed by ***finally*** block which is generally used to release the resources allocated in the try block.
- The catch block is not executed if no exception is thrown.

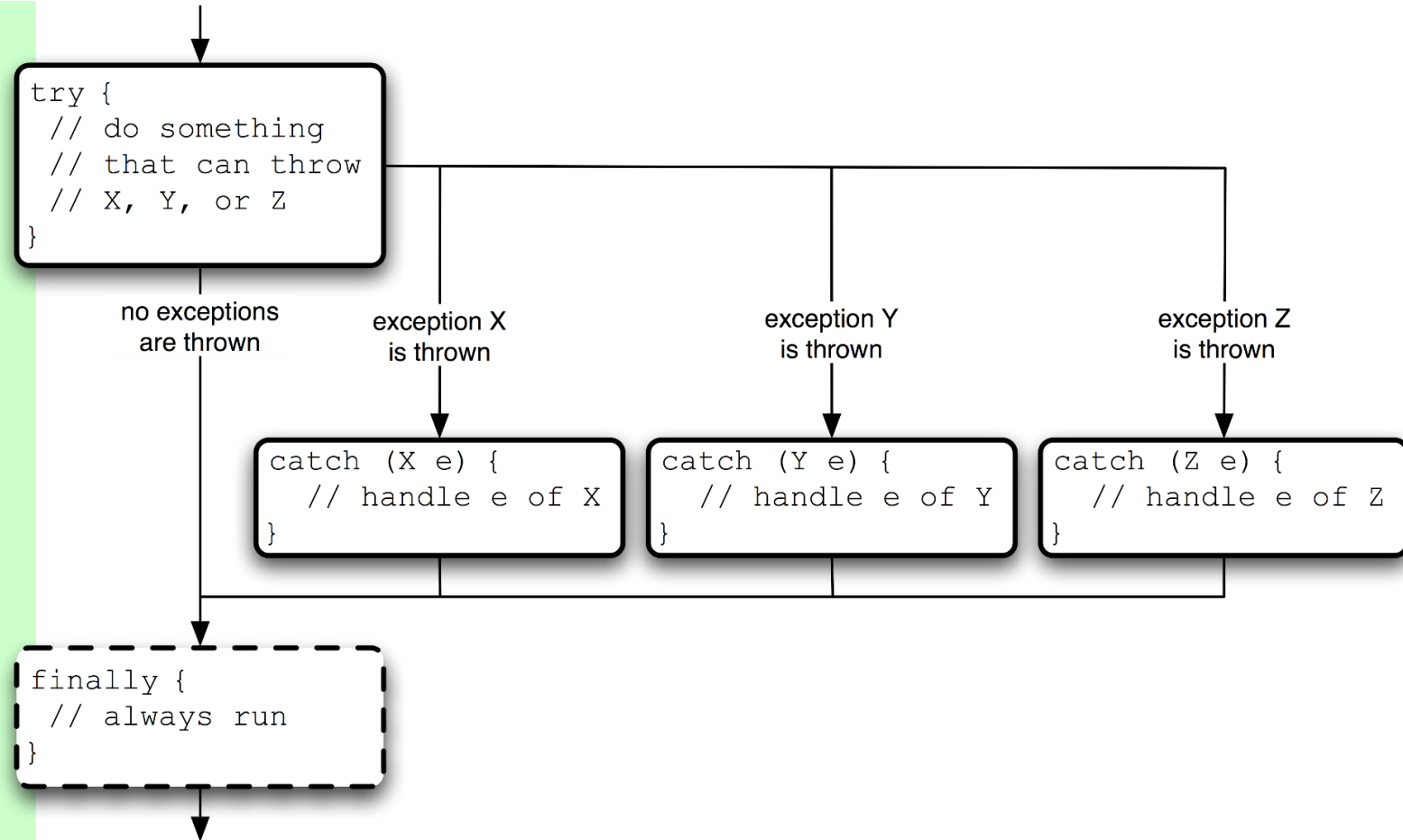
Handling Exceptions- Method I

There are *two* ways of handling exceptions:

- Handle within the method by *try .. catch* block

Method 1

```
try {  
    .....  
}  
catch (ExceptionTypeX e) {  
    // Exception handling code  
}  
catch (ExceptionTypeY e) {  
    // Exception handling code  
}  
catch (ExceptionTypeZ e) {  
    // Exception handling code  
}  
finally {  
    // clean-up code  
}
```



Handling Exceptions-Method II

Method 2

Method specifying that it *throws* the exception object to the caller rather than handling it.

```
public void someMethod() throws ExceptionTypeX, ExceptionTypeY {  
    // method body throws ExceptionTypeX and ExceptionTypeY  
}
```

