



EXCEPTION HANDLING IN JAVA

Vijay Kumar Meena
Assistant Professor
KIIT University

WHAT WE HAVE STUDIED SO FAR?

- Programming Paradigms
- Introduction to OOP
- Basics of Java
- OOP in Java
- Inheritance in Java
- Packages and Interfaces in Java
- Inner classes in Java
- String Handling in Java

WHAT IS AN EXCEPTION?

- An exception is an runtime error which occurs whenever a program tries to do something abnormal like dividing by zero, accessing array element which doesn't exist, etc.
- When an exception occurs, the program can handle it in various ways such as by catching the exception and taking some action or by allowing program to terminate with an error message.

J Demo.java X

J Demo.java

```
1  class Demo{
2      public static void main(String[] args){
3          int a = 4 / 0;
4          System.out.println("a = " + a);
5      }
6  }
```

PROBLEMS

DEBUG CONSOLE

TERMINAL

OUTPUT

PORTS



vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Handling\$ javac Demo.java

vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Handling\$ java Demo

Exception in thread "main" java.lang.ArithmeticException: / by zero
at Demo.main(Demo.java:3)

vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Handling\$

EXCEPTIONS IN JAVA

- In Java, an exception is an *object* of a corresponding exception class.
- When an error occurs in a method, it throws out an exception object that contains the information about where the error occurred and type of the error.
- The error (exception) object is passed on to the runtime system, which searches for the appropriate code that can handle the exception. This exception handling code is called ***exception handler***.
- Exception handling is a mechanism that is used to handle runtime errors such as *ClassNotFoundException* or *ArrayOutOfBoundsException*, etc. If runtime system does not find any exception handler for particular type of exception then it terminates the program with error.
- This ensures that the normal flow of application is not disrupted and program execution proceeds smoothly.

EXCEPTIONS IN JAVA

- Java exception handling is managed via five keywords -
- **try:** Program statements that you want to monitor for exceptions are contained within a *try* block.
- **catch:** Your code can catch the exception (using *catch*) and handle it in some rational manner.
- **throw:** To manually throw an exception. System-generated exceptions are automatically thrown by the Java runtime system.
- **throws:** Specifies which exceptions a given method can throw. It is used in method declaration.
- **finally:** Specifies any code block that must be executed whether or not an exception occurs.

EXCEPTION HANDLING IN JAVA

- The general form of an exception-handling block -

```
try {  
    // block of code to monitor for errors.  
}  
catch (ExceptionType1 exObj) {  
    // exception handler for ExceptionType1  
}  
catch (ExceptionType2 exObj) {  
    // exception handler for ExceptionType2  
}  
// ... You can add any number of catch blocks like this  
finally {  
    // block of code to be executed after try-catch block ends  
}
```

