Exception handling

Vũ Thị Hồng Nhạn

(vthnhan@vnu.edu.vn)

Dept. of Software Engineering, UET

Vietnam National Univ., Hanoi

Errors

- 3 types of errors you most certainly face when building a program
- **Syntax errors**
  - violation of Java's grammatical rules
  - Java code won't even compile
- **Runtime errors**
  - Happens while the program is running
  - Might cause the program to crash
- **Bugs (logic errors)**
  - Program just doesn't do what you'd except

Subjectively wrong

Objectively wrong

Runtime error

- **Happens** sometimes **while** the program is running
- it's usually caused by **issues** like user *entering an invalid input* or *trying to open a file that doesn't exist*
- ➔ Most **common runtime errors** *are formalized* into something called **Exceptions**
- ➔ How to handle exceptions and how to make a program continue to execute?

Error vs. Exception

- An exception is **an unwanted** **or** **unexpected event,**
  - which occurs **during** the execution of a program (at run time) that disrupts *the normal flow* of the program's instructions
- **Error** indicates **serious problem** that a reasonable application should not try to catch
- **Exception** indicate conditions that a reasonable application might try to catch

Exceptions

- A **formal definition** of *a potential problem*
- E.g., a popular exception called **FileNotFoundException**
  - that appears whenever you *try to open a file* that doesn't exist
- **How** the exception appear and **where** do they come from?
  - They are thrown around between methods
  - It all starts when **a** method *tries to perform an operation* that is invalid when it realizes that it cannot
  - it creates **an exception object** of the relevant **exception class** and *throws* it to *whoever catches it*
  - Then those who catch it can either *throw it again* or simply *handle it gracefully*

Exceptions...

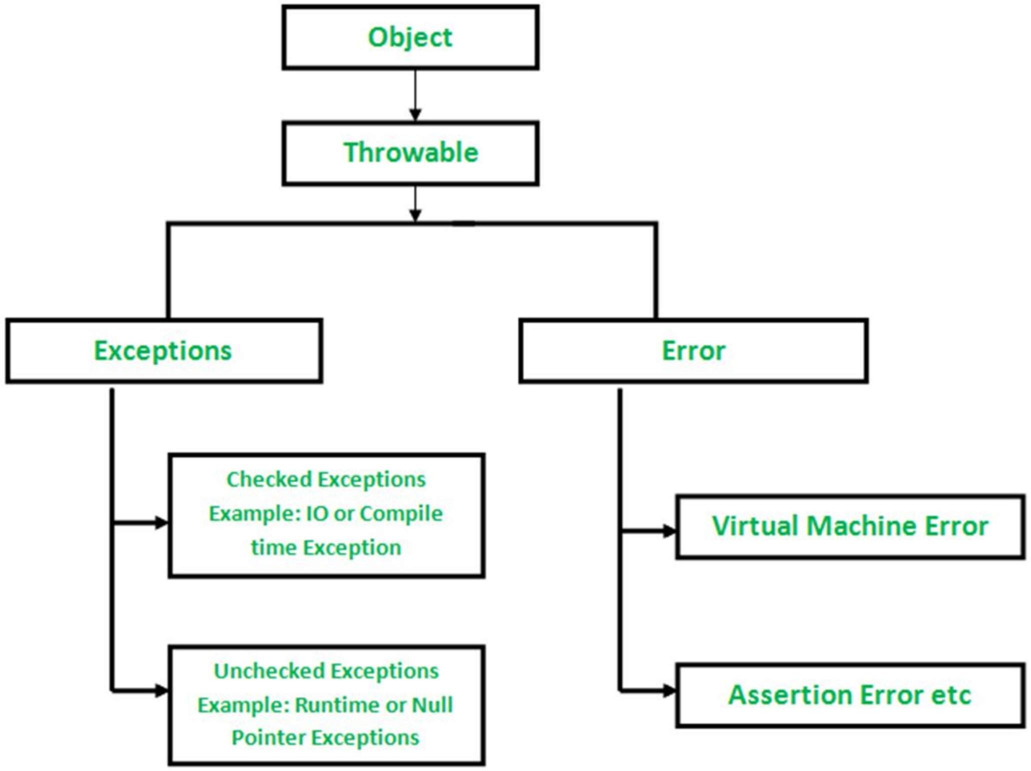
- **Methods** typically *communicate with each other* **using** input parameters and returning output results

- The way **methods** communicate **exceptions** with each other is by ***throwing*** *exceptions* if a method has *the potential of running* **into** *an invalid situation* like opening a file that might not exists
  - → *it might throw an exception*
  - This is done by adding a **throws** keyword followed by the **exception type** *when declaring that method*

```
public void openFile(String filename) throws FileNotFoundException{
        //open a file here
}
```

Exception hierarchy

- **Exception** class is used for exceptional conditions that user programs should catch
  - NullPointerException is an example of such an exception
- **Errors** are used by the Java run-time system (JVM)
  - To indicate errors *having to do with the run-time environment* itself (JRE)
  - StackOverflowError is an example of such an error


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

How JVM handles an exception?