Handling Un-Checked Exceptions

- Handling Unchecked exceptions is not mandatory for a compiler, but if not handled in the program, exception is thrown at run-time.

```
public class SumOfTwoIntegers {  
  
    public static void main(String[] args) {  
        try{  
            int number1=Integer.parseInt(args[0]);  
            int number2=Integer.parseInt(args[1]);  
            .....  
        }  
        catch(NumberFormatException nfe){  
            System.out.println("Invalid Data");  
        }  
    }  
}
```

If the exception is not handled, program will compile but following exception is thrown at run-time.

java.lang.ArrayIndexOutOfBoundsException

Not Handling Checked Exception

- Handling checked exceptions is mandatory.
- Java compiler returns an error and no class file is generated

```
public class BufferedReaderDemo {  
  
    public static void main(String[] args) {  
        String myName= getMyName();  
        System.out.println("My Name: "+ myName);  
    }  
  
    private static String getMyName() {  
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));  
        System.out.println("Enter your name: ");  
        String name = bufferedReader.readLine();  
        return name;  
    }  
}
```

Compile-time error

Handling Checked Exception: Method- I

```
public class BufferedReaderDemo {  
    public static void main(String[] args) {  
        String myName= getMyName();  
        System.out.println("My Name: "+ myName);  
    }  
    private static String getMyName() {  
        try{  
            BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));  
            System.out.println("Enter your name: ");  
            String name = bufferedReader.readLine();  
            return name;  
        }catch(IOException ioe){  
            System.out.println("Error while reading: \n"+ioe.getMessage());  
            return null;  
        }  
    }  
}
```

Handling Checked Exception: Method- II

```
public class BufferedReaderDemo1 {  
    public static void main(String[] args) throws IOException {  
        String myName= getMyName();  
        System.out.println("My Name: "+ myName);  
    }  
  
    private static String getMyName() throws IOException{  
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));  
        System.out.println("Enter your name: ");  
        String name = bufferedReader.readLine();  
        return name;  
    }  
}
```

Core Java

- Handling Multiple Exceptions
- Exception Life Cycle
- Nested try..catch blocks
- try.. with resources

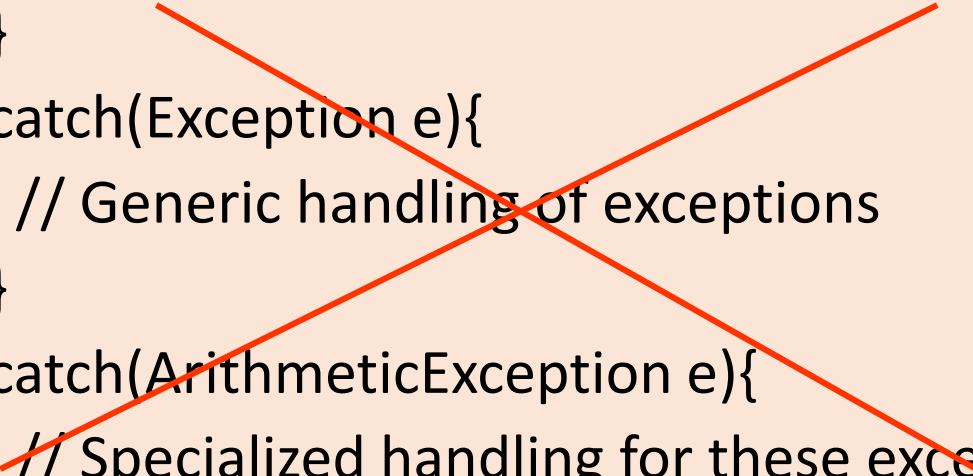
Handling Multiple Exceptions

Multiple Exceptions

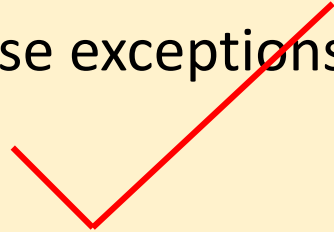
- If there are multiple exception classes that might arise in the **try** block, then several **catch** blocks are allowed to handle them separately, each handling different exception type.
- After one catch statement executes, the others are bypassed, and execution continues after the **try/catch** block.
- The catch blocks must be in sequence with the *most derived type first*, and the *most basic type last*. Otherwise, code will not compile

Handling Multiple Exceptions

```
try{  
    // try block code  
}  
catch(Exception e){  
    // Generic handling of exceptions  
}  
catch(ArithmeticException e){  
    // Specialized handling for these exceptions  
}
```



```
try{  
    // try block code  
}  
catch(ArithmeticException e){  
    // Specialized handling for these exceptions  
}  
catch(Exception e){  
    // Generic handling of exceptions  
}
```



Handling Multiple Exceptions in a single catch block: JDK 7 Onwards

```
public class ExceptionDemo {  
  
    public static void main(String[] args) {  
        try {  
            int x=Integer.parseInt(args[0]);  
            int y=Integer.parseInt(args[1]);  
            double average=divide(x,y);  
            System.out.println("Average Value: "+average);  
            String s1=null;  
            System.out.println(s1.length());  
        }catch(NumberFormatException | ArrayIndexOutOfBoundsException | ArithmeticException e ) {  
            e.printStackTrace();  
        }catch(Exception e) {  
            System.out.println("Exception Trapped in Exception Type");  
            e.printStackTrace();  
        }  
        System.out.println("End of main() method");  
    }  
  
    private static double divide(int x, int y) {  
        double z;  
        try {  
            z=x/y;  
            return z;  
        } catch (ArithmeticException e) {  
            //e.printStackTrace();  
        }  
        return 0;  
    }  
}
```

try...with resource

- The try-with-resources statement is a try statement that declares one or more resources.

Syntax:

```
try(<declare resource>){  
  
    } catch(...){  
  
    }
```

- A *resource* is an object that must be closed after the program is finished with it.
- The try-with-resources statement ensures that each resource is closed at the end of the statement.
- Any object that implements *java.lang.AutoCloseable*, which includes all objects which implement *java.io.Closeable*, can be used as a resource.
- Prior to Java SE 7, we use finally block to ensure that a resource is closed regardless of whether the try statement completes normally or abruptly.

try...with resource

Re-write the following code using try...with resources

```
private static String getName() {  
    String name="";  
    BufferedReader br=null;  
    System.out.println("Enter your name: ");  
    try {  
        br=new BufferedReader(new InputStreamReader(System.in));  
        name=br.readLine();  
    }  
    catch (IOException e) {  
        e.printStackTrace();  
    }  
    finally{  
        try {  
            br.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
    return name;  
}
```

Exception Call Stack

- After an exception object is created, it is handed off to the runtime system (*thrown*).
- The runtime system attempts to find a handler for the exception by backtracking the ordered list of methods that had been called.
 - This is known as the *call stack*.
- If a handler is found, the exception is *caught*. It is handled, or possibly re-thrown.
- If the handler is not found (the runtime backtracks all the way to the `main()` method), the exception stack trace is printed to the standard error channel (`stderr`) and the application aborts execution.

Call Stack Example

```
public class ExceptionCallStackDemo {
```

Input 20 and 0 as command-line arguments

```
    public static void main (String[] args) {  
        System.out.println(divideArray(args));  
    }
```

```
    private static int divideArray(String[] array) {  
        String s1 = array[0];  
        String s2 = array[1];  
        return divideStrings(s1, s2);  
    }
```

```
    private static int divideStrings(String s1, String s2) {  
        int i1 = Integer.parseInt(s1);  
        int i2 = Integer.parseInt(s2);  
        return divideInts(i1, i2);  
    }
```

```
    private static int divideInts(int i1, int i2) {  
        return i1 / i2;  
    }  
}
```

```
Exception in thread "main" java.lang.ArithmeticException: /  
by zero  
at  
com.Int.presentationtier.ExceptionDemo.divideInts(ExceptionDe  
mo.java:22)  
at com.  
Int.presentationtier.ExceptionDemo.divideStrings(ExceptionDem  
o.java:18)  
at com.  
Int.presentationtier.ExceptionDemo.divideArray(ExceptionDemo.  
java:12)  
at  
com.digitech.presentationtier.ExceptionDemo.main(ExceptionDem  
o.java:6)
```

Nested Try – Catch block

try-catch block within a try-catch block and so on are called as nested try-catch blocks.

Propagation of exception objects

If an inner try block does not have a matching catch statement for a particular exception, control is transferred to its outer try block's catch handler that is expected to contain matching catch statement. If found, the exception object is trapped and the control continues with the executable part of the outer try block

If not found, control will continue searching for appropriate catch handler in the next outer block(s) and if found the exception object will be trapped.

If none of the outer blocks have appropriate catch handler, then the exception object is thrown to Java run-time system.

Nested Try – Catch block example

Quiz Activity

```
public class NestedTryExample{
    public static void main(String args[]) {
        try {
            int a=args.length;
            int b= (int)20/a;
            System.out.println("b= "+ b);
            try {
                if(a==1)
                    a=a/(a-a);
                if(a==2) {
                    int c[]={1};
                    c[1]=2;
                }
            }
            catch(ArrayIndexOutOfBoundsException oe) {
                System.out.println("Array index out of bounds");
            }
            System.out.println("In outer try block");
        }
        catch(ArithmeticException ae) {
            System.out.println("Divide by 0"); }
    }
}
```

- **User-defined/Custom Exceptions**
 - Usage of *throw* keyword

User Defined/Custom Exceptions

- Java provides extensive set of in-built exceptions
- But there may be cases where we may have to define our own exceptions which are application specific.

For ex: If we have are creating an application for handling the database of eligible voters, the age should be greater than or equal to 18

In this case, we can create a user defined exception, which will be thrown in case the age entered is less than 18.

User defined exceptions contd..

Creating our own Exception classes is known as custom exceptions or user-defined exceptions.

Java custom exceptions are used to customize the exception according to user need.

Steps:

- Create a class which extends the **Exception** class for creating user-defined checked exception or create a class which extends **RuntimeException** for creating user-defined unchecked exception.
- The *toString()* method should be overridden in the user defined exception class in order to display meaningful information about the exception.
- Override *methods* of Throwable class (optional)

In the application

- Create an instance of the user-defined exception class.
- Use the **throw** keyword to throw the instance created.
 - **throw new <exception_classname(<parameters>)>**

User Defined/Custom Exceptions

Custom Exceptions should be checked or unchecked?

- If the exception is recoverable - it should be checked. If the exception is not recoverable and the program must halt - it should be unchecked.

Significance of `printStackTrace()` method

- We can use the `printStackTrace()` method to print the program's execution stack
- This method is used for debugging

`public void printStackTrace()` is a method of `Throwable` class.

`Throwable` is super class for all errors and exceptions but is not abstract class

```
public class Throwable extends Object implements Serializable
```

User defined exception: Example

```
package com.capgemini.it.service;

public class CredentialException_extends_Exception{
    private String name;
    public CredentialException() {

    }
    public CredentialException(String name) {
        this.name=name;
    }
    @Override
    public String getMessage() {
        return "Invalid Credentials";
    }
    @Override
    public String toString() {
        return "Userid: "+this.name+" is invalid";
    }
}
```

User defined exception: Example

```
package com.capgemini.it.ui;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

import com.capgemini.it.service.CredentialException;

public class CredentialTester {

    public static void main(String[] args) {
        try {
            getCredentials();
            System.out.println("Welcome to the App");
        } catch (CredentialException e) {
            System.out.println(e);
        }

    }
}
```


User defined exception: Example

```
private static void getCredentials() throws CredentialException{
try(
BufferedReader reader=
new BufferedReader(new InputStreamReader(System.in));
){
System.out.println("Enter userId: ");
String userid=reader.readLine();
if(!userid.equals("admin")) {
throw new CredentialException(userid);
}

}catch(IOException e) {

}
}
}
```

Re-throwing an exception object

A method can re-throw an exception object it has caught.

Mainly applied in layered architecture based applications.

A method in *data layer* may catch an exception and throw it to *service layer* which in turn re-throws to *UI layer*. The exception is handled in UI layer.

Custom Exception in Layered Architecture

```
package com.capgemini.it.service;

public class AdmissionCodeException extends
Exception{
    private String code="";
    public AdmissionCodeException() {

    }
    public AdmissionCodeException(String code) {
        this.code=this.code+code;
    }

    @Override
    public String toString() {
        return code;
    }

}
```

Custom Exception in Layered Architecture: data layer

```
public class StudentService {  
    public String getStudentName(String admissionCode) throws  
        AdmissionCodeException {  
        String [][] names= {  
            { "1101", "Ravi Kumar"},  
            { "1102", "Lakshmi"},  
            { "1103", "Madhavi"}  
        };  
  
        for(int i=0;i<names.length;i++) {  
            if(names[i][0].equals(admissionCode)) {  
                return names[i][1];  
            }  
        }  
        String error=admissionCode+"\nNo such admission code in our records";  
        throw new AdmissionCodeException(error);  
    }  
}
```

Custom Exception in Layered Architecture: service layer

```
public class StudentManager {  
    public String getStudentName(String admissionCode) throws  
        AdmissionCodeException{  
        String error="Admission code is null";  
        try {  
            StudentService service=new StudentService();  
            if(admissionCode !=null) {  
                return service.getStudentName(admissionCode);  
            }  
        }catch(AdmissionCodeException e) {  
            error= e+"\nErrors trapped in service layer";  
        }  
        throw new AdmissionCodeException(error);  
    }  
}
```

Custom Exception in Layered Architecture: UI layer

```
public class StudentTester {  
    public static void main(String[] args) {  
        String code="1108";  
        try {  
            String name=getStudentName(code);  
            System.out.println(name);  
        } catch (AdmissionCodeException e) {  
            System.out.println(e);  
        }  
    }  
}
```

```
private static String getStudentName(String code) throws AdmissionCodeException{  
    StudentManager manager=new StudentManager();  
    return manager.getStudentName(code);  
}  
  
}
```

Constructors and methods throwing exceptions in super class.