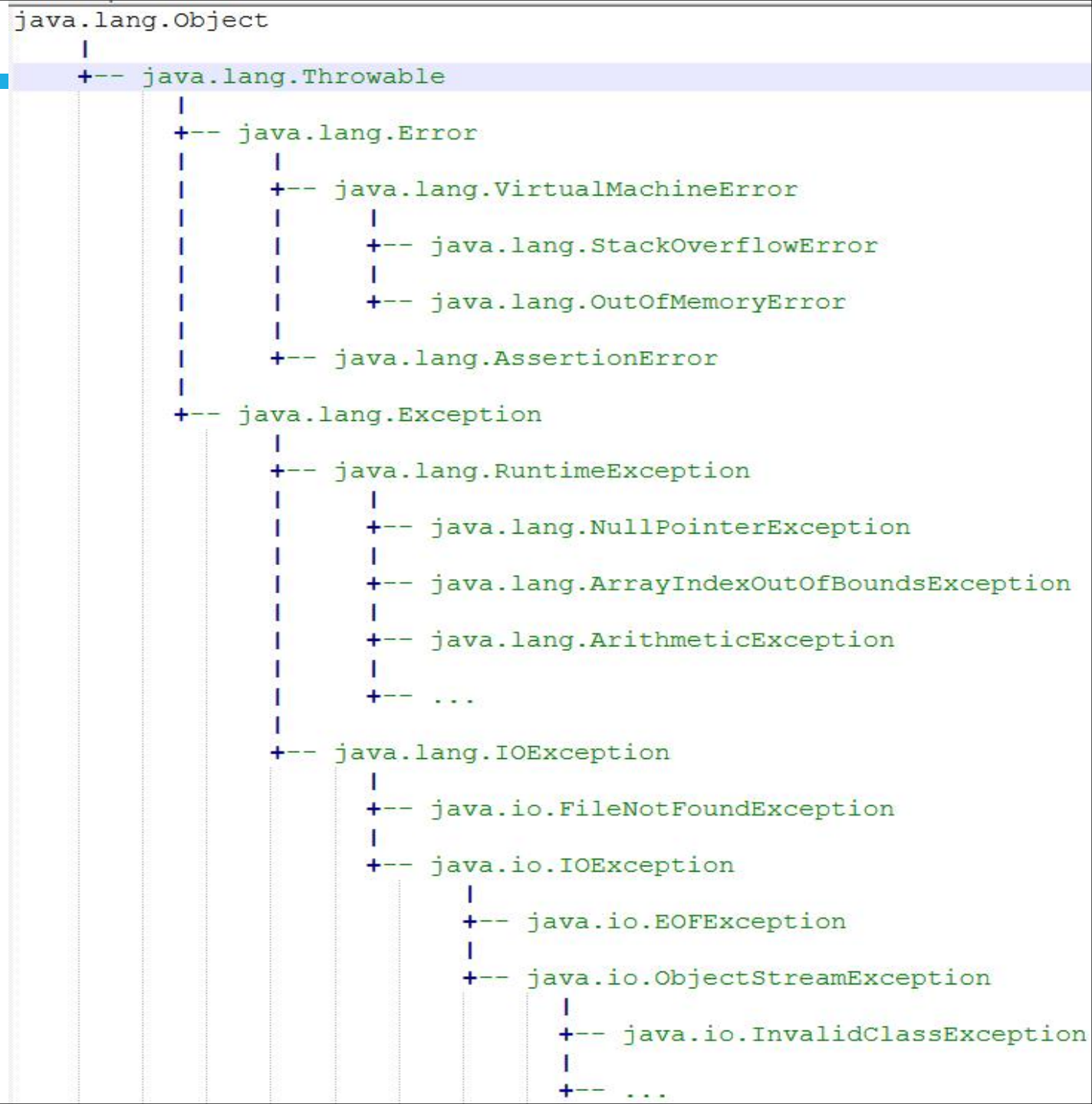
- *ExceptionType* is the type of exception that has occurred.

