Introduction to Java for C++ Programmers

Segment - 2

JAC 444

Professor: Mahboob Ali

Objectives

Upon completion of this lecture, you should be able to:

- Separate Error-Handling Code from Regular Code
- Use Exceptions to Handle Exceptional Events
- Create Your Exceptions

Exceptions

In this lesson you will be learning about:

- What is and how to treat an exception in Java
- How to separate error handling from regular code
- How to write exception handler
- Exception class hierarchy
- How to create your own exception classes

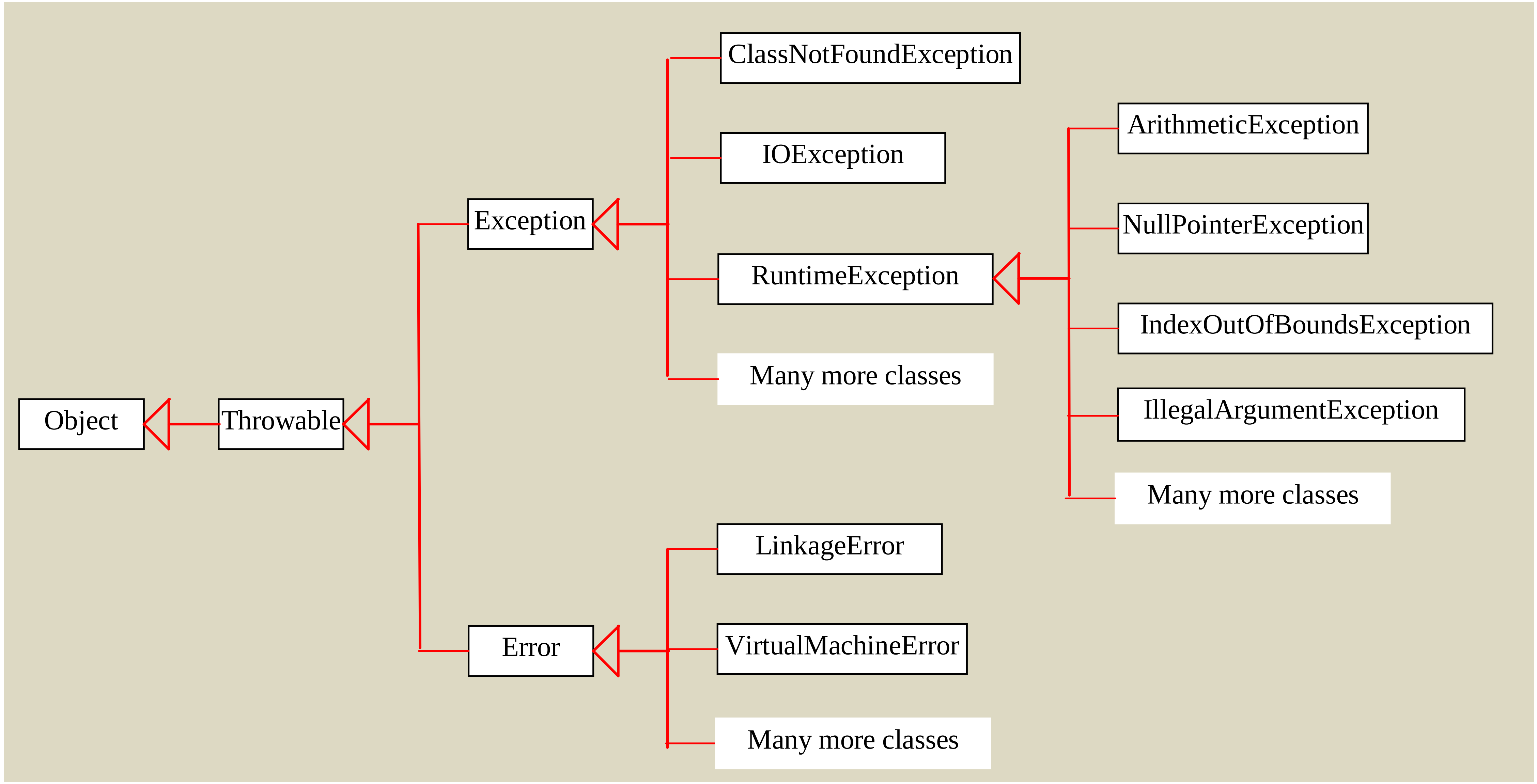
Motivation

- When a program runs into a runtime error, the program terminates abnormally. How can you handle the runtime error so that the program can continue to run or terminate gracefully?

