VIPS
Technical Campus
योग: कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

SCHOOL OF ENGINEERING AND TECHNOLOGY

Course : OBJECT ORIENTED PROGRAMMING

Paper Code:AIDS 202, IIOT202,AIML202

Faculty : Dr. Shivanka

Assistant Professor

VIPS

What are Java Exception?

- Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Java Exceptions

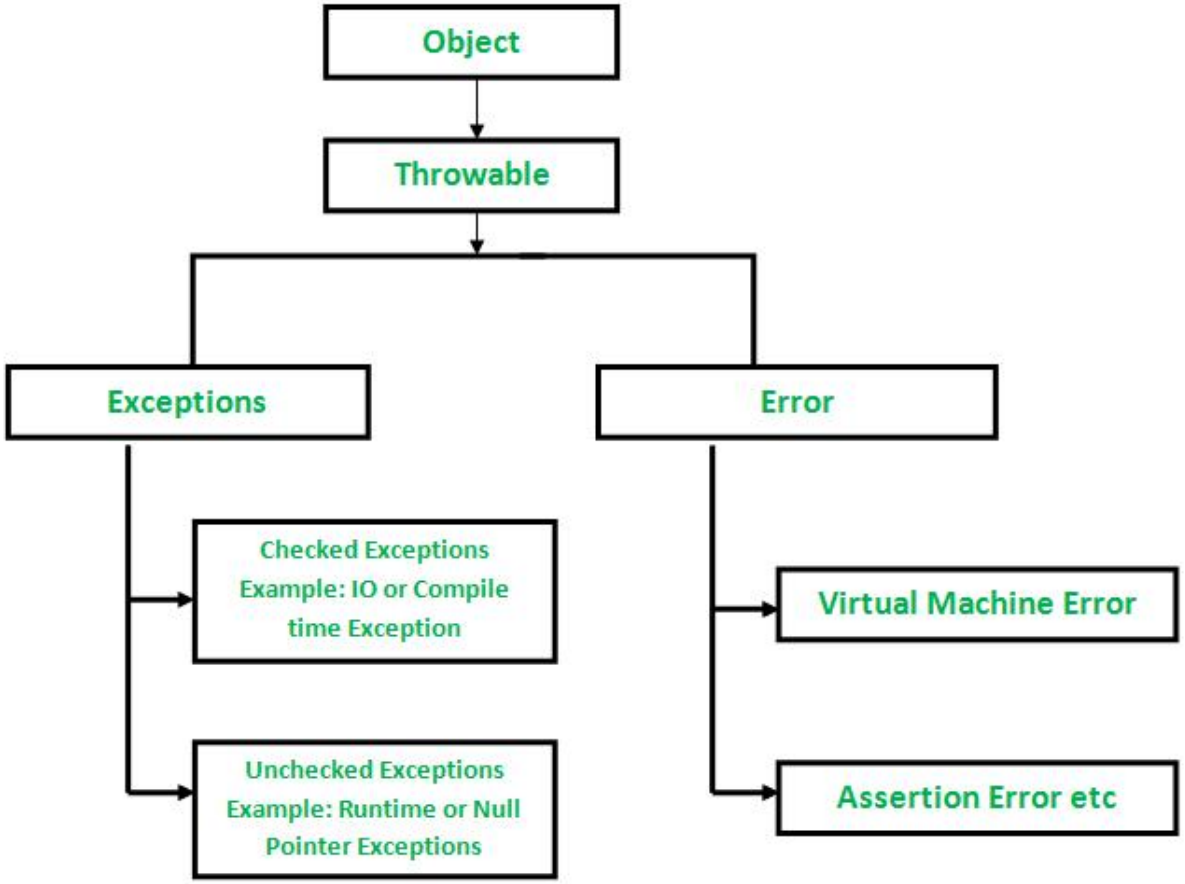
What is an Exception?

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an exception (throw an error).
- When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

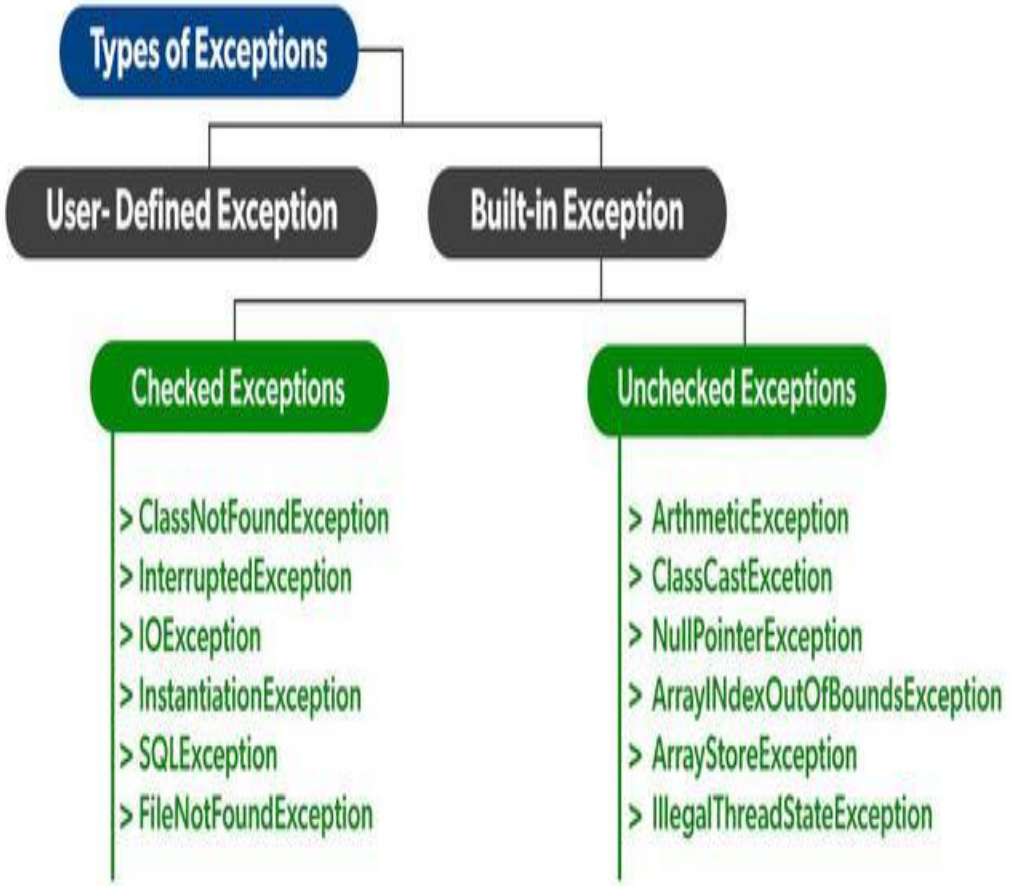
Exception Hierarchy

- All exception and error types are subclasses of class Throwable, which is the base class of the hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception

Exception Hierarchy



Types of Exceptions



- Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

Five Key words of Exceptional handling

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

Java try and catch

The **try statement** allows you to define a block of code to be tested for errors while it is being executed.