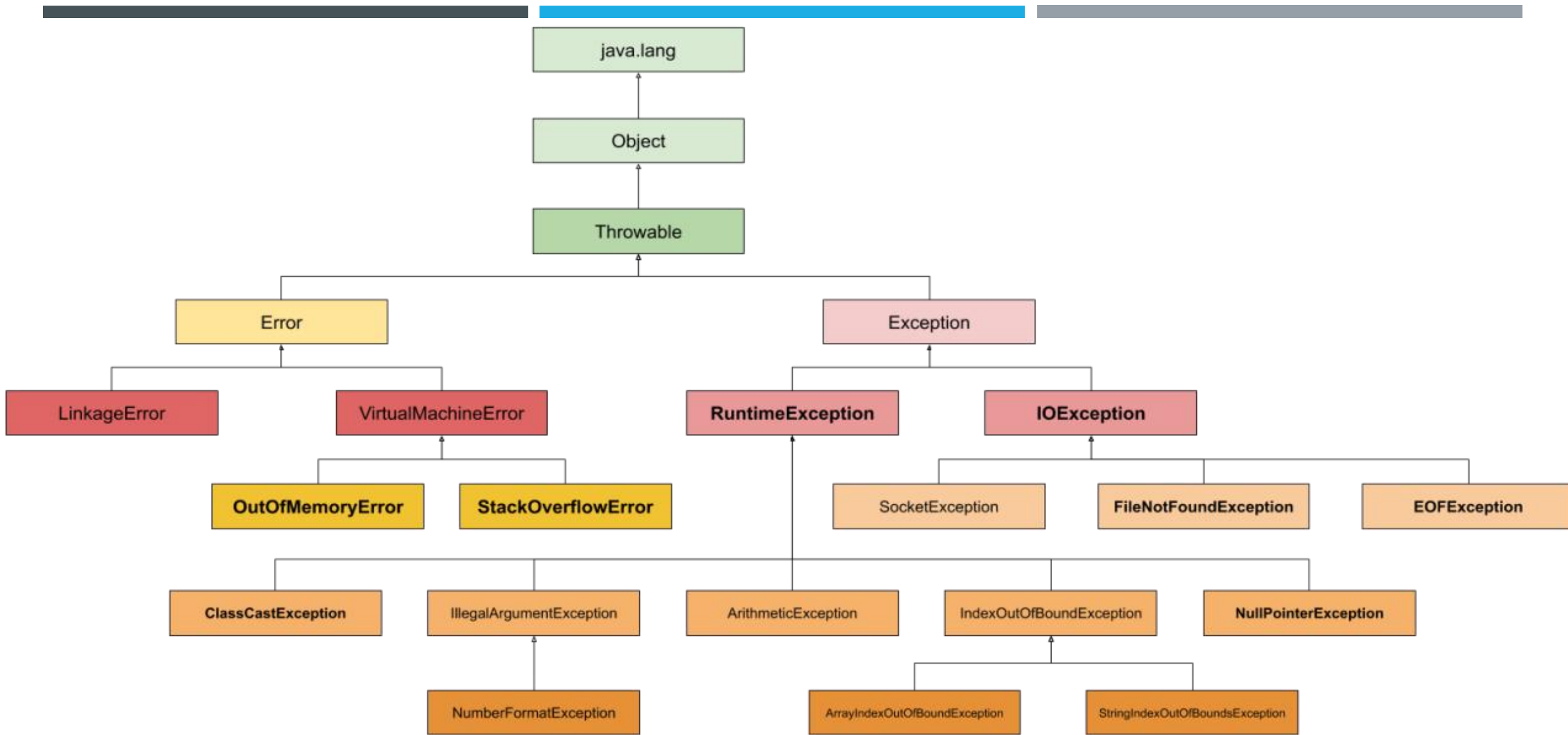
HIERARCHY OF STANDARD EXCEPTION CLASSES

- In Java, exceptions are instances of classes derived from the class **Throwable** which in turn is derived from the **Object** class.
- Whenever an exception is thrown, it implies that an object is thrown.
- Only object belonging to class that is derived from class Throwable can be thrown as exceptions.
- The next level of derived classes comprised two classes: the *Error class* and the *Exception class*.
- Error class involves errors that are mainly caused by the environment in which an application is running like *OutOfMemoryError*, *JVM Errors*, etc. These are usually the errors which are not in control of programmer.

HIERARCHY OF STANDARD EXCEPTION CLASSES

- All the classes are stored in *java.lang* package which is imported by default in any Java program.