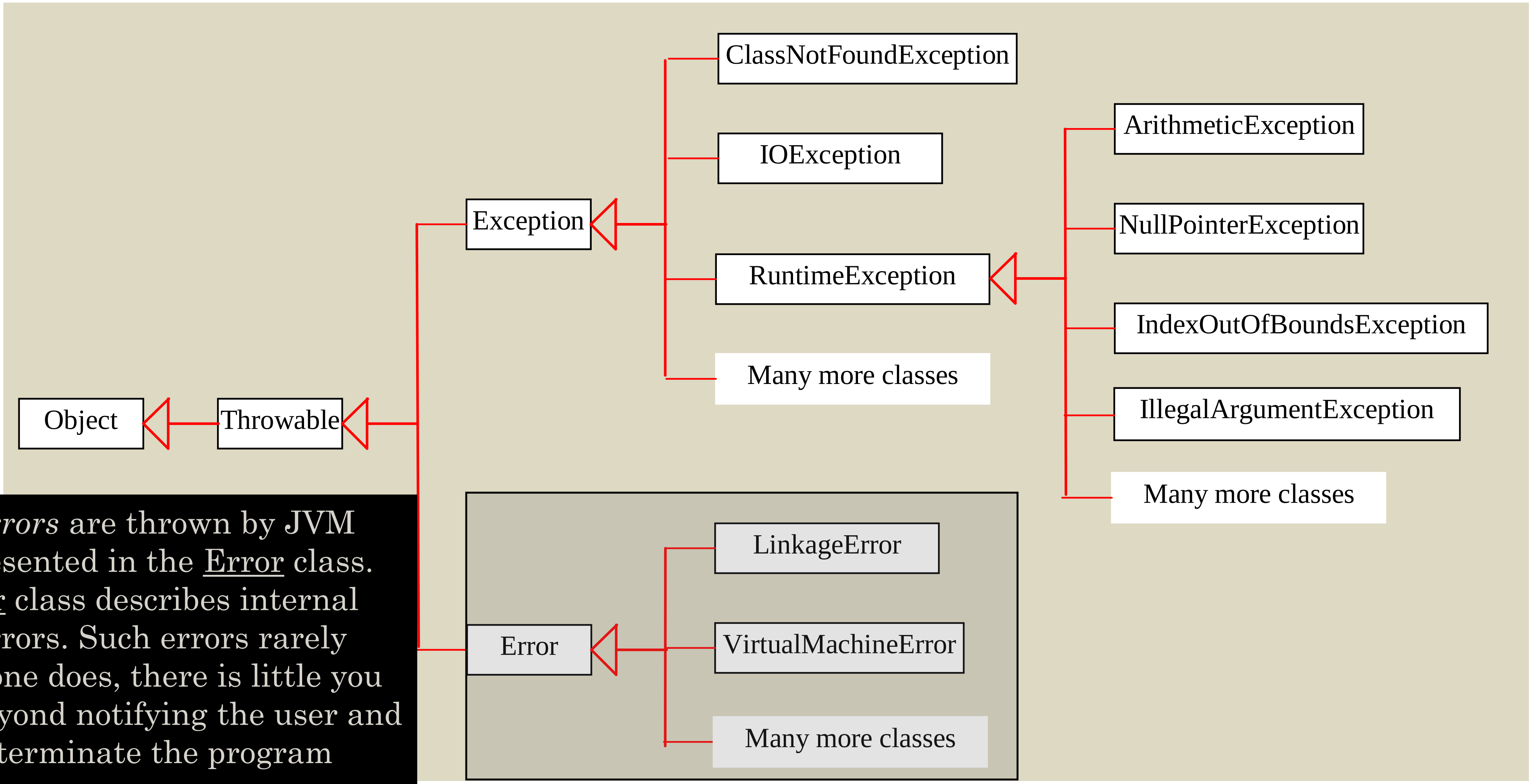
What is an exception?

- **Definition**: An exception is an event that occurs during the execution of a program that disrupts the normal flow of instruction.
- **Examples**: Serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element.
- **Java Solution**: The Java method creates an exception object and hands it off to the runtime system.