- **Default** exception handling
  - Whenever **inside a method**, if *an exception* has occurred, the method creates an **object** known as Exception Object
  - And hands it off to the run-time system (JVM)
  - **The exception object** contains *name & description* of the exception and the *current state* of the program where exception has occurred
  - Creating *the Exception Object* and handling it to the run-time system is called throwing an Exception
- There might be a **list of the methods** that had been called to get to the method where exception was occurred
  - This ordered list of the methods is called **Call Stack**

Procedure

- The run-time system search the **Call Stack** to find the method that contains the *block of code that can handle* the occurred exception
  - The block of the code is called **Exception handler**
- The run-time system starts searching **from** *the method in which exception occurred,* proceeds through **Call Stack** in the reverse order in which methods were called
- If it finds appropriate handler then it passes the occurred exception to it
  - i.e., the type of the exception object thrown matches the type of the exception object it can handle

Procedure...

- If run-time system *searches all the method* on the **Call Stack** and *couldn't find* the appropriate handler,
  - then run-time system hand over the Exception Object to **default exception handler**, which is part of run-time system
  - This handler prints the exception information and terminates program **abnormally**

Example: no handler found

```java
class ThrowsExecp{
    public static void main(String args[]){
        String str = null;
        System.out.println( str.length() );
    }
}
```

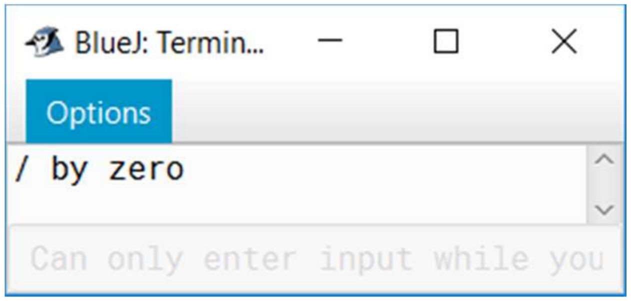
$$\overbrace{\text{java.lang.NullPointerException}}^{\text{Name of exception}}$$

$$\text{at } \underbrace{\text{ThrowsExecp.main(ThrowsExecp.java:6)}}_{\text{Description}}$$

Example: handler found

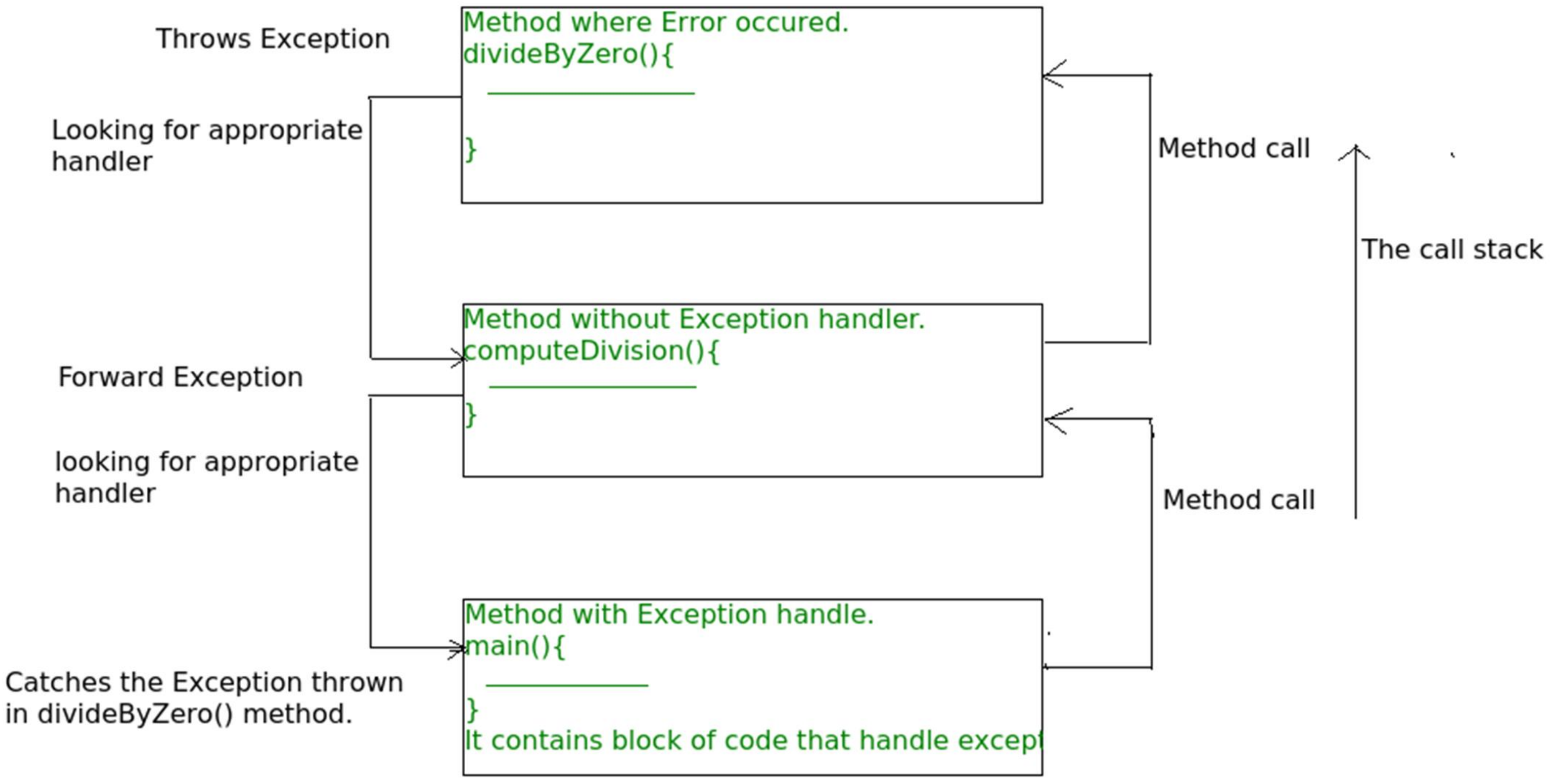
```
class ExceptionThrown{
        static int divideByZero(int a, int  b){
                int i=a/b;
                return i;
        }
        static int computeDivision(int a, int b){
                int res =0;
                try{
                        res = divdeByZero(a, b);
                }catch( NumberFormatException ex){
                        System.out.println(" NumberFormatException is occurred");
                }
                return res;
        }


        public static void main(String[] args){
             int a=1, b=0;
             try{
                     int i= computeDivision(a,b);
             }catch(ArithmeticException ex){ System.out.println(ex.getMessage()) }
        }
}
```



```
/ by zero
```

flow of Call Stack


Throws Exception
Looking for appropriate handler
Forward Exception
looking for appropriate handler
Catches the Exception thrown in divideByZero() method.
Method where Error occured.
divideByZero(){
}
Method without Exception handler.
computeDivision(){
}
Method with Exception handle.
main(){
}
It contains block of code that handle except
Method call
Method call
The call stack

The call stack and searching the call stack for exception handler.

How programmers handle exceptions

Customized exception handling

- 5 keywords are used in Java exception handling
  - **try**, **catch**, **throw**, throws, **finally**
- Statements that can raise **exceptions**
  - are contained within a **try** block
  - if *an exception* occurs within the **try** block → it's thrown
  - Your code can *catch* and *handle* this exception using **catch** block
- To manually throw an exception, use the keyword **throw**
- Any exception that is thrown *out of a method* must be specified by a throws clause
- Any code *that must be executed after a **try** block completes* is put in a **finally** block

Example

```
Class NoExceptionHandling{
        public static void main(String[] args){
                int[] A= new int[10];
                int i=A[10] //???
                System.out.println("Hello….I'm here to be executed!");
        }
}
```

- JVM terminates the program **abnormally**
- The last statement will **never be executed**
  - *To execute it* and to continue the *normal flow of the program*, **try-catch** clause must be included

try-catch clause

```
try{
        //block of code to monitor for errors
        //the code you think can rise an exception
}
catch(ExceptionType1 exObj){
        //exception handler for ExceptionTypes1
}
catch(ExceptionType1 exObj){
        //exception handler for ExceptionType2
}
//optional
finally{
        //block of code to be executed after try block ends
}
```

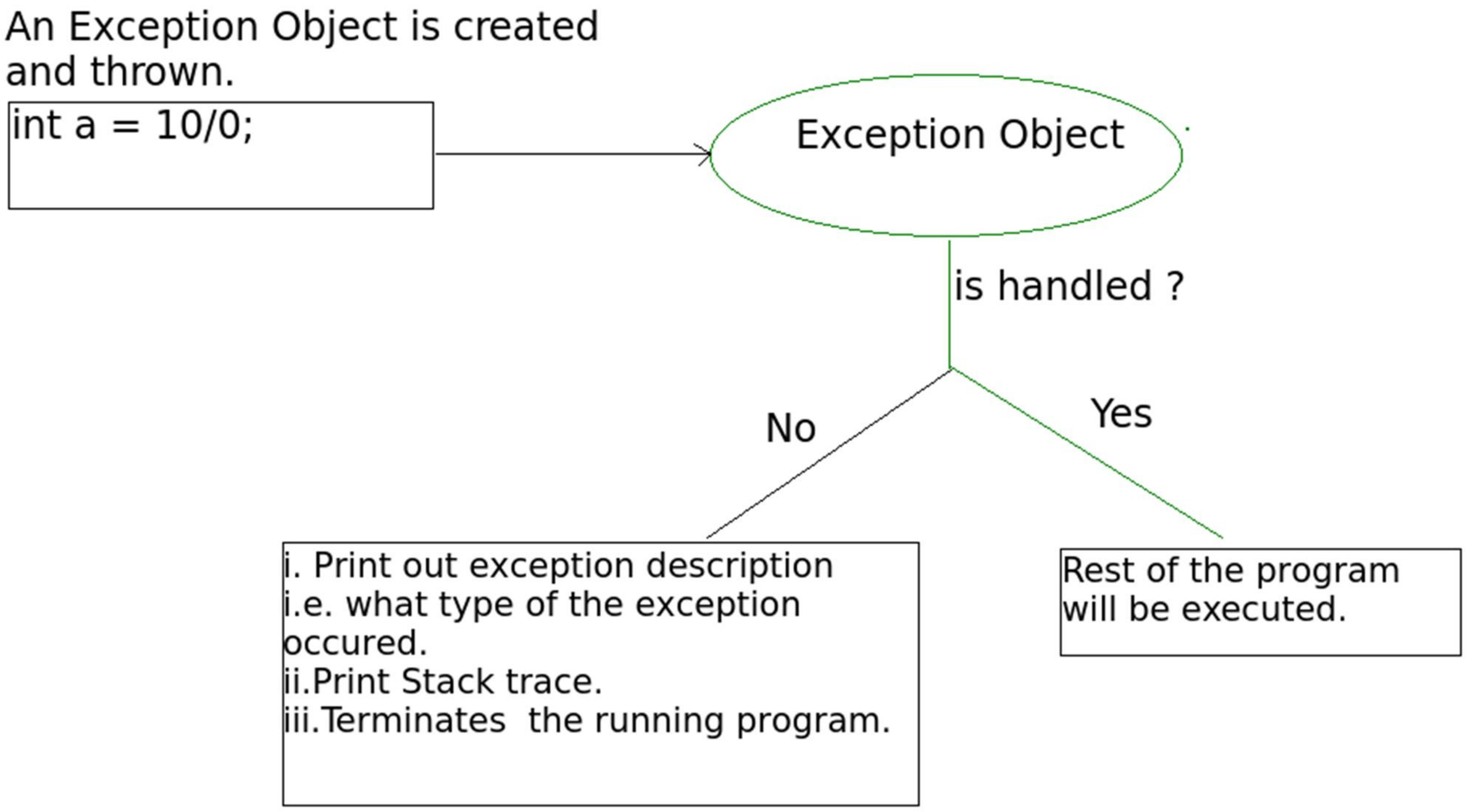
try-catch clause...

- In a method, more than one statements might throw **exceptions**
  - Put all of these statements within **try** block
  - & provide separate exception handler within a **catch** block for each exception

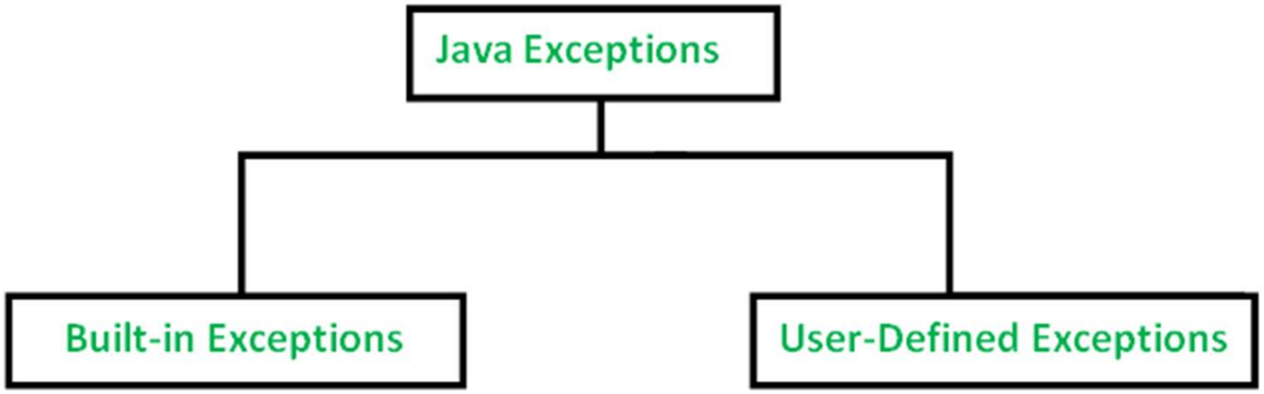
1. Each **catch** block is an exception handler
   - that handles the exception of the type indicated by its argument
   - ExceptionType must be the name of the class that inherits from **Throwable**
2. **finally** block is optional
   - it always get executed
   - Often used to put important codes like *closing the file* or *closing the connection*

Example...

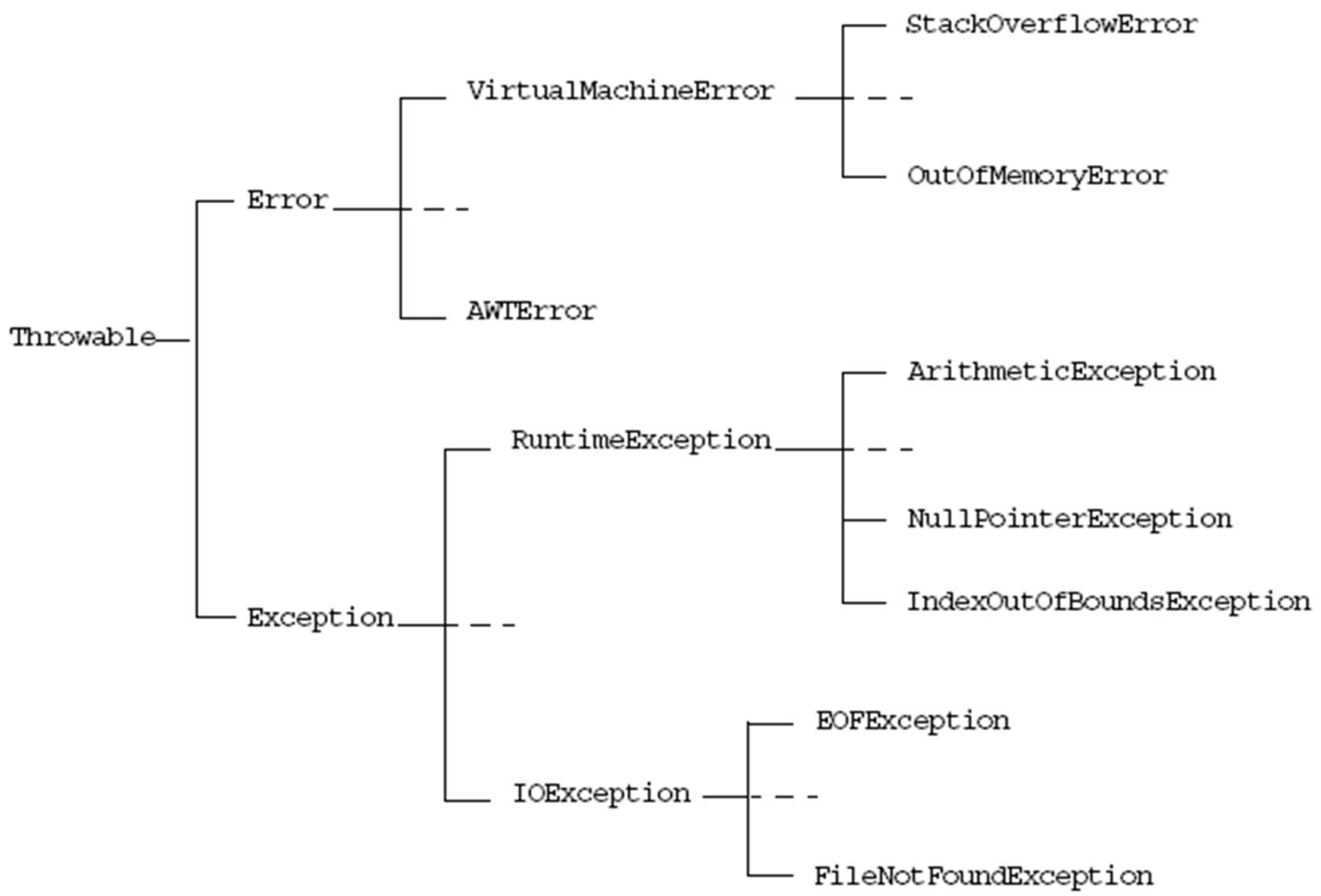
Summary


An Exception Object is created and thrown.
int a = 10/0;
Exception Object
is handled ?
No
Yes
i. Print out exception description i.e. what type of the exception occured.
ii.Print Stack trace.
iii.Terminates the running program.
Rest of the program will be executed.

Types of exception in Java



Hierarchy of exceptions in java



import built-in exceptions in Java

ArithmeticException	ArrayIndexOutofBoundException
ClassNotFoundException	FileNotFoundException
IOExeption	
NoSuchFieldException	NoSuchMethodException
NullPointerException	NumberFormatException (*cannot convert a string into a numeric format*)
StringIndexOutOfBoundException	RuntimeException(*any exception occurring during runtime*)

built-in exceptions in Java

Example 1

```
class StringIndexOutofBound_Ex{
    public static void main(String args[]){
        try{
            String s= “no fun with debugging”; //length=21
            char c= s.charAt(21); //accessing 22th element
            System.out.println(c);
        }
        catch(StringIndexOutOfBound e){
            System.out.println(e.get);
        }
    }
}
```

Output: String index out of range: 21

Example 2

```
class NumberFormatException_Ex{
    public static void main(String[] args){
        try{
          int num= Integer.parseInt(“abc”);
          System.out.println(num);
        }
        catch(NumberFormatException e){
          System.out.println(e.getMessage());
        }
    }
}
```

Output: Number format exception

User-defined exception (UDE)

- Used when built-in exceptions are unable to describe a certain situation
- All exceptions are subclass of Exception class, therefore…

```
class MyException extends Exception {
        MyException(String detail){ super(detail) }
}
```

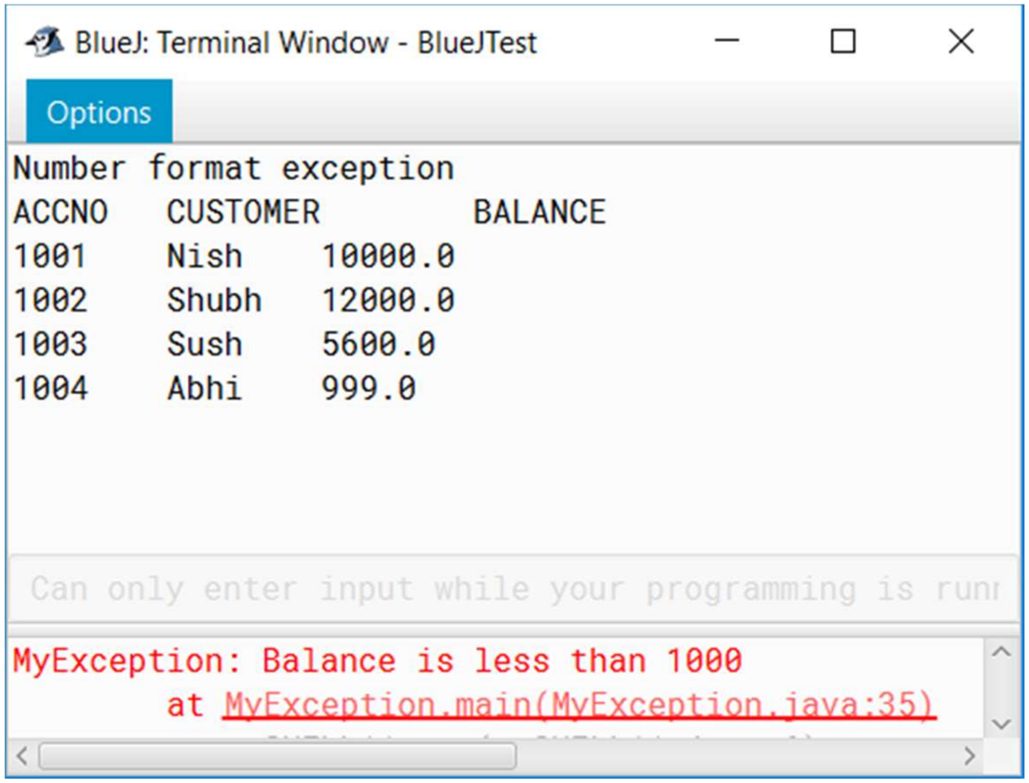
- To raise exception of UDE, we need to create an object to his exception class and throws it using **throw** clause

```
MyException e = new MyException("Exception Details");
throw e;
```

```
class MyException extends Exception{
        private static int         accNum[]  =  {1001, 1002, 1003, 1004};
        private static String    accName[]   =   {"Nish", "Shubh", "Sush", "Abhi“}
        private static double   balance[]   =  {10000.00, 12000.00, 5600.0, 999.00}
        MyException(){ }
        MyException(String s){ super(s); }

        public static void main(String[] args){
                try{
                        System.out.println(“ACCNO” +”\t” + “CUSTOMER”+”\t” + BALANCE);
                        for(int i=0;i<5;i++){
                                System.out.println(accNum[i] + ”\t” + accName[i] + ”\t” + balance[i]);
                                if(balance[i] < 1000){
                                        MyException e= new MyException(“Balance is less than 1000”);
                                        throw e;
                                }
                        }
                } catch(MyException e ){ e.printStackTrace()}
        }
}
```

Runtime error


BlueJ: Terminal Window - BlueJTest
Options
Can only enter input while your programming is runr

```
Number format exception
ACCNO    CUSTOMER        BALANCE
1001     Nish     10000.0
1002     Shubh    12000.0
1003     Sush     5600.0
1004     Abhi     999.0
```

```
MyException: Balance is less than 1000
        at MyException.main(MyException.java:35)
```

Checked vs. Unchecked Exception

Checked

- Are the exceptions that checked at compile time
- if some code **within** a method throws a checked exception
  - then, *the method* must **either** handle the exception **or** it must specify the exception using throws keyword
- Following program doesn't compile!

```
import  java.io.*;
class Main{
        public static void main(String[] args) {
                FileReader file = new FileReader("C:\\test.txt");//FileNotFoundException …
                BufferReader fileInput = new BufferReader(file);
                for(int i=0;i<2;i++) System.out.println(fileInput.readLine()); //IOException…
                fileInput.close(); //IOException must be caught, declared or thrown
        }
}
```

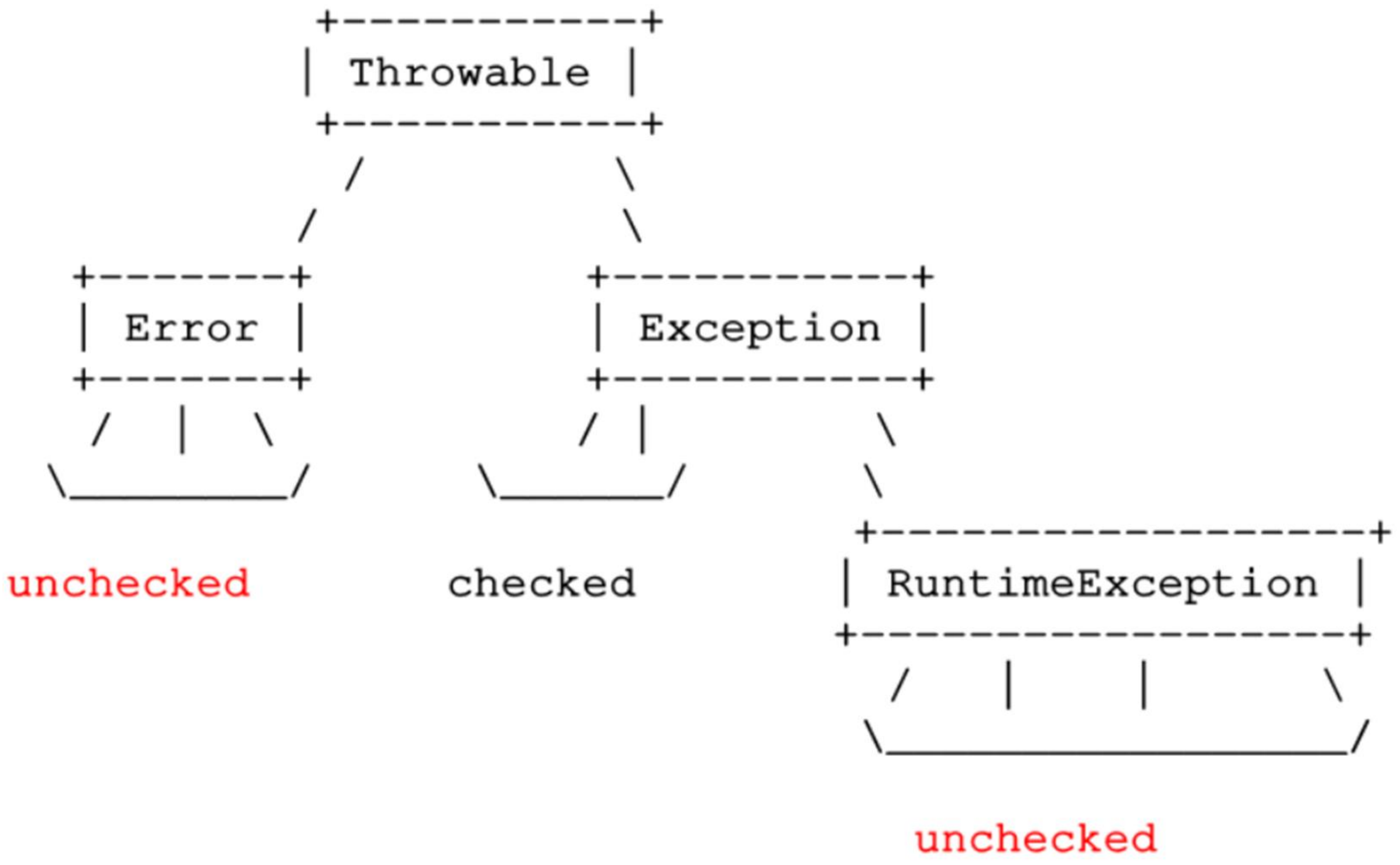
Checked...

- Need to either specify **list of exceptions** or use **catch-throw** block
- Choose the former and **throw** the list from the method using **throws**
  - Since FileNotFoundException is a subclass of IOException, so we just need to specify IOException in the throws list and make the program compiler-error-free

```
import  java.io.*;
class Main{
        public static void main(String[] args) throws IOException{
                FileReader file = new FileReader("C:\\test.txt");
                BufferReader fileInput = new BufferReader(file);
                for(int i=0;i<2;i++) System.out.println(fileInput.readLine());
                fileInput.close();
        }
}
```

Unchecked

- Are exceptions that are not checked at compiled time



Unchecked

- Following program compiles fine!
- but throws ArithmeticException when running

```
class Main{
    public static void main(String[] args){
        int x=10, y=0;
        int z = x/y;
    }
}
```

Unchecked → checked

```
int divide(int x, int y) throws Exception
{
    if (y==0)
        throw new Exception(”denominator = 0”);
    return x/y;
}
```

```
…
try {
    a = divide(x, y);
} catch(Exception e) {
    System.out.println(e.getMessage());
    …
}
```

- Once you’ve declared that a method throws an exception, Java forces you to **surround** that method with a **try** clause every time you try to call it

Catching base & derived classes as Exception

- If both **base** and **derived classes** are caught as exceptions
  - then **catch** block of derived class **must appear before** the base class
- E.g.,

```
class Base extends Exception{}
class Derived extends Base{}
public class Main{
    public static void main(String[] args){
        try{
            throw new Derived();
        }
        catch(Base b){}
        catch(Derived d){} //...
    }
}
```

switch them

This program cannot be compiled with error message

"*exception Derived has already been caught*"

String pool

String pool

- Is maintained by **String class**
- A storage in **heap** that stores string literals with the goal of decreasing the memory load and increasing the performance
- Known as String Intern Pool or String constant Pool

- When we create **a string literal**, JVM checks if that literal in the String pool
  - **If yes**, it returns a reference to the pooled instance
  - **Else**, a new String object takes place in the pool

E.g. creating String

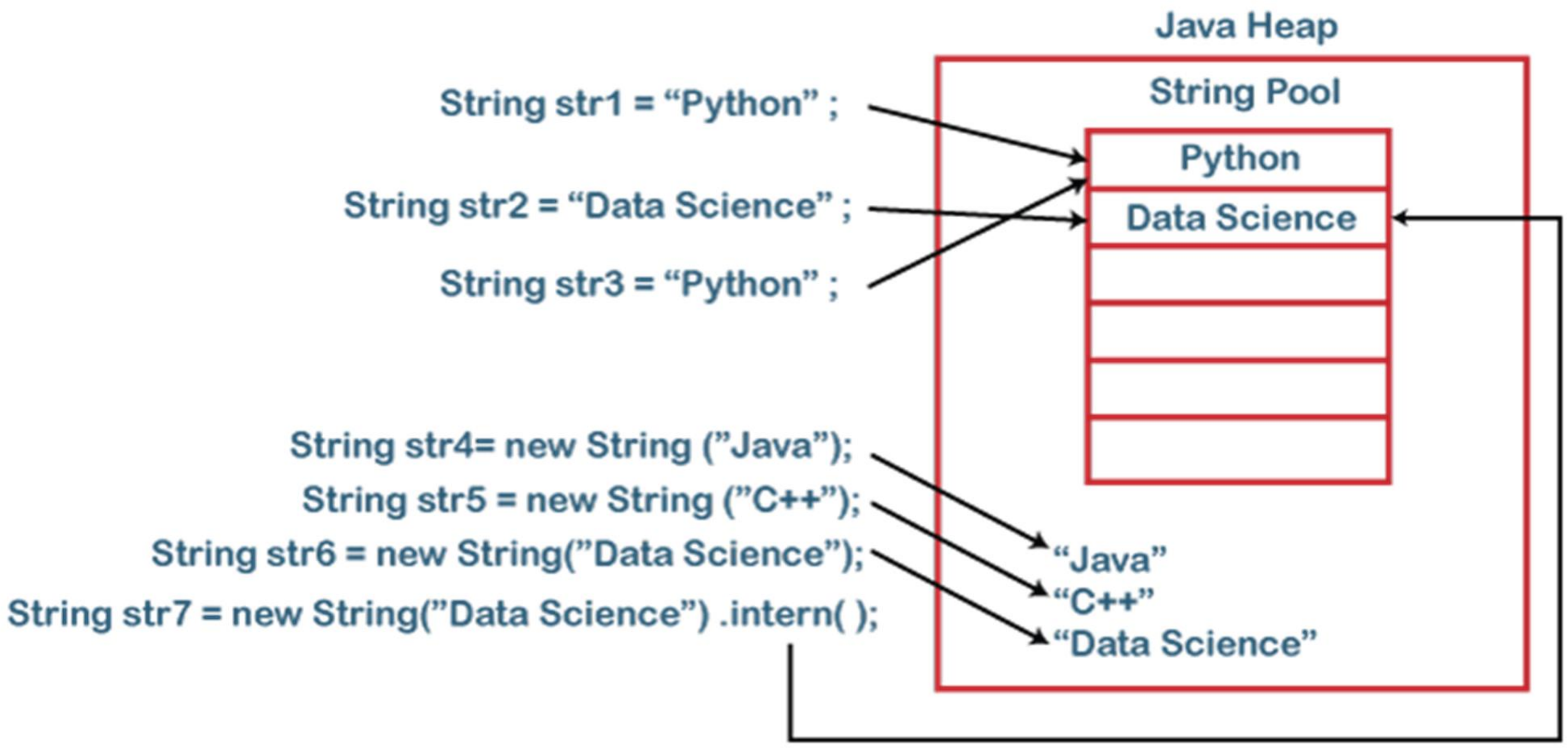
- Using **String literal**
  - String str1 = "Python";
  - String str2 = "Data Science";
  - String str3 = "Python";

- Using **new** keyword
  - String str4 = **new** String ("Java");
  - String str5 = **new** String ("C++");
  - String str6 = **new** String ("Data Science");

  - **→ creates a new string in the heap**

E.g.

String Pool Concept in Java



```
String str1 = “Python” ;
String str2 = “Data Science” ;
String str3 = “Python” ;
String str4= new String (”Java”);
String str5 = new String (”C++”);
String str6 = new String(”Data Science”);
String str7 = new String(”Data Science”) .intern( );
```

String.intern()

- Using **new keyword** creates a new string in **the heap**
  - We can stop by using the **intern()**

  - String str7 = new String(“Data Science").intern();
- Method intern() puts the string in the String pool or refers to another String object from pool having the same value
- It returns a string from the pool if the string pool already contains a string equal to the String object
- If the string is not already existing, the String object is added to the pool, and a reference to this String object is returned.

E.g.

- String str1 = "Python";
- String str3 = "Data Science";
- String str2 = "Python";
- String str4 = new String(“Python").intern();

- System.out.println((str1 == str2)+", equal."); // true
- System.out.println((str1 == str3)+", not equal."); // false
- System.out.println((str1 == str4)+", equal."); // true