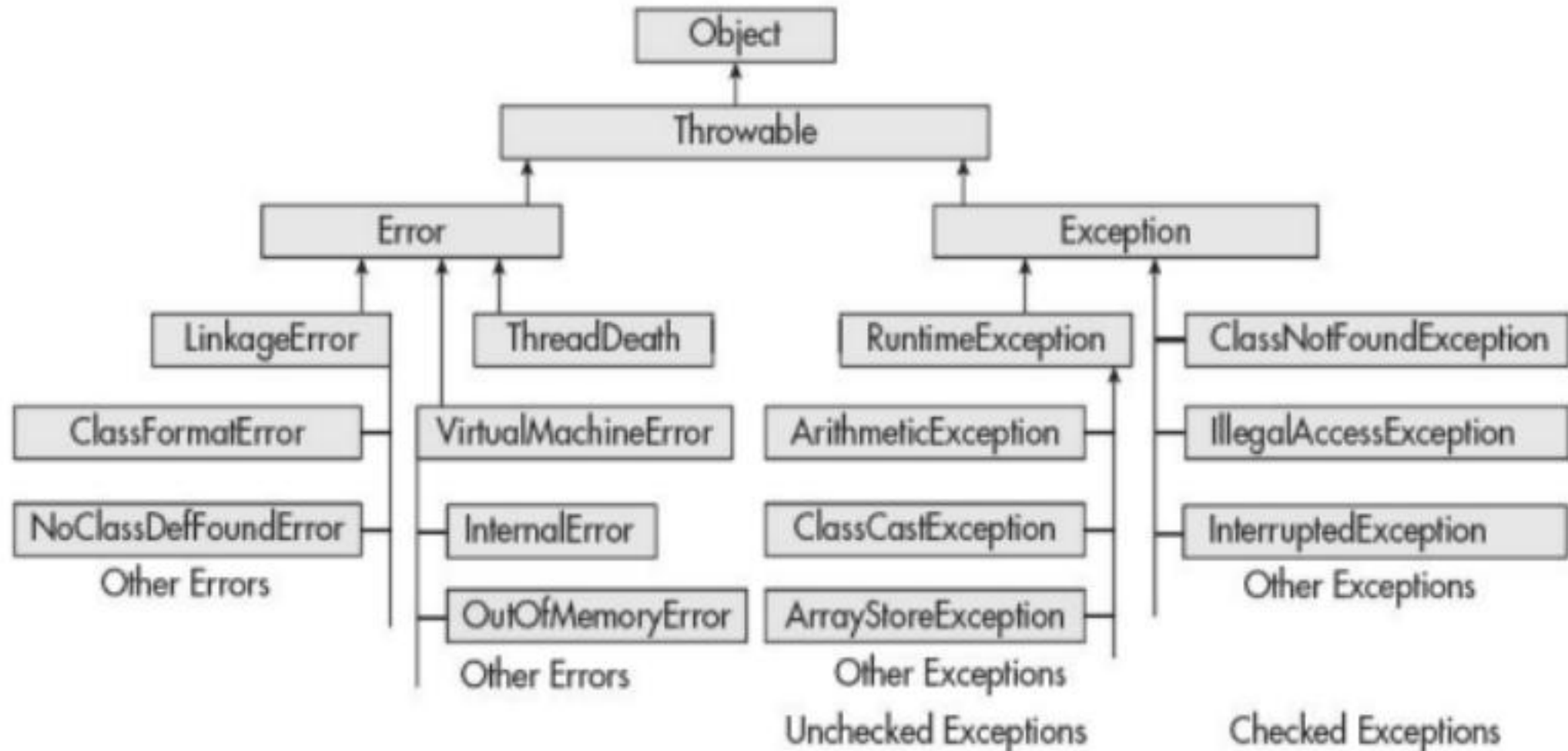




HEIRARCHY OF STANDARD EXCEPTION CLASSES

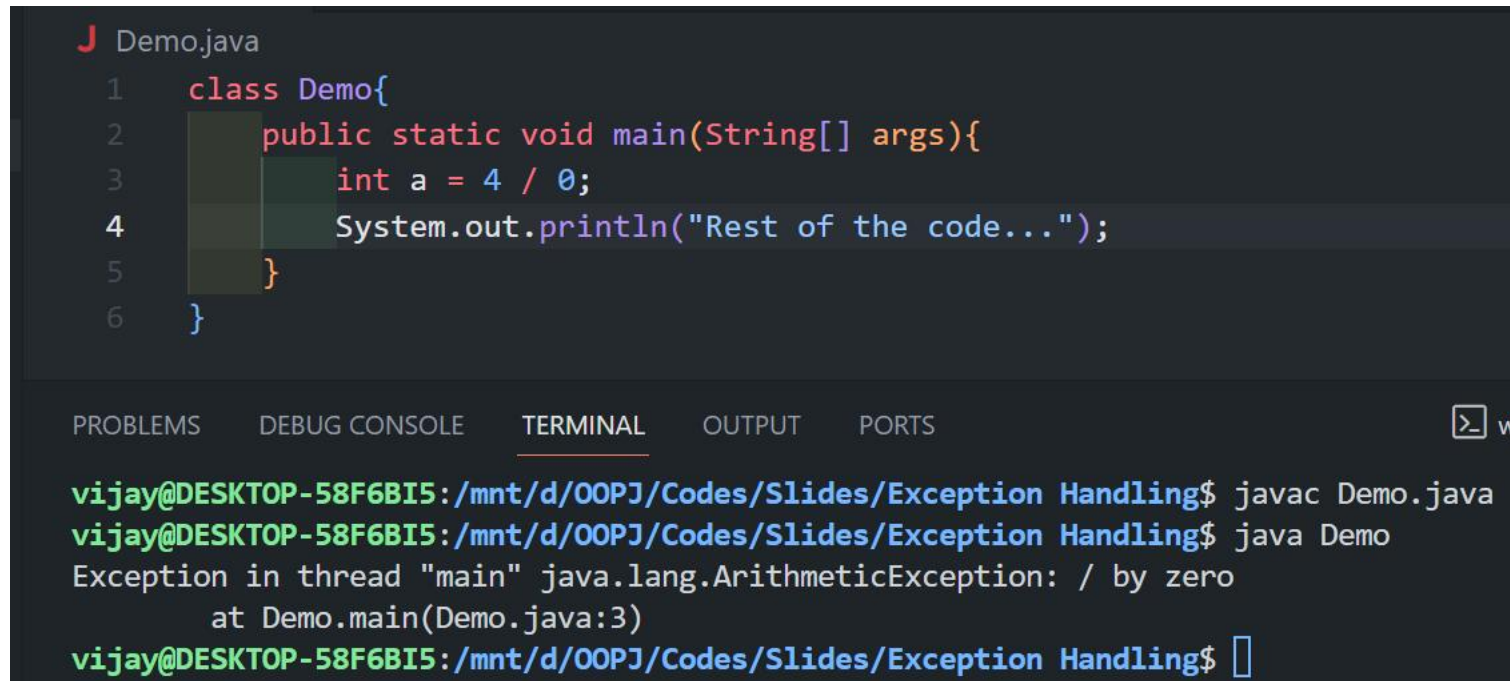
- The subclasses of Exception class are broadly subdivided into two categories -
 - **Unchecked exceptions:** These are subclasses of class *RuntimeException*, derived from Exception class. For these exceptions, the compiler does not check whether the method that throws these exceptions has provided any exception handler code or not.
 - **Checked exceptions:** These are direct subclasses of the Exception class and are not subclass of the class *RuntimeException*. For these exceptions, the compiler ensures that the methods that throw checked exceptions also deal with them i.e. provide code for handling those exceptions.
 - The methods provides the exception handler in the form of appropriate try-catch blocks.
 - The method may simply declare the list of exceptions that the method may throw and those who uses that method needs to provide code for appropriate exception handler.