Exception Types



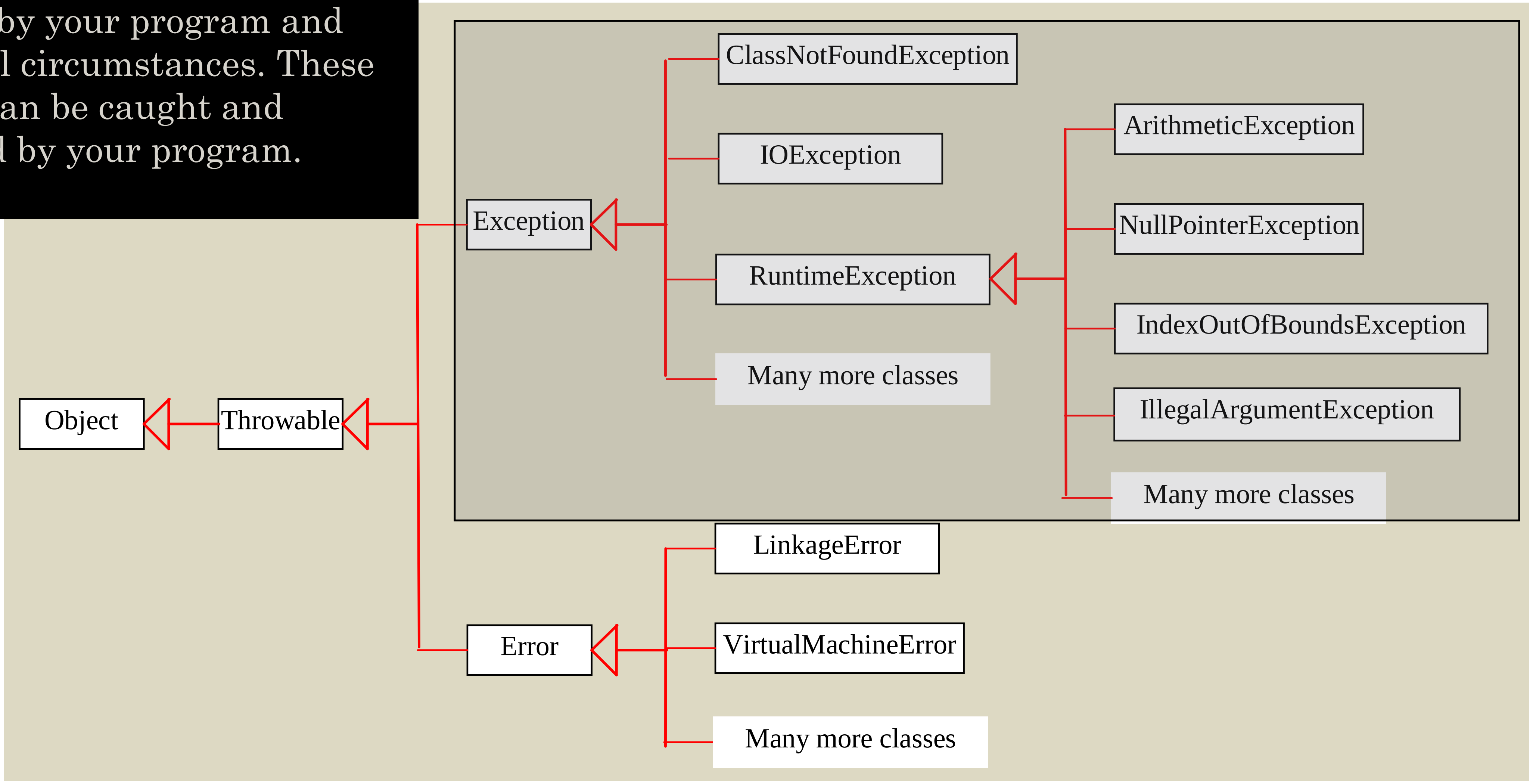
System Errors



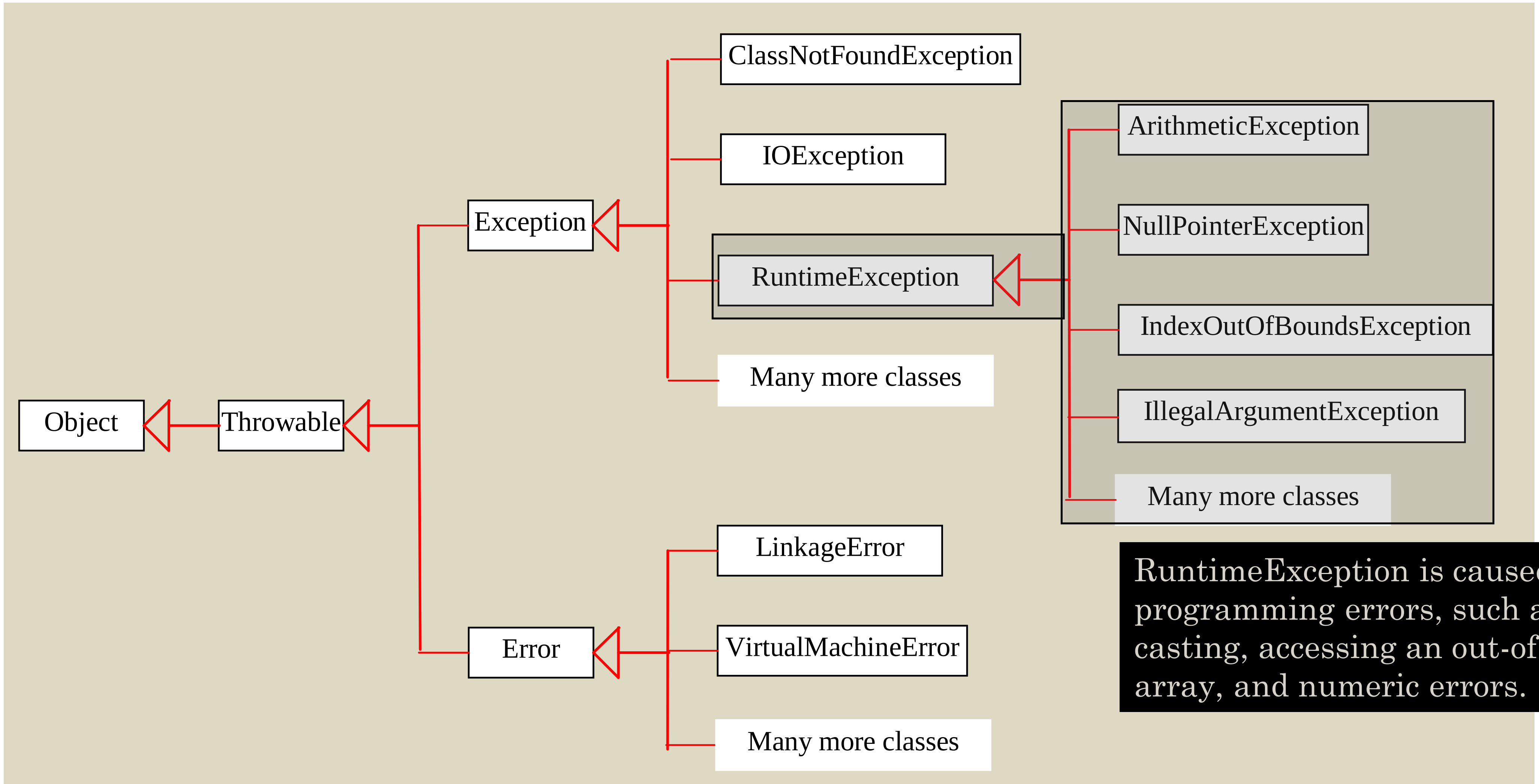
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



Runtime Exceptions



RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Advantages of Exceptions

- Separating Error Handling Code from “Regular” Code
- Propagating Errors Up the Call Stack
- Grouping Error Types and Error Differentiation

Error Handling Code

Problem: Read a file and copy its content into memory

```
… readFile ( … )  {

    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
…
}
```

Potential Errors

- What happens if the file can not be opened?
- What if the length of the file can not be determined?
- What happens if enough memory can not be allocated?
- What happens if the read fails?
- What happens if the file can not be closed?

Error Detection Code Solution

```
int readFile ( … ) {  initialize
  errorCode = 0;
  //open the file;
  if (theFileIsOpen) {
     //determine the length of the file;  if
     (gotTheFileLength) {
        //allocate that much memory;  if
        (gotEnoughMemory) {
           //read the file into memory;  if
           (readFailed) {
              errorCode = -1;
                }
        } else {  errorCode = -2;
        }
     } else {  errorCode = -3;
     }
    …
```

Java Solution: Exception Handler

```
void readFile() {
    try {
            open the file;  determine its
            size;
            allocate  that  much  memory;   read
            the  file  into  memory;   close  the
            file;
    } catch (fileOpenFailed) {
            doSomething;
    } catch (sizeDeterminationFailed)  {
            doSomething;
    } catch (memoryAllocationFailed) {
            doSomething;
    } catch (readFailed) {
            doSomething;
    } catch (fileCloseFailed) {
            doSomething;  …
```

