# Java Programming

Exception Handling

Module 6

# Agenda

1. Introduction to Exception Handling
2. Exception Handling Keywords
3. Checked and Unchecked Exceptions

# Objectives

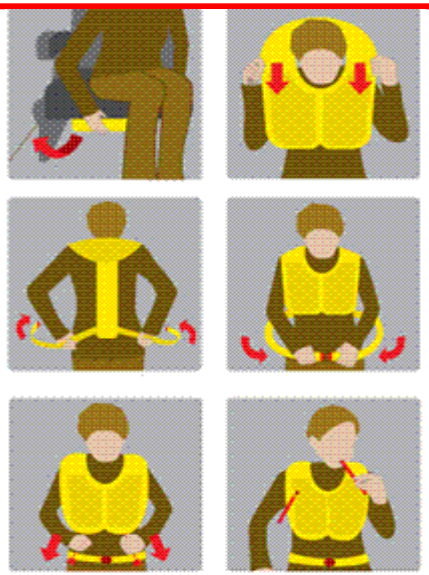
At the end of this module, you will be able to:

- Describe the exception handling mechanism of Java
- Describe the use of the keywords that comprise Java's exception handing mechanism
- Differentiate between checked and unchecked exceptions

# Introduction to Exception Handling

# Scenario

Meera is flying to NewYork



What are these?

# Scenario (Contd.).

The previous slide depicts 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?

# Scenario (Contd.).

You have to be aware of how tackle a situation in case of an emergency while you are flying aboard an aircraft .

You have to fasten your seat belts to protect yourself from mishaps that can occur during take off and landing.

This example demonstrates how you have to think in advance the many possibilities of mishaps that can occur and what are the preventive measures that can be taken.

# 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

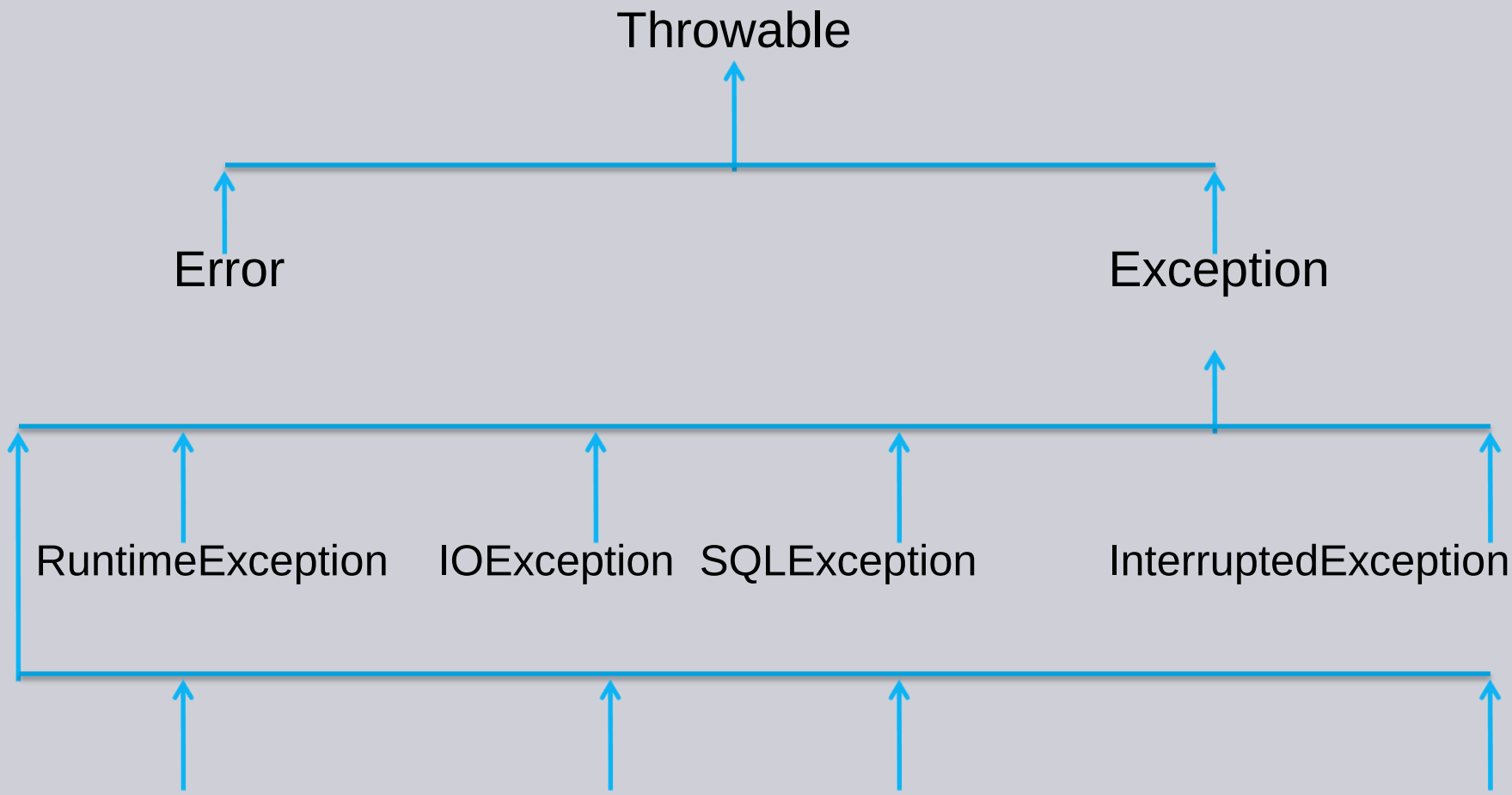
- The ability of a program to intercept run-time errors, take corrective measures and continue execution is referred to as exception handling

# 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

# Exception Types

Exceptions are implemented in Java through a number of classes. The exception hierarchy is as follows:




.

# 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)

# 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

# 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 run-time-system and the program terminates
- However, handling the exceptions will allow you to fix it, and prevent the program from terminating abnormally

# Exception Handling Keywords

# Exception Handling Keywords

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

```
try {
  // code comes here
  }
catch(TypeofException  obj) {
    //handle the exception
   }
       finally  {
            //code to be executed before the program ends
  }
```

# Exception Handling Keywords(Contd.).

- Any part of the code that can generate an error should be put in the **try** block
- Any error should be handled in the c**atch** block defined by the **catch** clause
- This block is also called the **catch block**, or the **exception handler**
- The corrective action to handle the exception should be put in the **catch** block

# How to Handle exceptions

```
class ExceptDemo{
  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.");
  }
}
```

# Quiz

- What will be the result, if we try to compile and execute the following code as

  java Ex1 Wipro Bangalore

```
Class Ex1 {
   public static void main(String[] xyz){
      for(int i=0;i<=args.length;i++)
       System.out.println(args[i]);
   }
  }
```

**It will compile successfully but will throw exception during runtime!**
**Why this exception is thrown?**

# Multiple Catch Statements

- A single block of code can raise more than one exception
- You can specify two or more **catch** clauses, each catching a different type of execution
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block

# Multiple Catch Statements (Contd.).

```
class MultiCatch{
 public static void main(String args[]){
  try{
     int l = args.length;
     System.out.println("l = " +l);
     int b = 42 / l;
     int arr[] = { 1 };
     arr[22] = 99;
  }
  catch(ArithmeticException e){
     System.out.println("Divide by 0: "+ e);
  }
```

# Multiple Catch Statements (Contd.).

```
   catch(ArrayIndexOutOfBoundsException e){
      System.out.println("Array index oob: "+e);
   }
   System.out.println("After try/catch
 blocks.");
  }
}
```

# Quiz

- What will be the result, if we try to compile and execute the following code as java Ex2 100

```
class Ex2 {
   public static void main(String[] args) {
     try {
       int i= Integer.parseInt(args[0]);
       System.out.println(i);
     }
     System.out.println("Wipro");
     catch(NumberFormatException e) {
       System.out.println(e);
     }
   }
}
```

**It will throw compilation Error**

## Multiple Catch Statements involving Exception Superclasses & Subclasses

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their exception superclasses
- This is because a catch statement that uses a superclass will catch exceptions of that type as well as exceptions of its subclasses
- Thus, a subclass exception would never be reached if it came after its superclass that manifests as an **unreachable code error**

# Quiz

- What will be the result, if we try to compile and execute the following code as java Ex2 100

```
class Ex2 {
   public static void main(String[] args) {
     try {
      int i= Integer.parseInt(args[0]);
      System.out.println(i);
     }
     catch(RuntimeException e) {
      System.out.println(e);
     }
     catch(NumberFormatException e) {
      System.out.println(e);}
  }
}
```

**It will throw compilation Error**

# Nested try Statements

- The **try** statement can be nested
- If an inner **try** statement does not have a **catch** handler for a particular exception, the outer block's catch handler will handle the exception
- This continues until one of the **catch** statement succeeds, or until all of the nested **try** statements are exhausted
- If no catch statement matches, then the Java runtime system will handle the exception

# Using throw

- System-generated exceptions are thrown automatically
- At times you may want to throw the exceptions explicitly which can be done using the **throw** keyword
- The exception-handler is also in the same block
- The general form of throw is:
  - throw **ThrowableInstance**
- Here, **ThrowableInstance** must be an object of type **Throwable**, or a subclass of **Throwable**

# Using throw (Contd.).

```
class ThrowDemo{
  public static void main(String args[]) {
    try {
        int age=Integer.parseInt(args[0]);
        if(age < 18)
          throw new ArithmeticException();
       else
          if(age >=60)
             throw new ArithmeticException("Employee is
  retired");
    }
  catch(ArithmeticException e) {
    System.out.println(e);
  }
  System.out.println("After Catch");
     }
}
```

# Match the following :

Match the content of Column A with the most appropriate content from column B :

| **Column A** | **Column B** |
|---|---|
| a) An exception is | a) Used to throw an exception explicitly |
| b) Throwable | b) Caused by Dividing an integer by zero |
| c) ArithmeticException is | c) An event that can disrupt the normal flow of instructions |
| d) Catch Block | d) This class is at the top of exception hierarchy |
| e) “throw” is | e) Exception Handler |

# Using throws

- Sometimes, a method is capable of causing an exception that it does not handle
- Then, it must specify this behavior so that callers of the method can guard themselves against that exception
- While declaring such methods, you have to specify what type of exception it may throw by using the **throws** keyword
- A **throws** clause specifies a comma-separated list of exception types that a method might throw:
  - type method-name( parameter list) throws exception-list

# Using throws (Contd.).

```
class ThrowsDemo{
  static void throwOne(){
    System.out.println("Inside throwOne.");
    throw new FileNotFoundException();
  }
  public static void main(String args[]){
    throwOne();
  }
}
```

***What happens when this code is compiled ?***

**Compilation Error……..why?**

# Implementing throws

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

# Using finally

- When an exception occurs, the execution of the program takes a non-linear path, and could bypass certain statements
- A program establishes a connection with a database, and an exception occurs
- The program terminates, but the connection is still open
- To close the connection, **finally** block should be used
- The finally block is guaranteed to execute in all circumstances

# Using finally (Contd.).

```
import java.io.*;
class FinallyDemo{
  static void funcA() throws
  FileNotFoundException{
    try{
       System.out.println("inside funcA(
  )");
       throw new FileNotFoundException( );
    }
    finally{
       System.out.println("inside finally
  of funA( )");
  }
}
```

# Using finally (Contd.).

```
  static void funcB(){
   try{
      System.out.println("inside funcB(
  )");
   }
   finally{
      System.out.println("inside finally
  of funB( )");
   }
  }
}
```

# Using finally (Contd.).

```
static void funcC() {
 try{
    System.out.println("inside funcC(
)");
 }
 finally{
    System.out.println("inside finally
of funcC( )");
 }
}
public static void main(String args[]){
 try{
    funcA();
 }
```

# Using finally (Contd.).

```
    catch (Exception e){
       System.out.println("Exception
  caught");
    }
    funcB( );
    funcC( );
  }
}
```

# Quiz

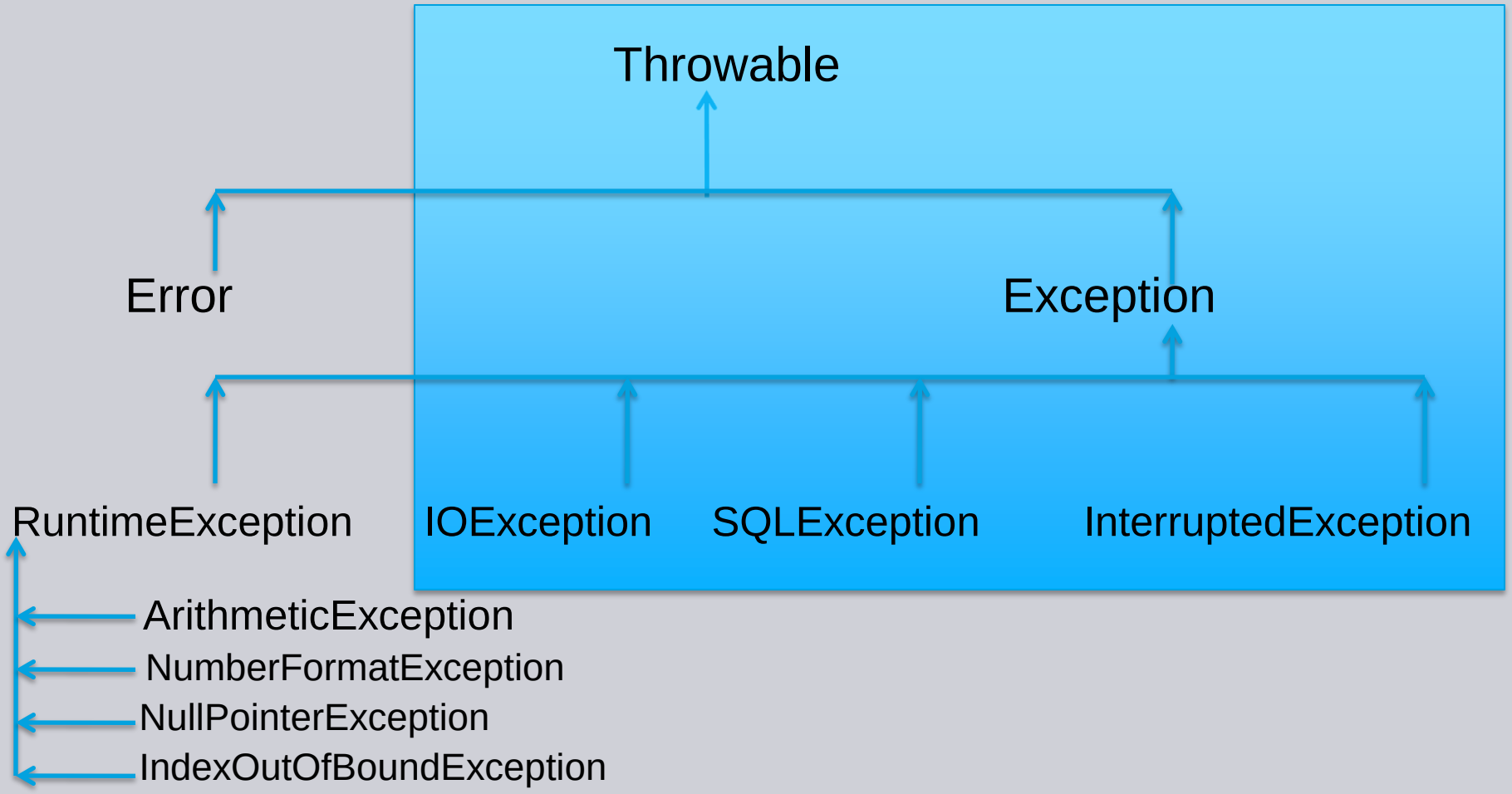
- What will be the result, if we try to compile and execute the following code as java Ex2 A

```
class Ex2 {
   public static void main(String[] args) {
     try {
      int i= Integer.parseInt(args[0]);
      System.out.println(i);
     }
     catch(NumberFormatException e) {
      System.out.println(e);
     }
     System.out.println("Exception Caught");
     finally { }
  }
}
```

**It will throw compilation Error**

# Checked and Unchecked Exceptions

# Checked Exceptions




.

# Checked Exception (Contd.).

- A checked exception is an exception that usually happens due to user error or it is an error situation that cannot be foreseen by the programmer

- A checked exception must be handled using a try or catch or at least declared to be thrown using throws clause. Non compliance of this rule results in a compilation error

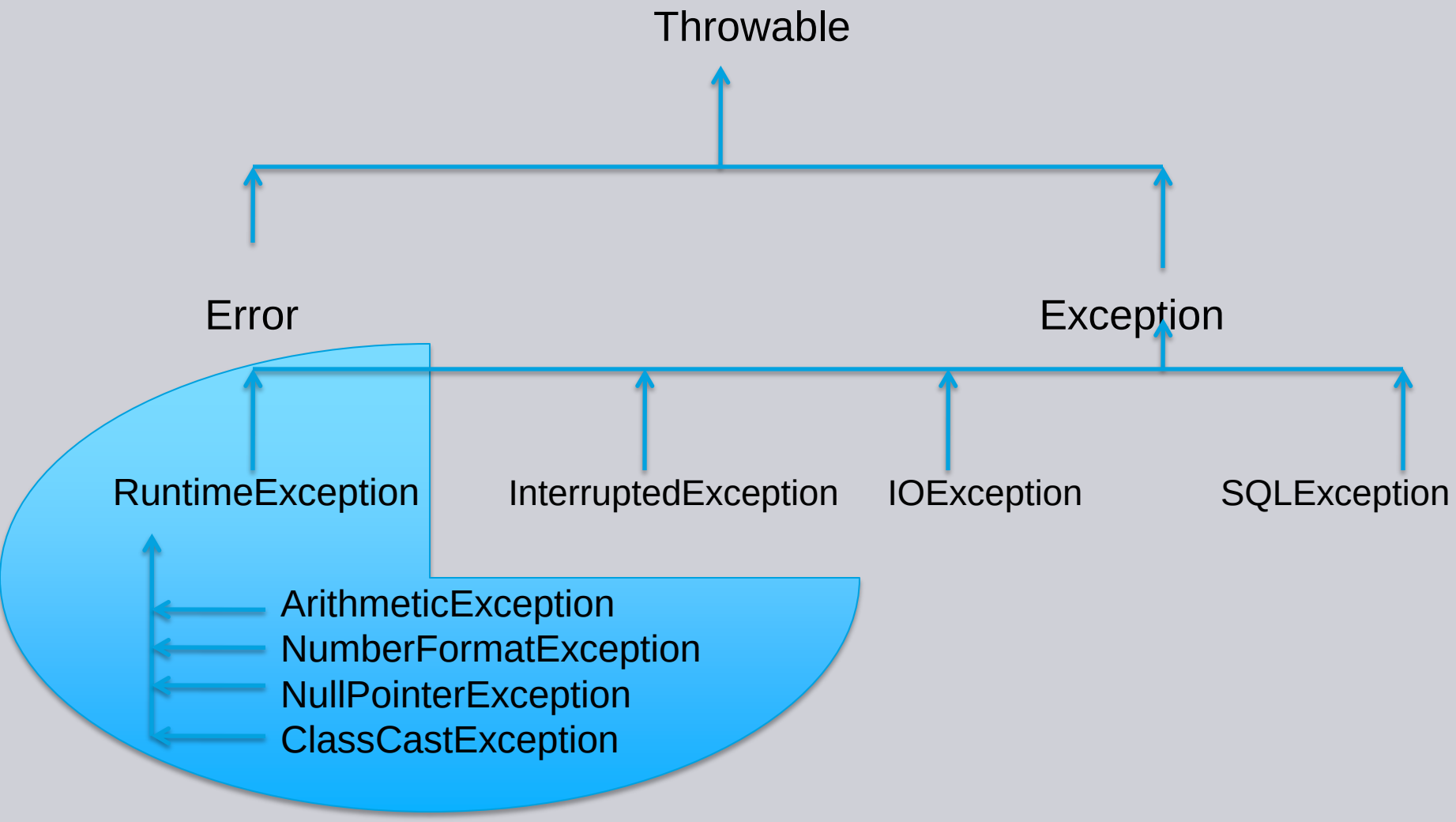
Ex: FileNotFoundException

If you try to open a file using

FileInputStream fx = new FileInputStream(“A1.txt”);

- During execution, the system will throw a FileNotFoundException, if the file A1.txt is not located, which may be beyond the control of a programmer

# Unchecked Exception




.