CHECKED AND UNCHECKED EXCEPTIONS



UNCAUGHT EXCEPTIONS

- Let's try to understand the problem if we don't catch exceptions.



The screenshot shows an IDE with a Java file named `Demo.java`. The code is as follows:

```
1 class Demo{
2     public static void main(String[] args){
3         int a = 4 / 0;
4         System.out.println("Rest of the code...");
5     }
6 }
```

Below the code editor, the `TERMINAL` tab is active, showing the following commands and output:

```
vi@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Handling$ javac Demo.java
vi@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Handling$ java Demo
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Demo.main(Demo.java:3)
vi@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Handling$
```

- The program terminates and remaining code is not executed.

CATCHING EXCEPTIONS

J Demo.java

```
1  class Demo{
2      public static void main(String[] args){
3          try{
4              int a = 4 / 0;
5              System.out.println("a = " + a);
6          }
7          catch (ArithmeticException e){
8              System.out.println(e);
9              System.out.println("catch block");
10         }
11         System.out.println("Rest of the code...");
12     }
13 }
```

PROBLEMS

DEBUG CONSOLE

TERMINAL

OUTPUT

PORTS



vi^{jay}@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Handling\$ javac Demo.java

vi^{jay}@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Handling\$ java Demo

java.lang.ArithmeticException: / by zero

catch block

Rest of the code...

vi^{jay}@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Handling\$

TRY{ } BLOCK

- The program code that is most likely to create exceptions is kept in the try block, which is followed by the catch block to handle the exception.
- In normal execution, the statements are executed and if there are no exceptions, the program flow goes to the code line after the catch blocks.
- However if there is an exception, an exception object is thrown from the try block.
- Exception object's data members keep the information about the type of exception thrown.
- The program flow comes out of the try block and searches for an appropriate catch block with the same type as its argument.

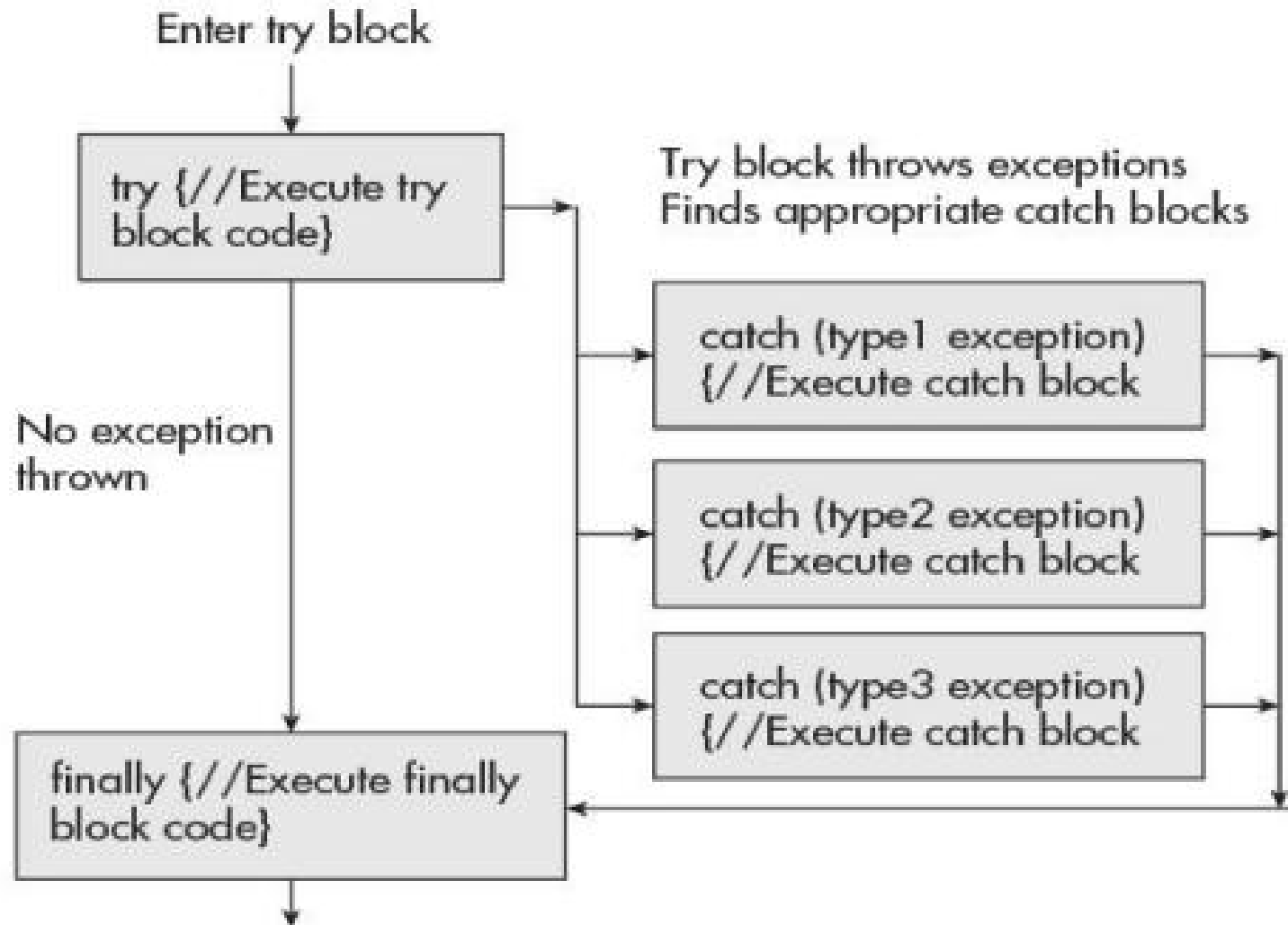
CATCH { } BLOCK

- A catch block is meant to catch the exception if the type of its argument matches with the type of exception thrown.
- If the type of exception does not match the type of the first catch block, the program flow checks the other catch blocks one by one.
- If the type of a catch block matches, its statements are executed.
- If none matches, the program flow records the type of exception, executes the finally block, and terminates the program.

FINALLY { } BLOCK

- This is the block of statements that is always executed even when there is an exceptional condition, which may or may not have been caught or dealt with.
- Thus finally block can be used as a tool for the clean up operations and for recovering the memory resources. For this, the resources should be closed in the finally block.
- This will also guard against situations when the closing operations are bypassed by statements such as continue, break or return.
- This **finally** clause is optional. However, each try statement requires at least one catch or a finally clause.

TRY, CATCH AND FINALLY BLOCK



J Demo.java

```
1  class TryCatchFinally
2  {
3      public static void main(String[] args)
4      {
5          // array of size 4
6          int[] arr = new int[4];
7          try
8          {
9              int i = arr[4];
10             System.out.println("Inside try block");
11         }
12         catch(ArrayIndexOutOfBoundsException ex)
13         {
14             System.out.println("Exception caught in the
15             catch block");
16         }
17         finally
18         {
19             System.out.println("Finally bloc executed");
20         }
21         // rest of the program
22         System.out.println("Rest of the program...");
23     }
```

vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Hand
ling\$ javac Demo.java

vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Hand
ling\$ java TryCatchFinally

Exception caught in the catch block

Finally bloc executed

Rest of the program...

vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception Hand
ling\$

WORKING OF 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 the 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.