Types of Exceptions

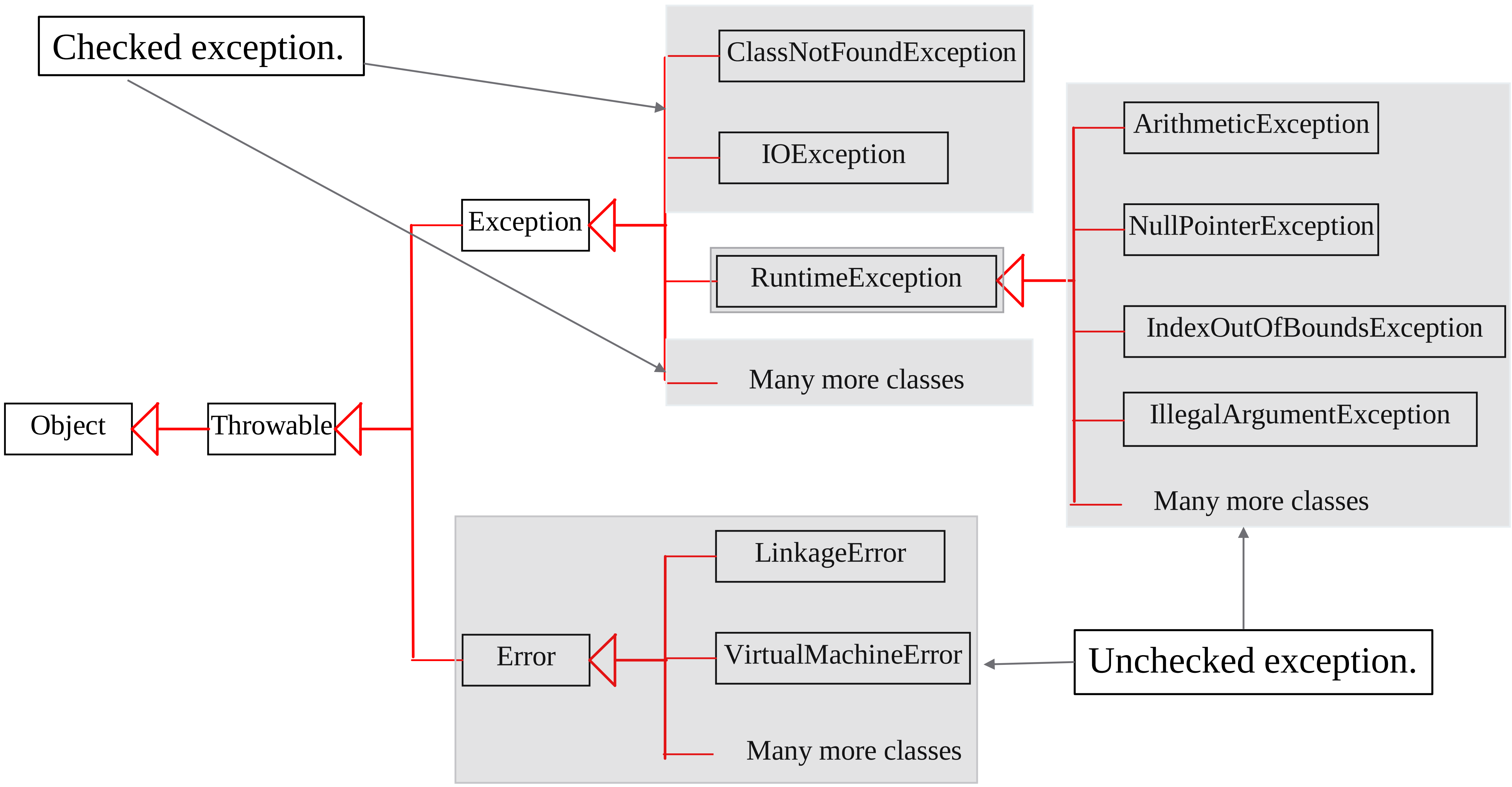
- There are basically two types of Exceptions

  1. Checked Exceptions ← compiler forces the programmer to check and deal with the exceptions
  2. Unchecked Exceptions ← reflect programming logic errors that are not recoverable

