Programming Language II
CSE-215

Prof. Dr. Mohammad Abu Yousuf
yousuf@juniv.edu

Exception Handling-1

- 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.

What is exception:

- **Dictionary Meaning:** Exception is an abnormal condition.
- In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception-Handling Fundamentals

- An *exception is an* abnormal condition that arises in a code sequence at run time.
- In other words, an exception is a runtime error.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown in the method that caused the error.*

Exception-Handling Fundamentals

What is exception handling:

- Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.
- That method may choose to handle the exception itself, or pass it on.
- Either way, at some point, the exception is *caught and processed.*
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Exception-Handling Fundamentals

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. Let's take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Exception-Handling Fundamentals

- Java exception handling is managed via five keywords: **try, catch, throw, throws,** and **finally.**

- Program statements that you want to monitor for exceptions are contained within a **try block.**

- If an exception occurs within the **try block**, it is thrown. Your code can **catch this exception (using catch)** and handle it in some rational manner.

Exception-Handling Fundamentals

- System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword **throw.**

- Any exception that is thrown out of a method must be specified as such by a **throws** clause.

- Any code that absolutely must be executed after a **try block completes is put in a finally** block.

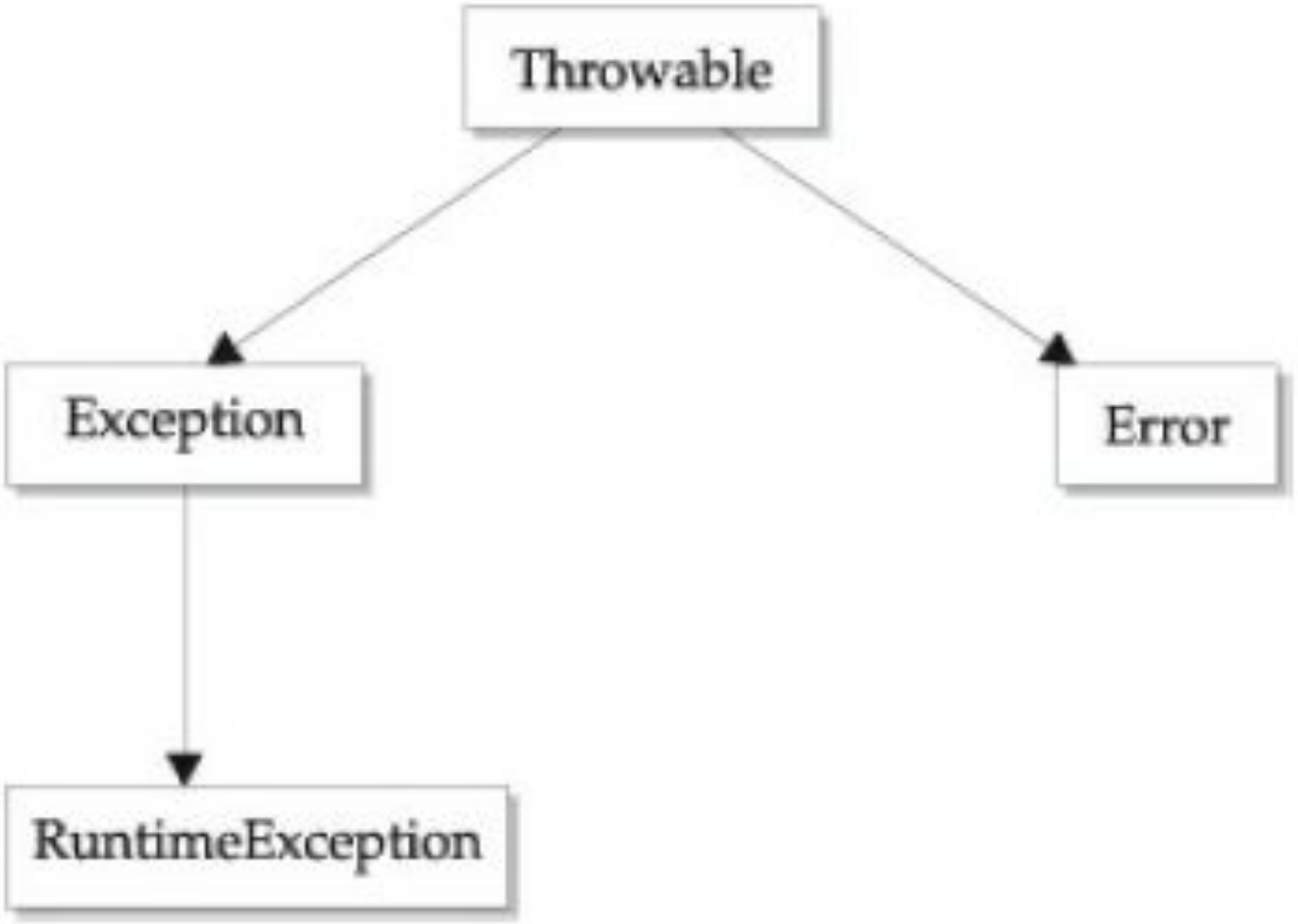
This is the general form of an exception-handling block:

```
try {
    // block of code to monitor for errors
}

catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}

catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
// ...
finally {
    // block of code to be executed after try block ends
}
```

Here, *ExceptionType is the type of exception that has occurred.*

Exception Types

- All exception types are subclasses of the built-in class **Throwable.**



Exception Types

- 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

Exception Types

1) **Checked Exception**

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.

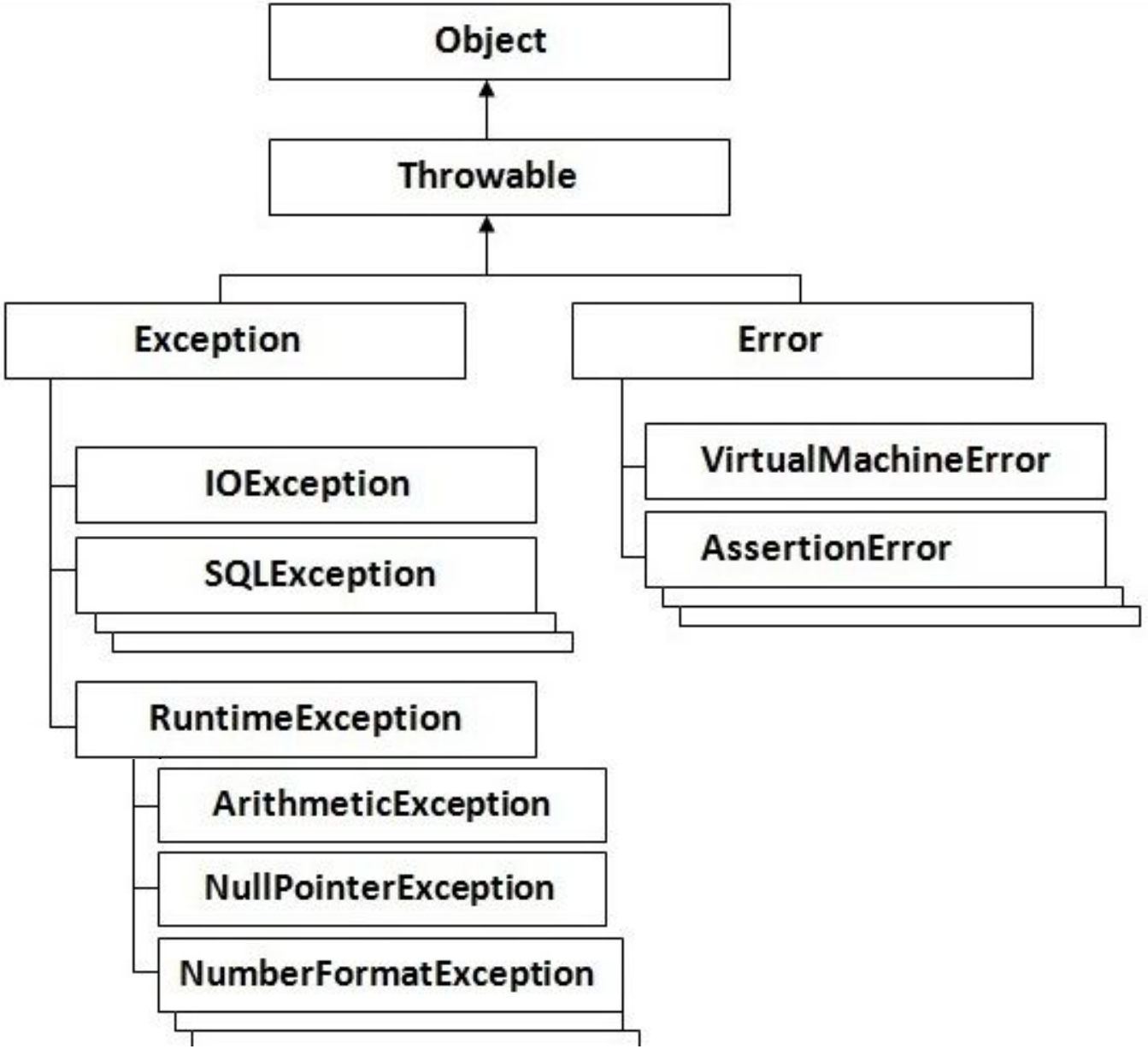
2) **Unchecked Exception**

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) **Error**

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Hierarchy of Java Exception classes



Problem without exception handling

- Let's try to understand the problem if we don't use try-catch block:

```
public class Testtrycatch1{
  public static void main(String args[]){
     int data=50/0;//may throw exception
     System.out.println("rest of the code...");
}
}
```

- Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

Problem without exception handling

- As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

- There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling of previous problem