- Handling in sub classes.

Constructors & Methods throwing exceptions

```
public class SuperClass {  
    public SuperClass() throws java.io.IOException{  
        System.out.println("Hi");  
    }  
    public void show() throws RuntimeException{  
        System.out.println("I'm in super class show() method");  
    }  
}
```

```
public class SubClass extends SuperClass {  
    // or throws java.io.Exception  
    public SubClass() throws java.io.IOException  
        System.out.println("Welcome");  
    }  
    // valid , we can ignore throwing an exception  
    public void show() throws ArithmeticException{  
        super.show();  
        System.out.println("I'm in sub class show() method");  
    }  
}  
public static void main(String[] args) throws Exception{  
    new SubClass().show();  
}
```

If a **constructor method** in super class throws an **checked exception**, then the sub class constructor method should throw *same type of exception or its super type*.

If a **constructor method** in super class throws an **unchecked exception**, then the constructor method in the sub class has following options:

- Can throw same type of exception*
- Can throw its super type exception*
- Can ignore throwing an exception*

If a instance method in super class throws an **unchecked exception**, then the overridden method in the sub class has following options:

- Can throw same type of exception*
- Can throw its sub type exception*
- Can ignore throwing an exception*

If a instance method in super class throws an **checked exception**, then the overridden method in the sub class has to throw **same exception type or its sub-type**.

Constructors & Methods throwing exceptions

```
class Parent{  
protected int a;  
Parent() throws Exception{  
    System.out.println("Parent class");  
}  
  
public void getData() {  
    System.out.println("Hi");  
}  
  
public void showData(){  
    System.out.println("a= " + a);  
}}
```

```
class Child extends Parent{  
Child() throws Exception{  
    super();  
    System.out.println("Child class");  
}  
  
@Override  
public void getData() throws IOException{  
    BufferedReader bufferedReader =  
        new BufferedReader(new InputStreamReader(System.in));  
    System.out.println("Enter a number ");  
    int a = Integer.parseInt(br.readLine());  
}  
}
```

Note: public void getData() throws RuntimeException is valid

If *getData()* of super class does not declare *throws IOException* and if the overridden method in subclass declares, it leads to compile error.

The following exception is throw:

Exception IOException is not compatible with throws clause in Parent.getData()

Quiz

What will be the result, if we try to compile the following code (FileNotFoundException is a subclass of IOException)

```
import java.io.*;

class Super {
    void m1() throws FileNotFoundException {
        FileInputStream fx = new FileInputStream("Super.txt");
    }
}

class Sub extends Super {
    void m1() throws IOException {
        FileInputStream fx = new FileInputStream("Sub.txt");
    }
}
```

Yes, it will throw compilation Error

Quiz (Contd.).

What will be the result, if we try to compile the following code (FileNotFoundException is a subclass of IOException)

```
import java.io.*;
class Super {
    void m1() throws IOException {
        FileInputStream fx = new FileInputStream("Super.txt");
    }
}
class Sub extends Super {
    void m1() throws FileNotFoundException {
        FileInputStream fx = new FileInputStream("Sub.txt");
    }
}
```

No Compilation Error

Quiz (Contd.).

What will be the result, if we try to compile the following code

```
class Super {  
    void m1() throws ArithmeticException {  
        int x = 100, y=0;  
        int z=x/y;  
        System.out.println(z);  
    }  
}  
  
class Sub extends Super {  
    void m1() throws NumberFormatException {  
        System.out.println("Wipro");  
    }  
}
```

No Error! Compilation successful

Quiz (Contd.).

What will be the result, if we try to compile the following code (FileNotFoundException & SQLException are not related hierarchically)

```
import java.io.*;
import java.sql.*;
class Super {
    void m1() throws FileNotFoundException {
        FileInputStream fx = new FileInputStream("Super.txt");
    }
}
class Sub extends Super {
    void m1() throws SQLException {
        FileInputStream fx = new FileInputStream("Sub.txt");
    }
}
```

It will throw compilation Error



Thank You!