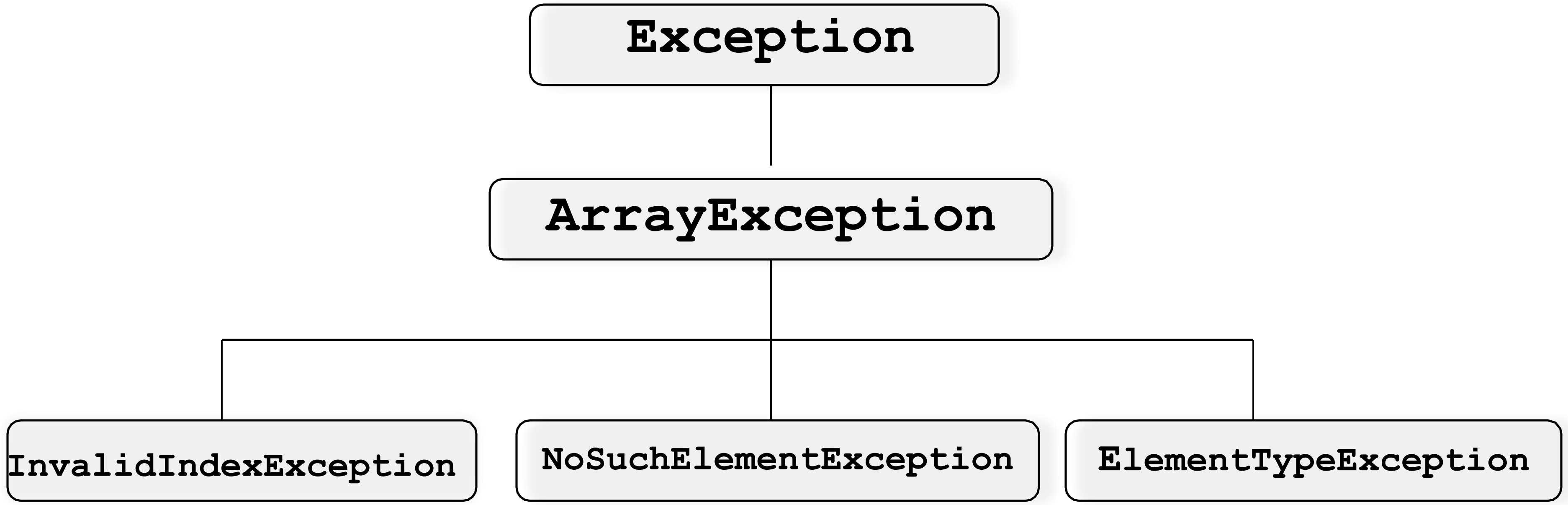
- But according to Sun microsystems there is a third type exception as well.

  3. Error

Checked and Unchecked Exceptions



ArrayException Example



How to Write an Exception Handler

There are five keywords in Java to be used for Exception Handling.

1. try
2. catch
3. finally
4. throw
5. throws

How to Write an Exception Handler

1. Write the **try**block
   It is a block that encloses the statements that might throw an exception

2. Write the **catch**block(s)
   It is used to handle the Exception associated with **try** block.
   Multiple catch blocks can be written for a single **try** block.

3. Write the **finally**block
   finallyblock provides a mechanism that allows your method to clean up after itself

Example Try and Catch

Not handled

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

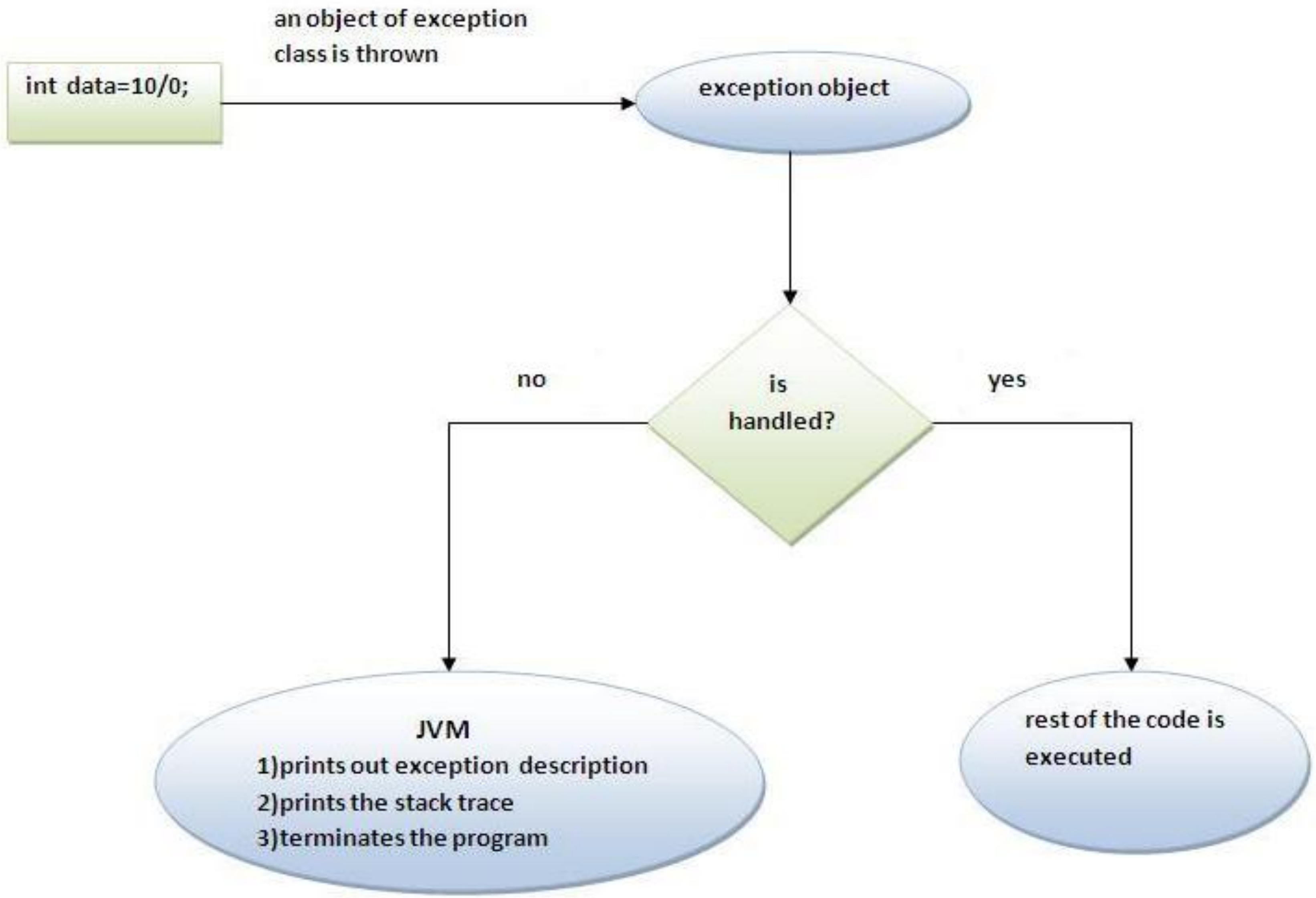
```
Exception in thread main
java.lang.ArithmeticException:/ by zero
```