# Unchecked Exceptions (Contd.).

- An unchecked exception is an exception, which could have been avoided by the programmer
- The class RuntimeException and all its subclasses are categorized as Unchecked Exceptions
- If there is any chance of an unchecked exception occuring in the code, it is ignored during compilation

# Error

- Error is not considered as an Exception
- Errors are problems that arise beyond the control of the programmer or the user
- A programmer can rarely do anything about an Error that occurs during the execution of a program
- This is the precise reason Errors are typically ignored in the code
- Errors are also ignored by the compiler
- Ex : Stack Overflow

# Activity

Listed below are some of the built in exception classes. List them in the table below as per their classification, whether they are checked exceptions or unchecked exceptions :

1)NullPointerException  2) ClassNotFoundException  3) IOException
4) InterruptedException  5)ArrayIndexOutOfBoundsException
6) NumberFormatException  7)  ClassCastException  8)SQLException
9) IllegalAccessException  10) NegativeArraySizeException

| Checked Exceptions | Unchecked Exceptions |
|---|---|
| | |
| | |
| | |
| | |
| | |

# User Defined 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.).

- While creating user defined exceptions, the following aspects have to be taken care :
  - The user defined exception class should extend from the Exception class and its subclass
  - If we want to display meaningful information about the exception, we should override the toString() method

# Example on User Defined Exceptions

```
class  MyException extends Exception  {
  public MyException() {
   System.out.println("User defined
  Exception thrown");
  }
  public String toString() {
   return "MyException Object : Age cannot
  be < 18 " ;
  }
}
```

Contd..

# Example on User Defined Exceptions (Contd.).

```
class MyExceptionDemo{
  static int flag=0;
  public static void main(String args[]) {
   try {
      int age=Integer.parseInt(args[0]);
      if(age < 18)
         throw new MyException();
   }
   catch(ArrayIndexOutOfBoundsException e)
  {
      flag=1;
      System.out.println("Exception : "+ e);
   }
```

Contd..

# Example on User Defined Exceptions (Contd.).

```
    catch(NumberFormatException e) {
       flag=1;
       System.out.println("Exception : "+ e);
    }
    catch (MyExceptionClass e) {
       flag=1;
       System.out.println("Exception : "+ e);
    }
    if(flag==0)
       System.out.println("Everything is
  fine");
  }
}
```

# Significance of printStackTrace() method

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

# Example on printStackTrace() method

```
import java.io.*;
class PrintStackExample {
 public static void main(String args[])
 {
  try {
     m1();
  }
  catch(IOException e) {
     e.printStackTrace();
  }
 }
```

**contd..**

# Example on printStackTrace() method

```
  static void m1() throws IOException {
    m2();
  }
  static void m2() throws IOException {
    m3();
  }
  static void m3() throws IOException{
    throw new IOException();
  }
}
```

**Expected Output**

java.io.IOException
    at PrintStackExample.m3(PrintStackExample.java:24)
    at PrintStackExample.m2(PrintStackExample.java:20)
    at PrintStackExample.m1(PrintStackExample.java:16)
    at PrintStackExample.main(PrintStackExample.java:5)

# Rule governing overriding method with throws

- The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method

For eg: A method that declares(throws) an SQLException cannot be overriden by a method that declares an IOException, Exception or any other exception unless it is a subclass of SQLException

- In other words, if a method declares to throw a given exception, the overriding method in a subclass can only declare to throw the same exception or its subclass

- This rule does not apply for unchecked exceptions

# 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 Error! Compilation successful**

# 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**

# Summary

In this session, you were able to:

- Describe the exception handling mechanism of Java
- Describe the use of the keywords that comprise Java’s exception handing mechanism
- Differentiate between checked and unchecked exceptions

# References


1. Oracle (2012). *The Java Tutorials: Exceptions*. Retrieved March 12, 2012, from, http://docs.oracle.com/javase/tutorial/essential/exceptions/

2. Tutorial Point (2012). *The Java Tutorials: Exceptions*. Retrieved March 12, 2012, from, http://www.tutorialspoint.com/java/java_exceptions.htm

3. Schildt, H. Java: *The Complete Reference. J2SETM*. Ed 5. New Delhi: McGraw Hill-Osborne, 2005.

# Thank You