The **catch statement** allows you to define a block of code to be executed, if an error occurs in the try block.

The **try and catch keywords** come in pairs:

Syntax

```
try {
  // Block of code to try
}
catch(Exception e) {
  // Block of code to handle errors
}
```

Built-in Exceptions:

- Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.
- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

Syntax Errors, Runtime Errors, and Logic Errors

You learned that there are three categories of errors: **Syntax errors, Runtime errors, and Logic errors.**

Syntax errors arise because the rules of the language have not been followed. They are detected by the compiler.

Runtime errors occur while the program is running if the environment detects an operation that is impossible to carry out.

Logic errors occur when a program doesn't perform the way it was intended to.

Error handling through Exceptions

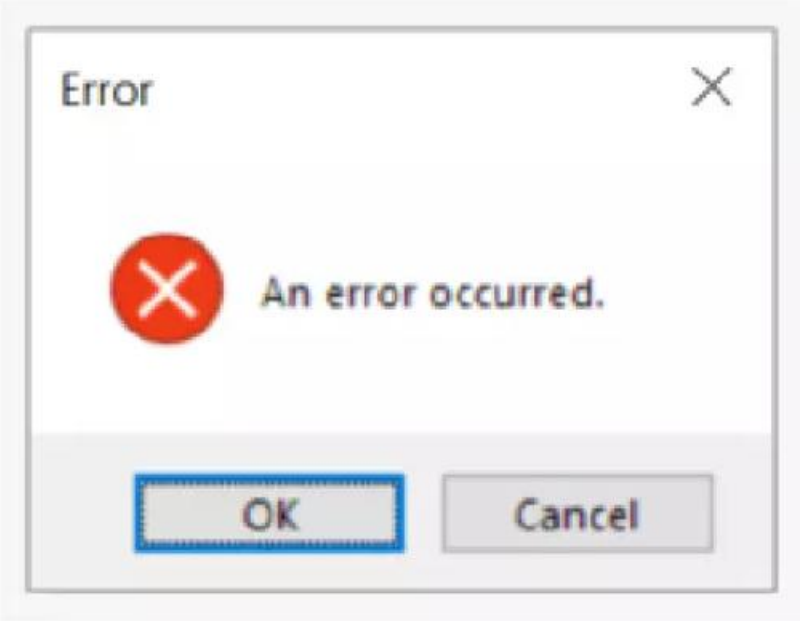
- Gradually, programmers began to realize that the traditional method of handling errors was too cumbersome for most error handling situations.

- This gave rise to the Exception concept
  - ✓ When an error occurs, that represents an **Exceptional condition.**
  - ✓ **Exceptions** cause the current program flow to be interrupted and transferred to a registered exception handling block.
  - ✓ This might involve unrolling the method calling stack.

- **Exception handling** involves a well-structured goto.

Definition of Error

Error is an unexpected event that cannot be handled at runtime. Errors can terminate your program. Most of the time, programs cannot recover from an error. Errors cannot be caught or handled. They are generally caused by the environment in which the code is running. e.g, The image below is seen by almost all the computer users.



Definition of Exception

An Exception is the occurrence of an event that can disrupt the normal flow of the program instructions. Exceptions can be caught and handled in order to keep the program working for the exceptional situation as well, instead of halting the program flow.If the exception is not handled, then it can result in the termination of the program. Exceptions can be used to indicate that an error has occurred in the program.

When an exception occurs, it creates an exception object. It holds information about the name and description of the exception and stores the program's state when the exception occurred.

The below image shows the flow of an exception. definition of exception

Exception -Terminology

- When an error is detected, an exception is *thrown*.
- Any exception which is thrown, must be caught by and exception handler
  - ✓ If the programmer hasn't provided one, the exception will be caught by a catch-all exception handler provided by the system.
  - ✓ The default exception handler may terminate the application.
- Exceptions can be rethrown if the exception cannot be handled by the block which caught the exception
- Java has 5 keywords for exception handling:
  - try
  - catch
  - finally
  - throw
  - throws

Exceptional flow of control

- Exceptions break the normal flow of control.
- When an exception occurs, the statement that would normally execute next is not executed.
- What happens instead depends on:
  - ✓whether the exception is caught,
  - ✓where it is caught,
  - ✓what statements are executed in the ‘catch block’,
  - ✓and whether you have a ‘finally block’.

Major reasons why an exception Occurs

- ❖Invalid user input
- ❖Device failure
- ❖Loss of network connection
- ❖Physical limitations (out of disk memory)
- ❖Code errors
- ❖Opening an unavailable file

- Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

Example: Array out of bound Exception

Consider the following example:

This will generate an error, because myNumbers[10] does not exist.

```
public class Main {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
    System.out.println(myNumbers[10]); // error!
  }}
```

The output will be something like this:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
        at Main.main(Main.java:4)
```

If an error occurs, we can use try...catch to catch the error and execute some code to handle it:

Example to use try...catch

Example

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

- The output will be:

- Something went wrong.

Finally Keyword

The finally statement lets you execute code, after try...catch, regardless of the result:

Example

```
public class Main {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    } }}
```

- The output will be:

- Something went wrong.
- The 'try catch' is finished.

Runtime Errors


```
1
2
3
4
5
6
7
8
9
10
11
12
13
```

```
import java.util.Scanner;

public class ExceptionDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter an integer: ");
    int number = scanner.nextInt();

    // Display the result
    System.out.println(
      "The number entered is " + number);
  }
}
```

If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

```
Terminated
```

Catch Runtime Errors


```
1  import java.util.*;
2
3  public class HandleExceptionDemo {
4    public static void main(String[] args) {
5      Scanner scanner = new Scanner(System.in);
6      boolean continueInput = true;
7
8      do {
9        try {
10         System.out.print("Enter an integer: ");
11         int number = scanner.nextInt();
12
13         // Display the result
14         System.out.println(
15           "The number entered is " + number);
16
17         continueInput = false;
18       }
19       catch (InputMismatchException ex) {
20         System.out.println("Try again. (" +
21           "Incorrect input: an integer is required)");
22         scanner.nextLine(); // discard input
23       }
24     } while (continueInput);
25   }
```

If an exception occurs on this line, the rest of lines in the try block are skipped and the control is transferred to the catch block.

Example to use try...catch

To illustrate how easily this can be done, the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error:

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

Output:
This program generates the following output:

```
Division by zero.