Handled with Try and Catch

```
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...");
     }
}
```

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

Internal working of Java try-catch


an object of exception class is thrown
int data=10/0;
exception object
is handled?
no
yes
JVM
1)prints out exception description
2)prints the stack trace
3)terminates the program
rest of the code is executed

Example catch Block(s)

```
public class TestMultipleCatchBlock{
    public static void main(String args[]){
    try{
    int a[]=new int[5];
    a[5]=30/0;
    }
  catch(ArithmeticException e){System.out.println("task1 is completed");}
  catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
  catch(Exception e){System.out.println("common task completed");}

  System.out.println("rest of the code...");
    }
}
```

General Exception has to be the last one to come to avoid compiler errors

Output:task1 completed
rest of the code...

Rules for Multiple Catch

Rule 1: At a time only one Exception is occurred and at a time only one catch block is executed.

Rule 2: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception.

Q: How can we handle both Exceptions in the last Example at the same time?

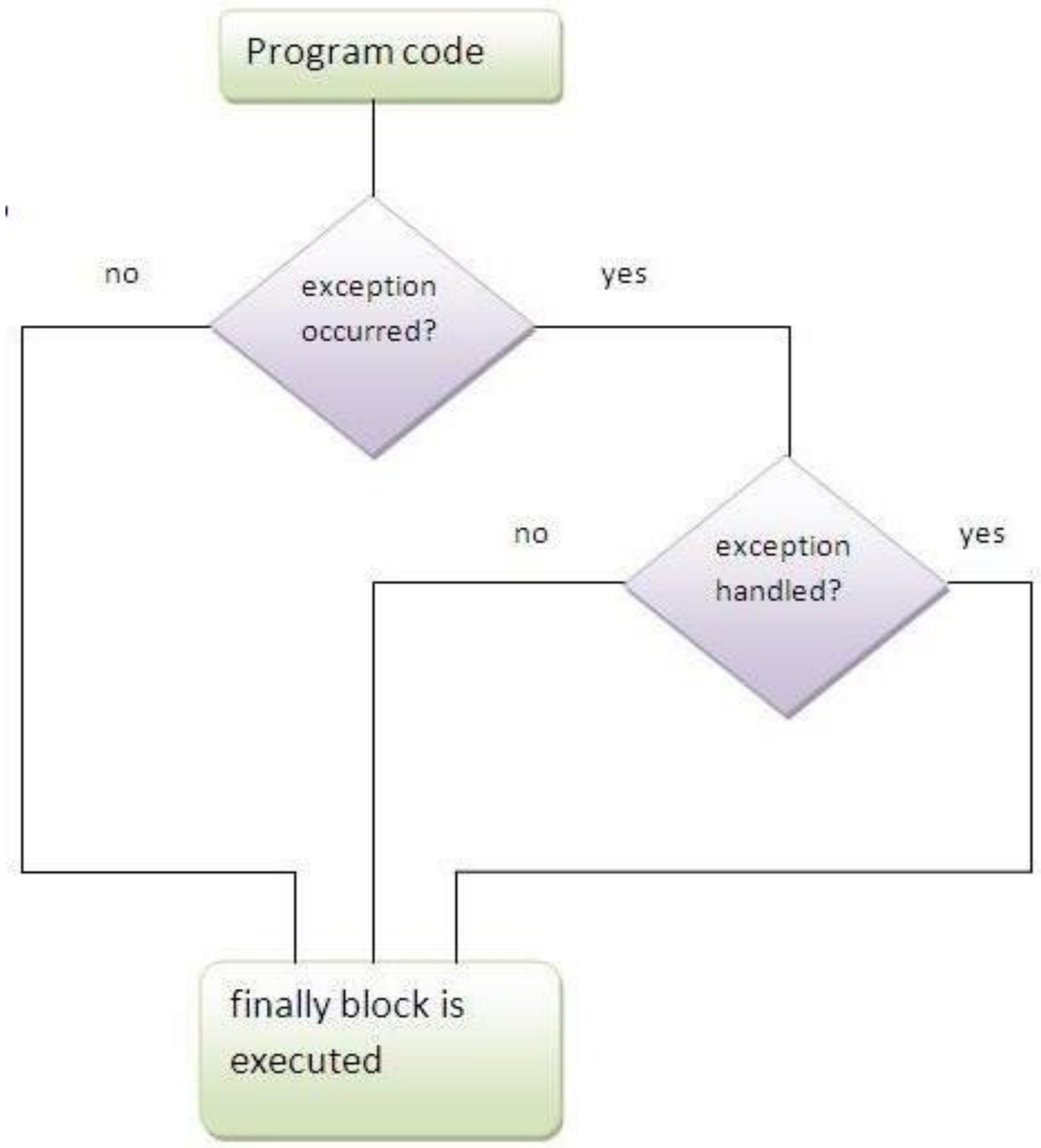
Java Nested Try Blocks

```
Class MultiTry{
    public static void main(String args[]){
        try{
            try{
                System.out.println("going to divide");
                int b =39/0;
            }catch(ArithmeticException e){System.out.println(e);}

            try{
                int a[]=new int[5];
                a[5]=4;
        }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

        System.out.println("other statement”);
         }catch(Exception e){System.out.println("handeled");}

         System.out.println("normal flow..");
        }
}
```

```
going to divide
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsExcep
tion: 5
other statement
normal flow..
```