MULTIPLE CATCH CLUASES

- Often the programs throw more than one type of exception.
- The programmer has to provide a catch block for each type of exception.
- 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.
- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error.

MULTIPLE CATCH BLOCKS

- If you try to compile this program, you will receive an error message stating that the second catch statement is unreachable because the exception has already been caught. Since `ArithmeticException` is a subclass of `Exception`, the first catch statement will handle all `Exception`-based errors, including `ArithmeticException`. This means that the second catch statement will never execute. To fix the problem, reverse the order of the catch statements.

```
/* This program contains an error.
```

```
A subclass must come before its superclass in  
a series of catch statements. If not,  
unreachable code will be created and a  
compile-time error will result.
```

```
*/  
class SuperSubCatch {  
    public static void main(String args[]) {  
        try {  
            int a = 0;  
            int b = 42 / a;  
        } catch (Exception e) {  
            System.out.println("Generic Exception catch.");  
        }  
        /* This catch is never reached because  
           ArithmeticException is a subclass of Exception. */  
        catch (ArithmeticException e) { // ERROR - unreachable  
            System.out.println("This is never reached.");  
        }  
    }  
}
```

NESTED TRY AND CATCH BLOCKS

- In nested try-catch blocks, one try-catch block can be placed within another try's body.
- Nested try block is used in cases where a part of block may cause one error and the entire block may cause another error.
- Each time a try statement is entered, the context of that exception is pushed on the stack.
- If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

NESTED TRY-CATCH BLOCKS

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

```
// An example of nested try statements.
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* If no command-line args are present,
               the following statement will generate
               a divide-by-zero exception. */
            int b = 42 / a;

            System.out.println("a = " + a);

            try { // nested try block
                /* If one command-line arg is used,
                   then a divide-by-zero exception
                   will be generated by the following code. */
                if(a==1) a = a/(a-a); // division by zero

                /* If two command-line args are used,
                   then generate an out-of-bounds exception. */
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99; // generate an out-of-bounds exception
                }
            } catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index out-of-bounds: " + e);
            }

            } catch(ArithmeticException e) {
                System.out.println("Divide by 0: " + e);
            }
        }
    }
}
```


SOME IMPORTANT POINTS

- If you don't handle exception, before terminating the program, JVM executes finally block (if any).
- For each try block there can be zero or more catch blocks, but only one finally block.
- At a time only exception is occurred and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general i.e. catch of `ArithmeticException` must come before catch for `Exception` type.

THROW KEYWORD

- So far, you have only been catching exceptions that are thrown by the Java run-time system.
- It is possible for your program to throw an exception explicitly, using the throw statement.
- The general form of **throw** is - *throw ThrowableInstance*
- We can throw either checked or unchecked exception in Java by throw keyword.
- The throw keyword is mainly used to throw custom exceptions.

THROW KEYWORD

- ThrowableInstance must be an object of class *Throwable* or a subclass of *Throwable*.
- There are two ways you can obtain a Throwable object - using a parameter in a catch clause or creating one with the new operator.
- The flow of execution stops immediately after the throw statement. And any subsequent statements are not executed.
- The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception.
- If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on.
- If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

J ThrowDemo.java

```
1  class ThrowDemo{
2      static void validate(String name){
3          try{
4              if(name.equals("")){
5                  // Create a new exception object and throw it
6                  throw new NullPointerException("Not valid");
7              }
8          }
9          catch(NullPointerException e){
10             System.out.println("Caught inside validate.");
11
12             throw e; // rethrow the exception
13         }
14     }
15
16     public static void main(String[] args){
17         try{
18             validate("");
19         }
20         catch(NullPointerException e){
21             System.out.println("Recought: " + e);
22         }
23     }
24 }
```

```
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception H
andling$ javac ThrowDemo.java
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception H
andling$ java ThrowDemo
Caught inside validate.
Recought: java.lang.NullPointerException: Not valid
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception H
andling$
```

THROWS KEYWORD

- 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. 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 otherwise it will generate a compile-time error.

THROWS KEYWORD

- General form of throws clause is:
 - *return-type method-name (parameter-list) throws exceptions-list {...}*
- Here, exception-list is a comma-separated list of the exceptions that a method can throw.

```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

THROWS KEYWORD

```
// This is now correct.
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);
        }
    }
}
```

Here is the output generated by running this example program:

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

DIFFERENCE BETWEEN THROW AND THROWS

throw	throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

BUILT-IN EXCEPTIONS

- Inside the standard package `java.lang`, Java defines several exception classes.
- `RuntimeException` exceptions need not be included in any method's throws list.
- In Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in `java.lang` are listed in Table 1.
- Table 2 lists those exceptions defined by `java.lang` 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*.
- Apart from these, Java also defines several more exceptions that are related to other standard packages in Java.

Table 1: Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
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.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
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.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 2: Java's Checked Exceptions Defined in **java.lang**

Exception	Meaning
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.
ReflectiveOperationException	Superclass of reflection-related exceptions.

USER DEFINED EXCEPTIONS

- A programmer may create his/her own exception class by extending the Exception class which in turn extends the Throwable class.
- Using Java custom exception, the programmer can write their own exceptions and messages.
- The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.

USER DEFINED EXCEPTIONS

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

```
// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Normal exit");
    }

    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        } catch (MyException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Method	Description
final void addSuppressed(Throwable <i>exc</i>)	Adds <i>exc</i> to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the try -with-resources statement.
Throwable fillInStackTrace()	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause()	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
StackTraceElement[] getStackTrace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement . The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.
final Throwable[] getSuppressed()	Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the try -with-resources statement.
Throwable initCause(Throwable <i>causeExc</i>)	Associates <i>causeExc</i> with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace()	Displays the stack trace.
void printStackTrace(PrintStream <i>stream</i>)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter <i>stream</i>)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement <i>elements</i> [])	Sets the stack trace to the elements passed in <i>elements</i> . This method is for specialized applications, not normal use.
String toString()	Returns a String object containing a description of the exception. This method is called by println() when outputting a Throwable object.

Table 10-3 The Methods Defined by **Throwable**

CHAINED EXCEPTIONS

- The chained exception feature allows you to associate another exception with an exception. The second exception describes the cause of the first exception.
- For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. To highlight I/O error, you can use chained exceptions.
- There are two constructors in the Throwable class for chained exceptions -
 - `Throwable(Throwable causeException);` Here `causeException` is the exception that causes the current exception.
 - `Throwable(String msg, Throwable causeException);` Allows you to define a message description also.

CHAINED EXCEPTIONS

- Throwable class supports two methods for chained exceptions -
 - Throwable `getCause()`; Returns the exception that underlies the current exception. If there are no underlying exceptions then **null** is returned.
 - Throwable `initCause(Throwable causeExc)`; Associates `causeExc` with the invoking exception and returns a reference to the exception. Cause exception can only be set once therefore this function can be called at max once for each exception object.

J ChainedException.java

```
1  class ChainedException{
2      static void demoproc(){
3          // Create an exception
4          NullPointerException exc = new NullPointerException
5              ("top layer");
6          // Add a cause
7          exc.initCause(new ArithmeticException("cause"));
8
9          throw exc;
10     }
11
12     public static void main(String[] args){
13         try{
14             demoproc();
15         }
16         catch(NullPointerException e){
17             // display top level exception
18             System.out.println("Caught: " + e);
19
20             // display cause exception
21             System.out.println("Cause: " + e.getCause());
22         }
23     }
24 }
```

```
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception H
andling$ javac ChainedException.java
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception H
andling$ java ChainedException
Caught: java.lang.NullPointerException: top layer
Cause: java.lang.ArithmeticException: cause
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Exception H
andling$
```

NEW FEATURE: MULTI-CATCH CLAUSE

- In JDK7, there are some features added in Java exception handling. One of which is allowing a catch block to handle multiple types of exceptions.

```
// Demonstrate the multi-catch feature.
class MultiCatch {
    public static void main(String args[]) {
        int a=10, b=0;
        int vals[] = { 1, 2, 3 };

        try {
            int result = a / b; // generate an ArithmeticException

            //      vals[10] = 19; // generate an ArrayIndexOutOfBoundsException

            // This catch clause catches both exceptions.
        } catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e);
        }

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