After catch statement
```

Approaches to handling an exception

1. Prevent the exception from happening.
2. Catch it in the method in which it occurs, and either
   a. Fix up the problem and resume normal execution.
   b. Rethrow it.
   c. Throw a different exception.
3. Declare that the method throws the exception.
4. With 1. and 2.a. the caller never knows there was an error.
5. With 2.b., 2.c., and 3., if the caller does not handle the exception, the program will terminate and display a stack trace.

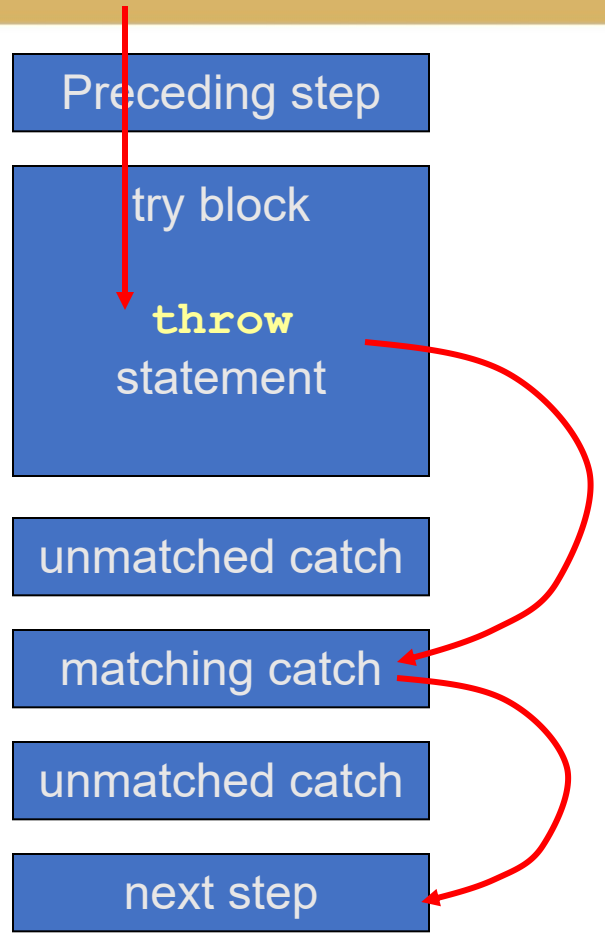
Exceptions -Syntax


```
try
{
   // Code which might throw an exception
   // ...
}
catch(FileNotFoundException x)
{
   // code to handle a FileNotFound exception
}
catch(IOException x)
{
   // code to handle any other I/O exceptions
}
catch(Exception x)
{
   // Code to catch any other type of exception
}
finally
{
   // This code is ALWAYS executed whether an exception was thrown
   // or not.  A good place to put clean-up code. ie. close
   // any open files, etc...
}
```

Execution of try catch blocks

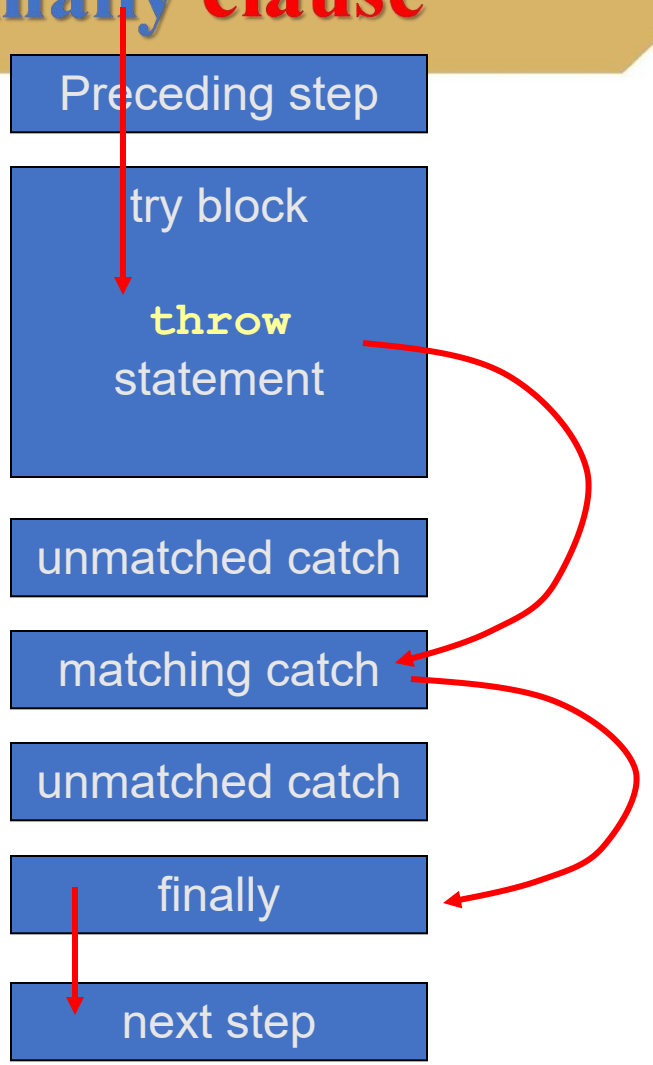
- For normal execution:
  - try block executes, then finally block executes, then other statements execute
- When an error is caught and the catch block throws an exception or returns:
  - try block is interrupted
  - catch block executes (until throw or return statement)
  - finally block executes
- When error is caught and catch block doesn't throw an exception or return:
  - try block is interrupted
  - catch block executes
  - finally block executes
  - other statements execute
- When an error occurs that is not caught:
  - try block is interrupted
  - finally block executes

Sequence of Events for throw

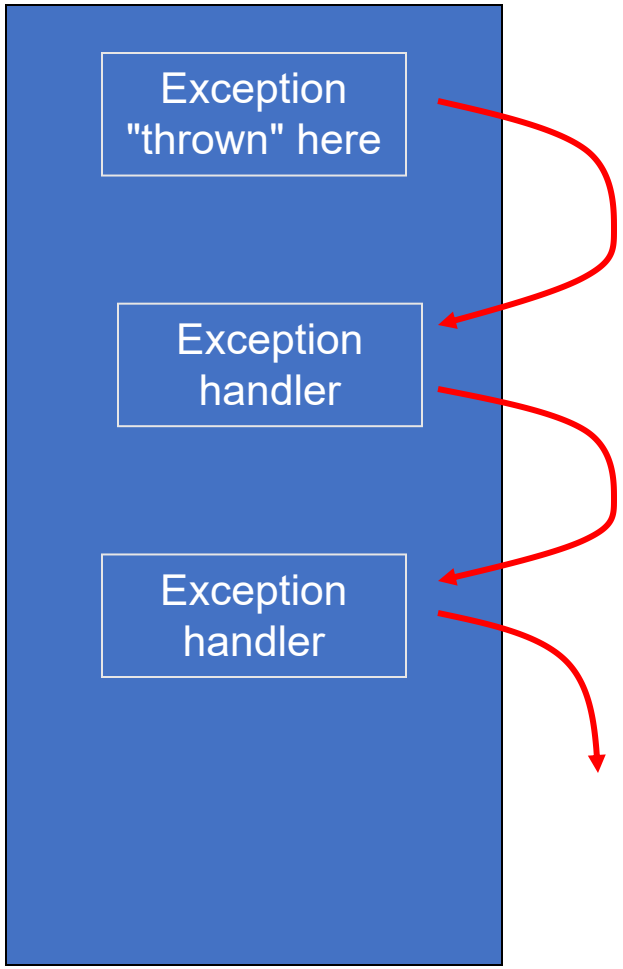


Sequence of Events for No throw

Sequence of Events for finally clause



Exception Handler



Thrown exception matched against first set of exception handlers

If it fails to match, it is matched against next set of handlers, etc.

If exception matches none of handlers, program is abandoned

Trace a Program Execution

Suppose no exceptions in the statements

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

Next statement;
```

Trace a Program Execution


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

Next statement;
```

The final block is always executed

Trace a Program Execution


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

Next statement;
```

Next statement in the method is executed

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

Trace a Program Execution


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

Next statement;
```

The exception is handled.

Trace a Program Execution


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

Next statement;
```

The final block is always executed.

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.

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.

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

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

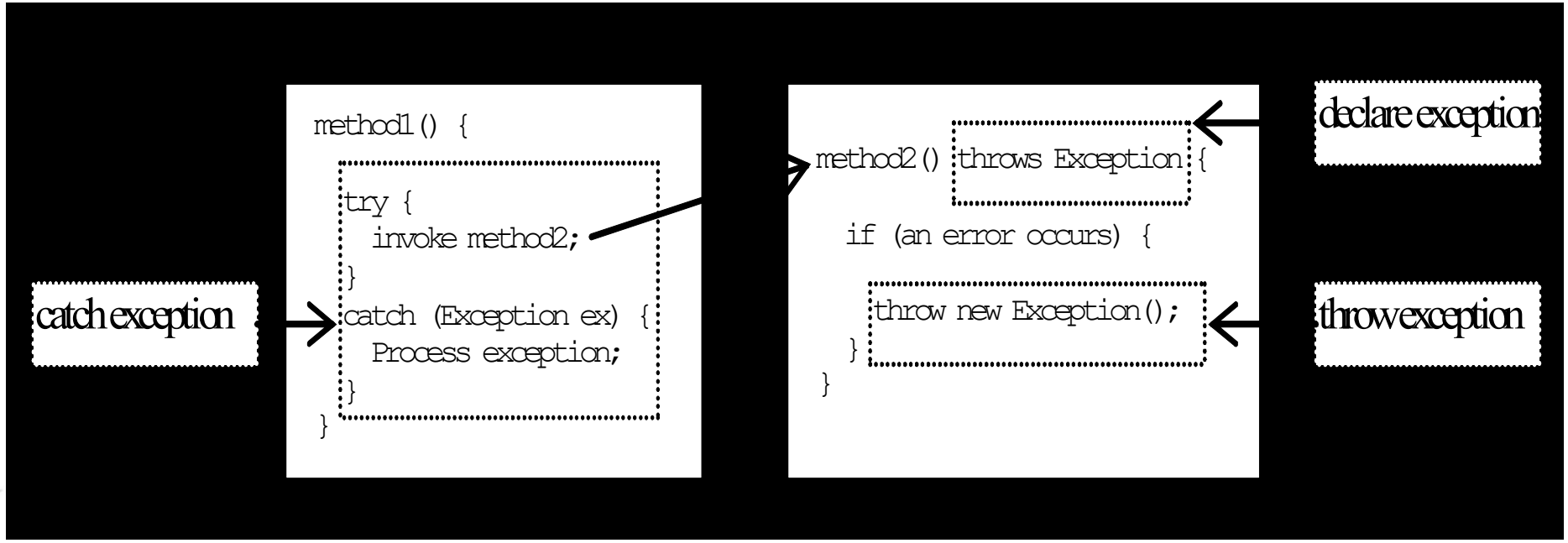
Trace a Program Execution


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

Next statement;
```

Execute the final block

Declaring, Throwing, and Catching Exceptions



Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()   throws IOException

public void myMethod() throws IOException,

        OtherException
```

Automatically Passing Exceptions

If your method is defined to throw an exception, you need not catch it within your method

```
public void addURL(String urlText) throws MalformedURLException
{
  URL  aURL = new URL(urlText);

  // if the above throws a MalformedURLException, we need not have
  // a try/catch block here because the method is defined to
  // throw that exception.  Any method calling this method MUST
  // have a try/catch block which catches MalformedURLExceptions.
  // Control is automatically transferred to that catch block.

}
```

Using throws clause

If you don't want the exception to be handled in the same function you can use the throws class to handle the exception in the calling function.

```
public class myexception{
    public static void main(String args[])
  {
            try{
                        checkEx();
            } catch(FileNotFoundException ex){          }
     }
    public void checkEx() throws FileNotFoundException
  {
            File f = new File("myfile");
            FileInputStream fis = new FileInputStream(f);
            //continue processing here.
     }
}
```

In this example, the main method calls the checkEx() method and the checkEx method tries to open a file, If the file in not available, then an exception is raised and passed to the main method, where it is handled.

Exceptions -throwing multiple exceptions

- A Method can throw multiple exceptions.
- Multiple exceptions are separated by commas after the throws

```
public class MyClass
{
  public int computeFileSize() throws IOException, ArithmeticException
  [...]
```

```
    public void method1()
    {
      MyClass anObject = new MyClass();
      try
      {
            int theSize = anObject.computeFileSize();
      }
      catch(ArithmeticException x)
      {
            // ...
      }
      catch(IOException x)
      {
            // ...
```

Exception -The catch-all Handler

- ✓ Since all Exception classes are a subclass of the **Exception** class, a catch handler which catches "Exception" will catch all exceptions.
- ✓ It must be the last in the catch List

```
public void method1()
{
  FileInputStream aFile;
  try
  {
       aFile = new FileInputStream(...);
       int aChar = aFile.read();
       //...
  }
  catch(IOException x)
  {
       // ...
  }
  catch(Exception x)
  {
       // Catch All Exceptions
```

The finally block


```
public void method1()
{
   FileInputStream aFile;
   try
   {
         aFile = new FileInputStream(...);
         int aChar = aFile.read();
         //...
   }
   catch(IOException x)
   {
         // ...
   }
   catch(Exception x)
   {
         // Catch All other Exceptions
   }
   finally
   {
         try
         {
                  aFile.close();
         }
         catch (IOException x)
         {
                  // close might throw an exception
         }
   }
```

Java throw keyword

- The flow of execution stops immediately after the throw statement; 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 programand prints the stack trace.
- The Java throw keyword is used to throw an exception **explicitly**.
- We specify the exception object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

Demonstrate throw Example

This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ). The demoproc( ) method then sets up another exception_x0002_handling context and immediately throws a new instance of NullPointerException, which is is caught on the next line. The exception is then rethrown.

```
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);
} } }
```

Resulting output:
Caught inside demoproc.
Recaught:
java.lang.NullPointerException: demo

Throw incorrect example


```
// 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();
}
}
```

To make this example compile, you need to make two changes.

First, you need to declare

that throwOne( ) throws IllegalAccessException.

Second, main( ) must define a try/catch

statement that catches this exception.

Throws corrected example

The corrected example is shown here:

```
// 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);
}}}
```

Output:

Here is the output generated by running this example program:

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

Throwing Exceptions

- You can **throw** exceptions from your own methods.
- To throw an exception, create an instance of the exception class and "throw" it.
- If you throw checked exceptions, you must indicate which exceptions your method throws by using the throws keyword

```
public void withdraw(float anAmount) throws InsufficientFundsException
{
  if (anAmount<0.0)
       throw new IllegalArgumentException("Cannot withdraw negative amt");

  if (anAmount>balance)
       throw new InsuffientFundsException("Not enough cash");

  balance = balance - anAmount;
}
```

Re-throwing Exceptions

- If you catch an exception but the code determines it cannot reasonably handle the exception, it can be rethrown:

```
public void addURL(String urlText) throws MalformedURLException
{
  try
  {
       URL  aURL = new URL(urlText);
       // ...
  }
  catch(MalformedURLException x)
  {
       // determine that the exception cannot be handled here
       throw x;
  }
}
```

Throwing Exceptions Example

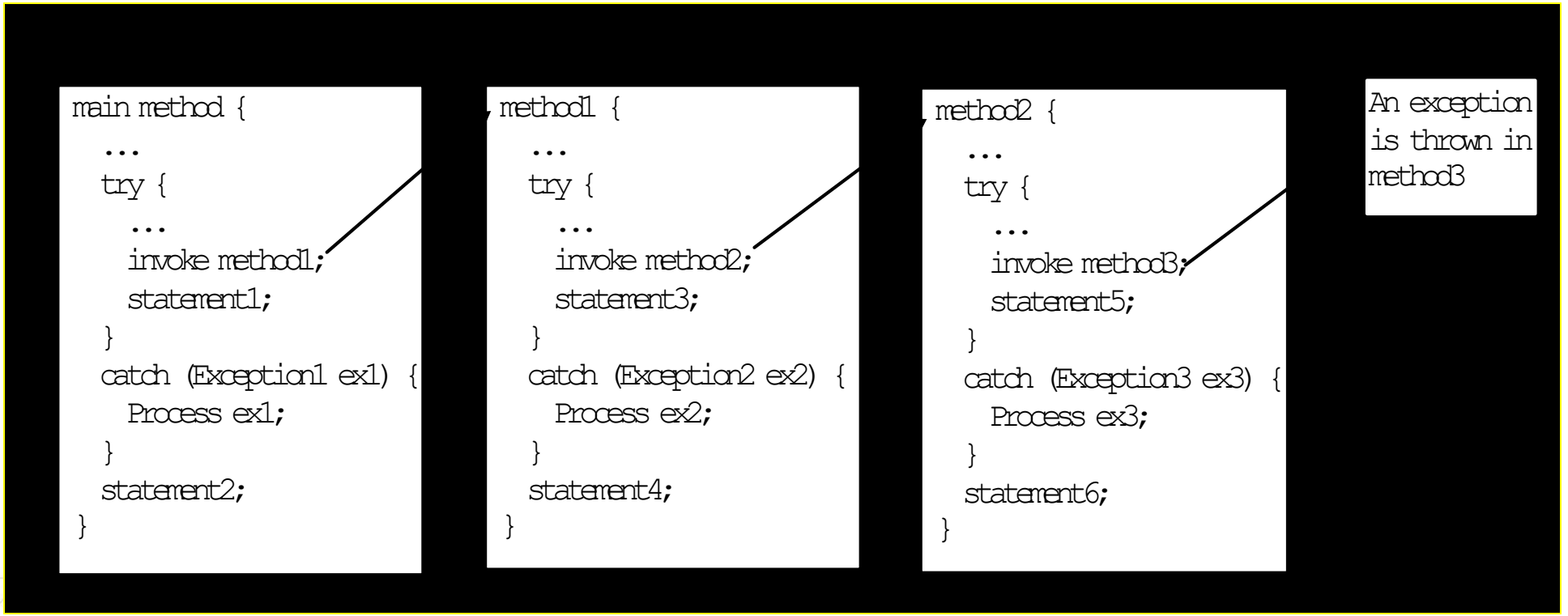

```
/** Set a new radius */
 public void setRadius(double newRadius)
     throws IllegalArgumentException {
   if (newRadius >= 0)
     radius =  newRadius;
   else
     throw new IllegalArgumentException(
       "Radius cannot be negative");
 }
```

When to Throw Exceptions

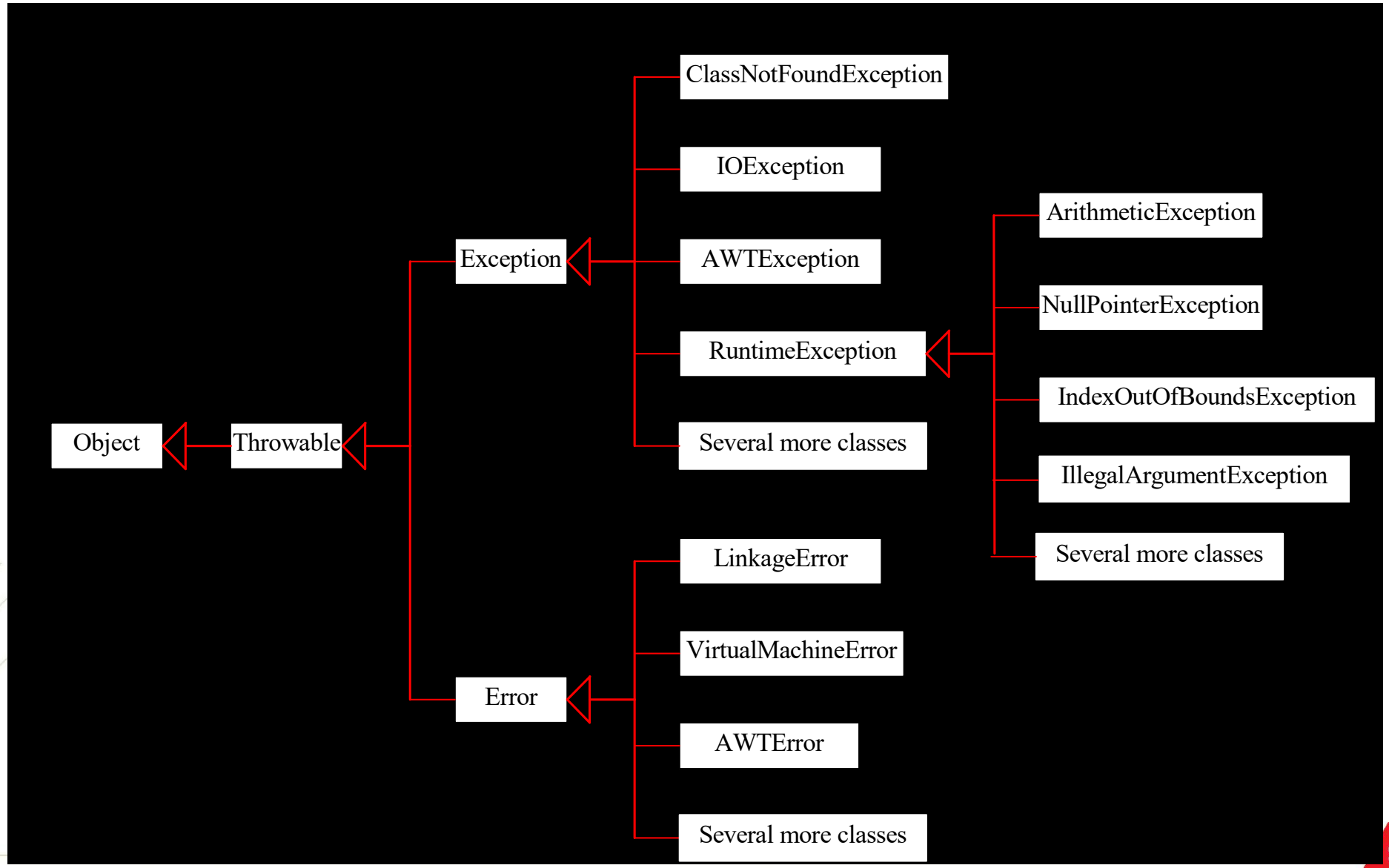
An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it.

If you can handle the exception in the method where it occurs, there is no need to throw it.