The `finally` Block

- **finally** block is used to execute code like, closing connection, closing file etc.
- Always executed whether exception is handled or not.
- Followed by `Try` and `Catch` block.
- For each `Try` there can be zero or more `Catch` blocks, but always one `Finally` block.



Exception Doesn’t occur

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Output:

```
5
finally block is always executed
rest of the code...
```

Exception occurs and Not Handled

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e){System.out.println(e);}
  finally{System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");
  }
}
```

Unhandled Exception
Why? $\longrightarrow$

Output:

```
finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero
```

Exception occurs and Handled

```
public class TestFinallyBlock2{
 public static void main(String args[]){
 try{
  int data=25/0;
  System.out.println(data);
 }
 catch(ArithmeticException e){System.out.println(e);}
 finally{System.out.println("finally block is always executed");}
 System.out.println("rest of the code...");
 }
}
```

Output:

```
java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...
```

Is there can be a situation in which finally blocked is not executed?

Yes, there are couple of situations,

- If exception is thrown in finally.
- If JVM crashes in between, for example, System.exit();

When to Use Exceptions

When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

When to Use Exceptions

is better to be replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

throw keyword

- Used to explicitly throw an Exception.
- Checked or unchecked exception can be thrown using the keyword.
- The **throw** statement is used to create an exception object.

- throw new exception;

```
public class TestThrow1{
   static void validate(int age){
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

Output:

```
java.lang.ArithmeticException:not valid
```

Specifying Exceptions

One can specify exceptions in the method definition with the keyword:

Throws exception

The **throws** clause is composed of the throws keyword followed by a comma-separated list of all the exceptions thrown by method.

Example:

```
public void writeList(...)throws IOException,
                              ArrayIndexOutOfBoundsException{
        ...
}
```

The `Throwable` Class

throw	**throws**
Used to explicitly throw and exception.	Used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
Followed by an instance.	Followed by a class.
Used within the method.	Used with the method signature.
Cannot throw multiple exceptions.	can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

`final`	`finally`	`finalize`
Is a **keyword**	Is a **block**	Is a **method**
• Final is used to apply restrictions on class, method and variable. • Final class can't be inherited. • Final method can't be overridden. • Final variable value can't be changed.	• Finally is used to place important code, it will be executed whether exception is handled or not.	• Finalize is used to perform clean up processing just before object is garbage collected.

animation

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose no exceptions in the statements

animation

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed

animation

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}


Next statement;
```

Next statement in the method is executed

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The exception is handled.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller