EXCEPTION HANDLING

Prepared by

Dr. Rajesh Kumar Ojha
Asst. Prof., CSE, Silicon University

Introduction

- Exception is a run-time error which arises during the execution of java program. The term exception in java stands for an **"exceptional event"**.
- So Exceptions are nothing but some abnormal and typically an event or conditions that arise during the execution which may interrupt the normal flow of program.
- An exception can occur for many different reasons, including the following:

  A user has entered invalid data.

  A file that needs to be opened cannot be found.

  A network connection has been lost in the middle of communications, or the JVM has run out of memory.

- “If the exception object is not handled properly, the interpreter will display the error and will terminate the program.
- Now if we want to continue the program with the remaining code, then we should write the part of the program which generate the error in the try{} block and catch the errors using catch() block..
- Exception turns the direction of normal flow of the program control and send to the related catch() block and should display error message for taking proper action. This process is known as **Exception handling**.”
- The purpose of exception handling is to detect and report an exception so that proper action can be taken and prevent the program which is automatically terminate or stop the execution because of that exception.

Java exception handling is managed by using five keywords: try, catch, throw, throws and finally.

- **Try:**

Piece of code of your program that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown.

- **Catch:**

Catch block can catch this exception and handle it in some logical manner.

- **Throw:**

System-generated exceptions are automatically thrown by the Java run-time system. Now if we want to manually throw an exception, we have to use the throw keyword.

Throws:

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

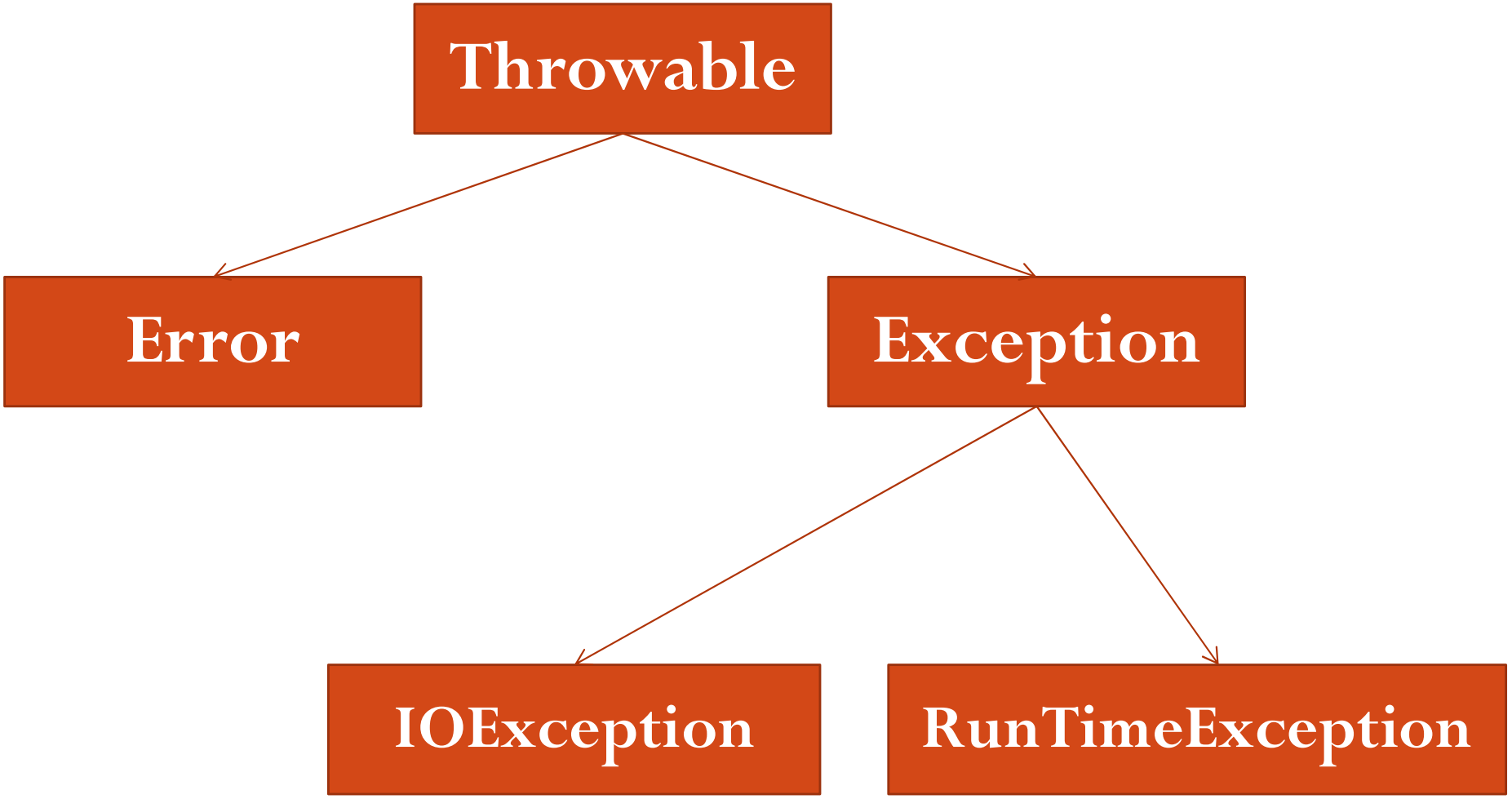
Finally:

- Any code that absolutely must be executed before a method returns, is put in a finally block.

General form:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1  e1) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 e2) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

Exception Hierarchy:



- All exception classes are subtypes of the java.lang.Exception class.
- The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors :

- These are not normally trapped form the Java programs.
- Errors are typically ignored in your code because you can rarely do anything about an error.
- These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment.

Classes of Exception

- IOException or Checked Exceptions class
- RuntimeException or Unchecked Exception class

IOException or Checked Exceptions :

- Exceptions that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*.
- For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Java's Checked Exceptions Defined in java.lang**

Exception	Description
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

RuntimeException or Unchecked Exception:

- Exceptions need not be included in any method's **throws** list. These are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.
- As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
- **Java's Unchecked RuntimeException Subclasses are given in the table ion the next slide**

Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered

Try and catch with example

```
public class TC_Demo
{
    public static void main(String[] args)
    {
        int a=10;
        int b=5,c=5;
        int x,y;
        try
        {
            x = a / (b-c);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by zero");
        }
        y = a / (b+c);
        System.out.println("y = " + y);

    }
}
```

o/p: Divide by zero and y=1

- Note that program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, and continues the execution, as if nothing has happened.
- If we run same program without try catch block we will not gate the y value in output. It displays the following message and stops without executing further statements.
- Exception in thread "main" java.lang.ArithmeticException: / by zero at Thrw_Excp.TC_Demo.main(TC_Demo.java:10)

Multiple catch blocks :

```
public class MultiCatch
{
    public static void main(String[] args)
    {
        int a [] = {5,10};
        int b=5;
        try
        {
            int x = a[2] / b - a[1];
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by zero");
        }
```

Multiple catch blocks :

```
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index error");
        }
        catch(ArrayStoreException e)
        {
            System.out.println("Wrong data type");
        }
        int y = a[1]/a[0];
        System.out.println("y = " + y);
    }
}
```

O/P: Array index error

Y=2

- Note that array element a[2] does not exist. Therefore the index 2 is outside the array boundry.
- When exception in try block is generated, the java treats the multiple catch statements like cases in switch statement.
- When you are using multiple catch blocks, it is important to remember that exception subclasses must come before any of their superclasses.

Nested try Statement

```
public class NestedTry
{
        public static void main(String args[])
        {
            int num1 = 100;
            int num2 = 50;
            int num3 = 50;
            int result1;
try
            {
                result1 = num1/(num2-num3);
                System.out.println("Result1 = " + result1);
```

```
try
        {
            result1 = num1/(num2-num3);
            System.out.println("Result1 = " + result1);
        }
        catch(ArithmeticException e)
        {
            System.out.println("This is inner catch");
        }
    }
    catch(ArithmeticException g)
    {
        System.out.println("This is outer catch");
    }
  }
}
```

O/P: This is outer Catch

Finally

- Java supports another statement known as **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements.
- The finally block is executed in all circumstances. Even if a try block completes without problems, the finally block executes.

```
public class Finally_Demo
{
    public static void main(String args[])
    {
        int num1 = 100;
        int num2 = 50;
        int num3 = 50;
        int result1;

        try
        {
            result1 = num1/(num2-num3);
            System.out.println("Result1 = " + result1);
        }
```

```
catch(ArithmeticException g)
        {
           System.out.println("Division by zero");

        }
        finally
        {
           System.out.println("This is final");
        }

    }
}
```

O/P: Division by Zero

This is final

throw

Our program can throw an exception explicitly, using the throw statement.

- **throw** ***ThrowableInstance;***

Note:

ThrowableInstance must be an object of type Throwable or a subclass ofThrowable.

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

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

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.

- A **throws clause** lists the types of exceptions that a method might throw.

```
type method-name(parameter-list) throws exception-list
{
  // body of method
}
```

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

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

User defined Exception subclass

- You can also create your own exception sub class simply by extending java **Exception** class.
- You can define a constructor for your Exception sub class (not compulsory) and you can override the **toString()** function to display your customized message on catch.

Points to Remember

1. Extend the Exception class to create your own ecxeption class.
2. You don't have to implement anything inside it, no methods are required.
3. You can have a Constructor if you want.
4. You can override the toString() function, to display customized message.

```
class MyException extends Exception {
private int ex; MyException(int a) {
ex=a;
}
public String toString() {
return "MyException[" + ex +"] is less than zero";
}
}
class Test {
static void sum(int a,int b) throws MyException {
if(a<0) {
throw new MyException(a);
}
```

```
else {
System.out.println(a+b);
}
 }
public static void main(String[] args) {
try { sum(-10, 10);
 } catch(MyException me) {
System.out.println(me);
}
}
}
```

Method Overriding with Exception Handling

- If super class method does not declare any exception, then sub class overriden method cannot declare checked exception but it can declare unchecked exceptions.

```
import java.io.*;
class Super {
void show() {
System.out.println("parent class");
}
}
public class Sub extends Super {
void show() throws IOException          //Compile time error
{
System.out.println("parent class");
}
public static void main( String[] args ) {
Super s=new Sub();
 s.show();
}
}
```

1. As the method **show()** doesn't throws any exception while in Super class, hence its overriden version also can not throw any checked exception.
2. If Super class method throws an exception, then Subclass overriden method can throw the same exception or no exception, but must not throw parent exception of the exception thrown by Super class method.
3. It means, if Super class method throws object of **NullPointerException** class, then Subclass method can either throw same exception, or can throw no exception, but it can never throw object of **Exception** class (parent of NullPointerException class).

Thank you