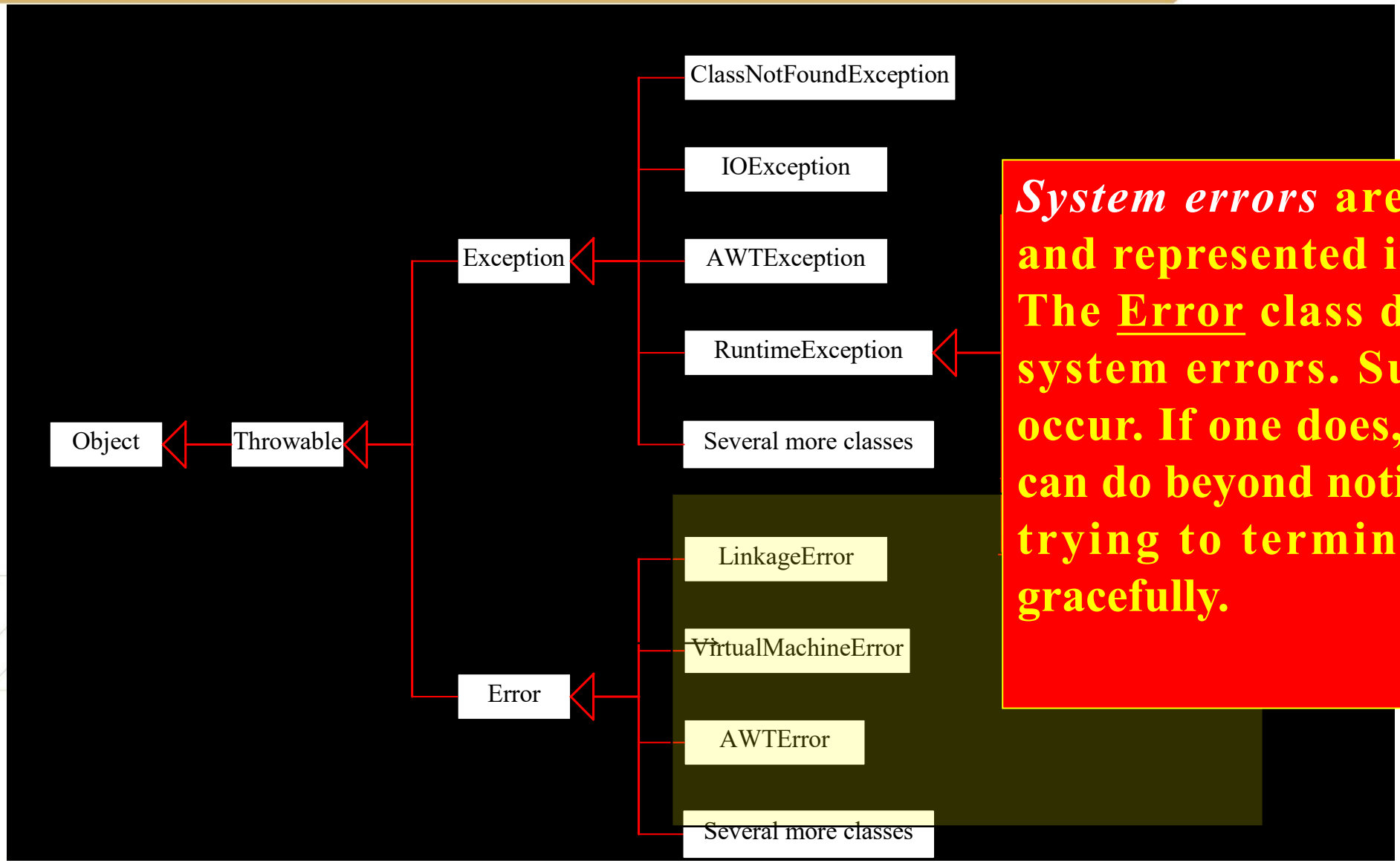
Catching Exceptions



Exception Classes

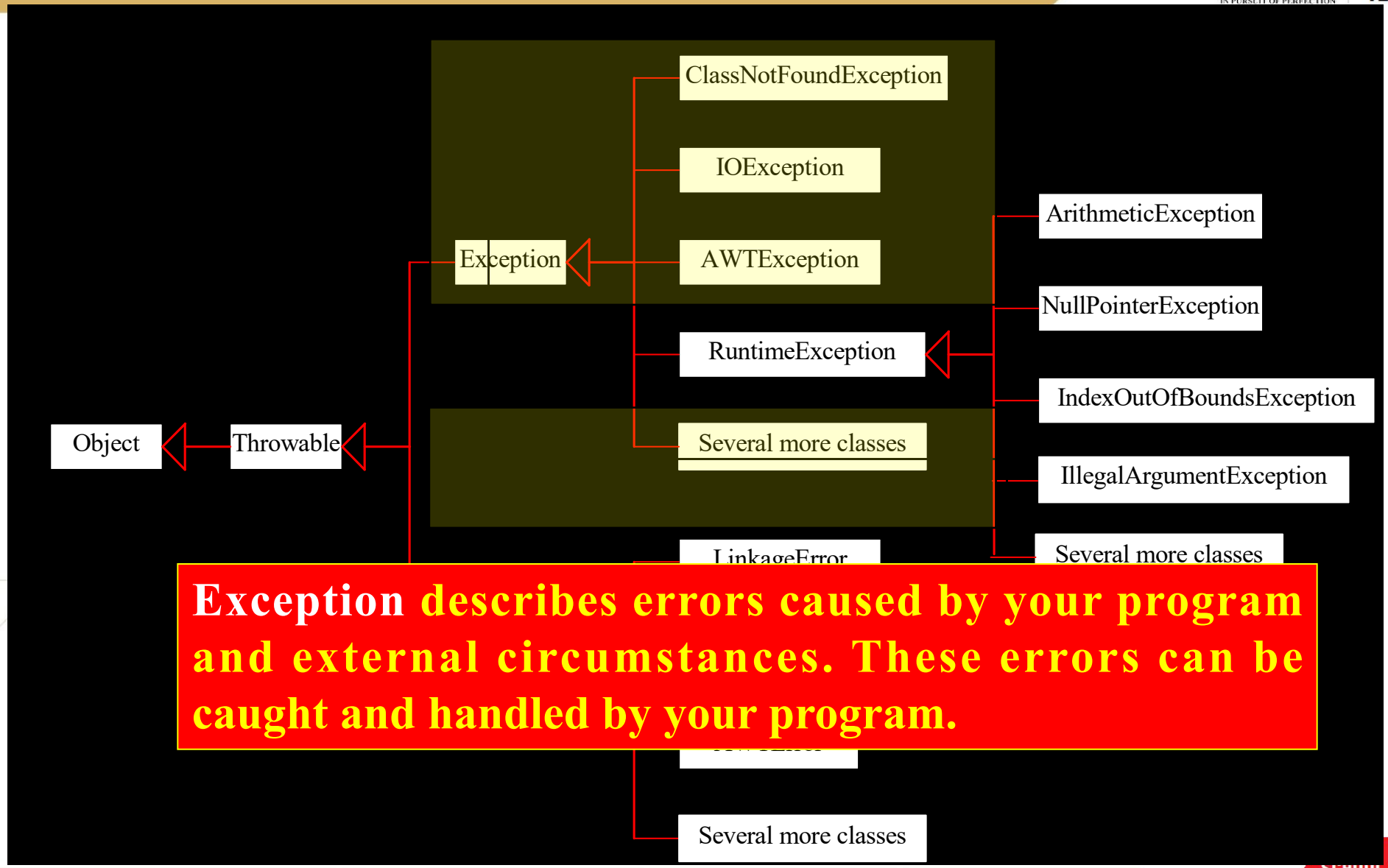


System Errors



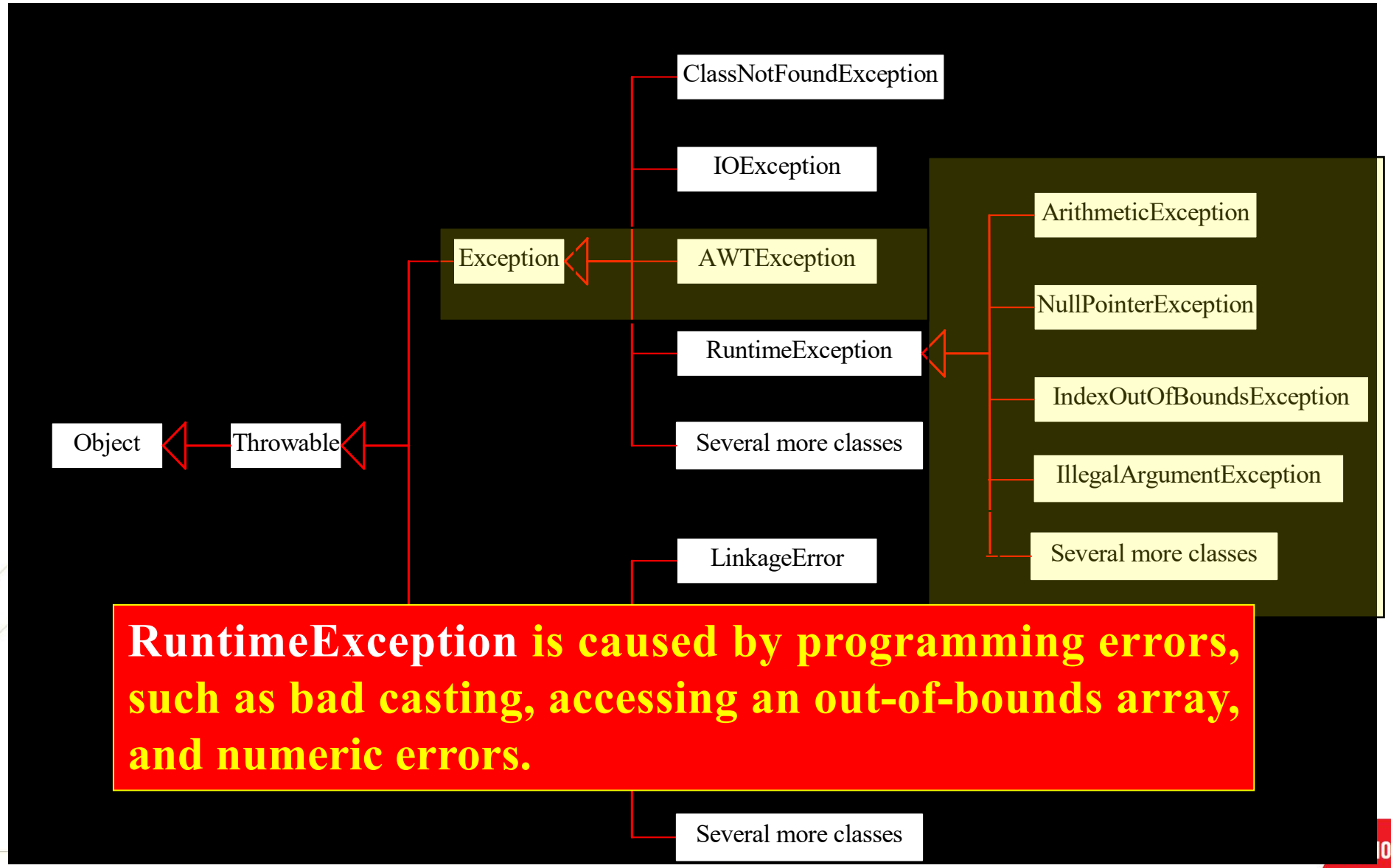
System errors **are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.**

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.

Checked Exceptions vs. Unchecked Exceptions

RuntimeException, Error and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Catch or Declare Checked Exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

Unchecked Exceptions

In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it; an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array. These are the logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

Java Catch Multiple Exceptions

Java Multi-catch block

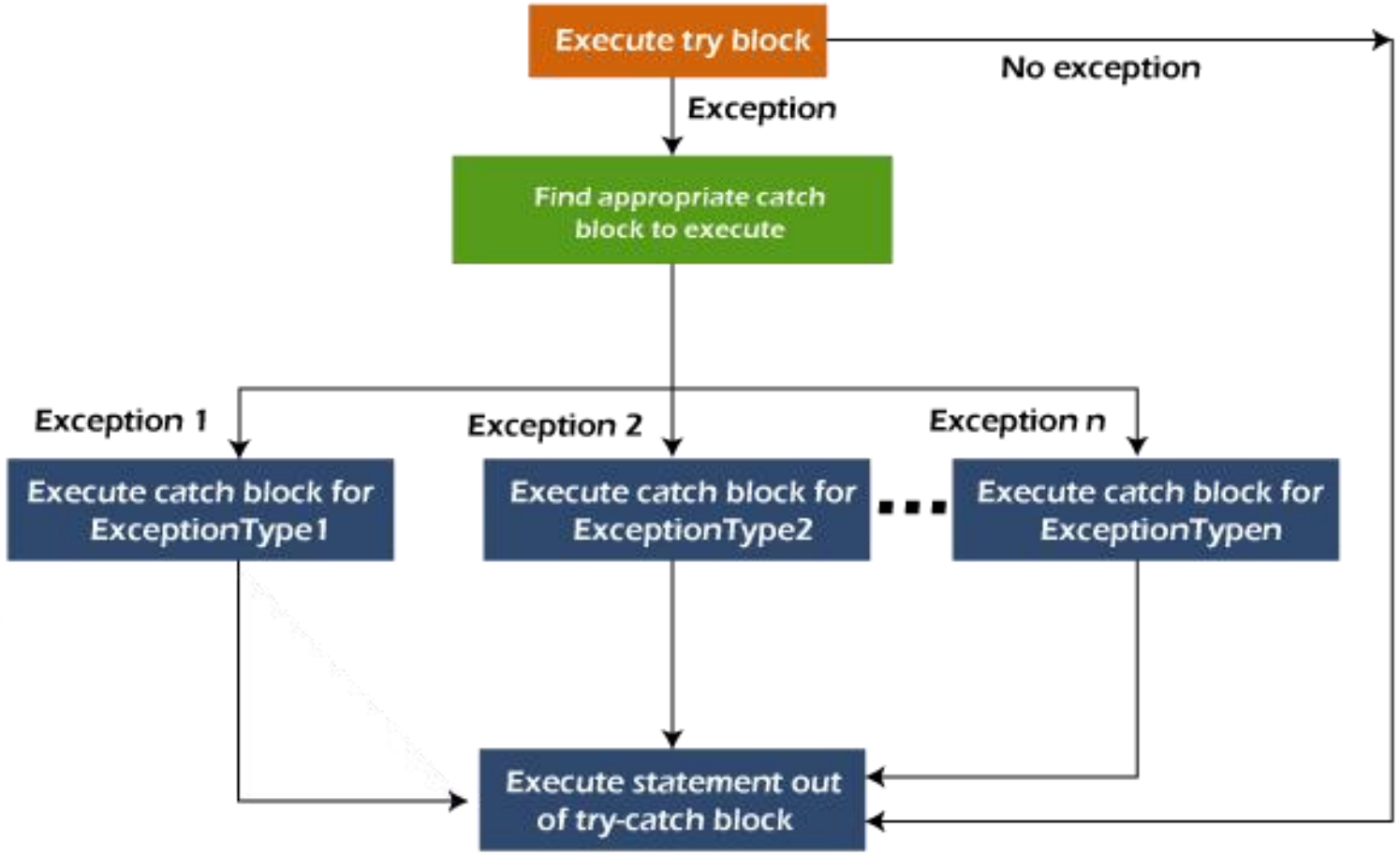
A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

At a time only one exception occurs 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 for ArithmeticException must come before catch for Exception.

Flowchart of Multi-catch Block



Multiple catch statements


```
/ Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
}
 catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException
e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
} }
```

Output:

```
C:\>java MultiCatch
a = 0
Divide by 0:
java.lang.ArithmeticException:
/ by zero
After try/catch blocks.
```

Why use Custom Exceptions?

- Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.
- To catch and provide specific treatment to a subset of existing Java exceptions.
- **Business logic exceptions**: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

Custom Exception

- Consider the following example, where we create a custom exception named WrongFileNameException:

```
public class WrongFileNameException extends Exception {

    public WrongFileNameException(String errorMessage) {

        super(errorMessage);
    }
}
```

Custom Exception Example


```
/ 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);
}}
```

This example defines a subclass of Exception called MyException. This subclass is quite simple: it has only a constructor plus an overloaded toString( ) method that displays the

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

Example 1:Custom Exception

Simple example of Java custom exception. In the following code, constructor of InvalidAgeException takes a string as an argument. This string is passed to constructor of parent class Exception using the super() method.

- Also the constructor of Exception class can be called without using a parameter and calling super() method is not mandatory.

TestCustomException1.java

Java Custom Exception

- In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

- Consider the example 1 in which InvalidAgeException class extends the Exception class.

- Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

Example Custom Exception


```
// class representing custom exception
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {
     // calling the constructor of parent Exception
      super(str);
   } }  // class that uses custom exception InvalidAgeException
public class TestCustomException1
{ // method to check the age
   static void validate (int age) throws InvalidAgeException{
     if(age < 18){
// throw an object of user defined exception
     throw new InvalidAgeException("age is not valid for  vote");      }
    else {
     System.out.println("welcome to vote");
     }     }     // main method
 public static void main(String args[])
 {   try
     {    // calling the method
       validate(13);  }
     catch (InvalidAgeException ex)
     {     System.out.println("Caught the exception");
    // printing the message from InvalidAgeException object
         System.out.println("Exception occured: " + ex);     }
System.out.println("rest of the code...");      } }
```

Output:

- Java Custom Exception

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Example 2:Custom Exception


```
// class representing custom exception
class MyCustomException extends Exception
{ }
// class that uses custom exception MyCustomException
public class TestCustomException2
{
  // main method
   public static void main(String args[])   {
      try
      { // throw an object of user defined exception
         throw new MyCustomException();
      }
   catch (MyCustomException ex)
        {
            System.out.println("Caught the exception");

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

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException2.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException2
Caught the exception
null
rest of the code...
```

Thank You