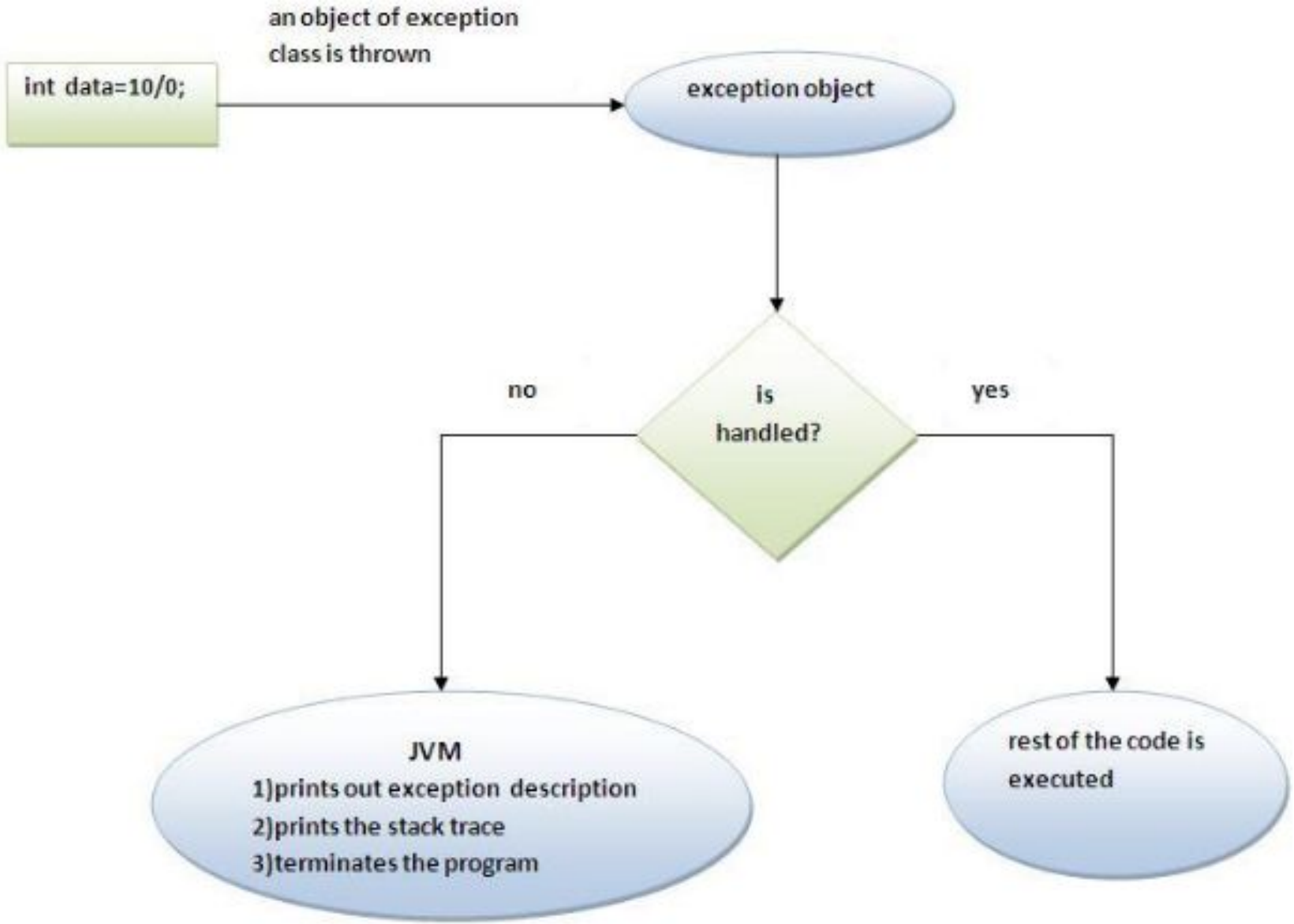
```
public class Testtrycatch2{
  public static void main(String args[]){
   try{
      int data=50/0;
   }catch(ArithmeticException e){System.out.println(e);}
   System.out.println("rest of the code...");
}
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Internal working of java try-catch block



Internal working of java try-catch block

- 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.

Using try and catch

- Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself.

- To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try block.**

- Immediately following the try block**, include a catch** clause that specifies the exception type that you wish to catch.

Example 1

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
     d = 0;
     a = 42 / d;
     System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
     System.out.println("Division by zero.");
    }

    System.out.println("After catch statement.");
  }
}
```

This program generates the following output:

Division by zero.

After catch statement.

Example 1

- Notice that the call to println( ) inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try block**

- Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try / catch mechanism.** into the catch block.

- A **try and its catch statement form a unit. The scope of the catch clause is restricted** to those statements specified by the immediately preceding **try statement.**

- **A catch** statement cannot catch an exception thrown by another **try statement.**

- The statements that are protected by **try must be surrounded by curly braces.**

Example 2

- In this example, program each iteration of the **for loop obtains two random integers. Those two** integers are divided by each other, and the result is used to divide the value 12345.

- The final result is put into **a. If either division operation causes a divide-by-zero error, it is** caught, the value of **a is set to zero, and the program continues.**

Example 2

```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();

    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    }
  }
}
```

Displaying a Description of an Exception

- You can display this description in a println( ) statement by simply passing the exception as an argument.
- For example, the catch block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
  System.out.println("Exception: " + e);
  a = 0; // set a to zero and continue
}
```

- When the program is run, each divide by- zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

Thank you