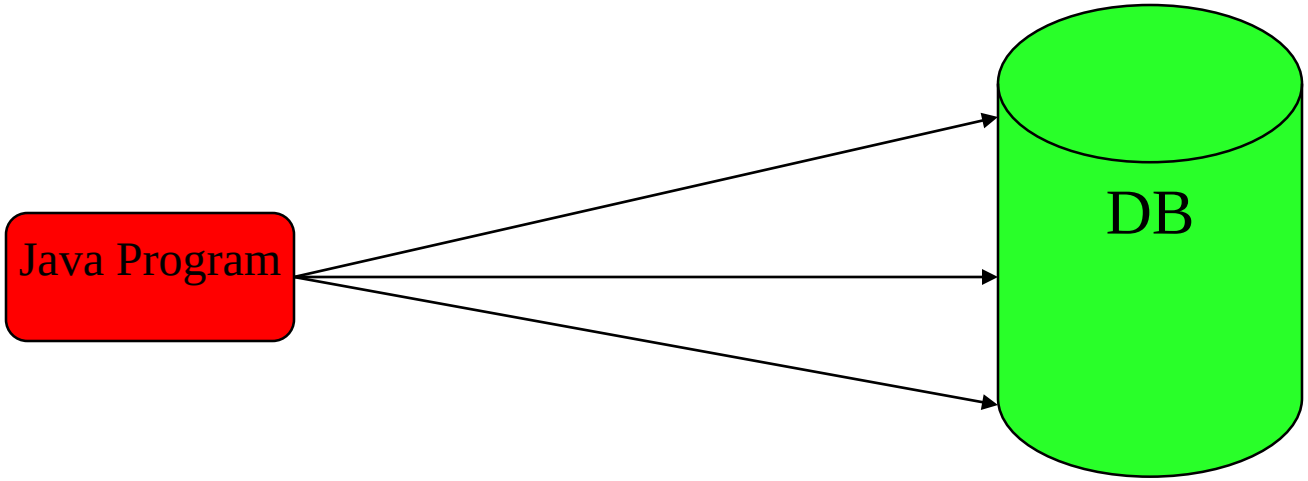
Java Exception Handling

What Is an Exception?

- An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- Main advantage is to gracefully terminate program.

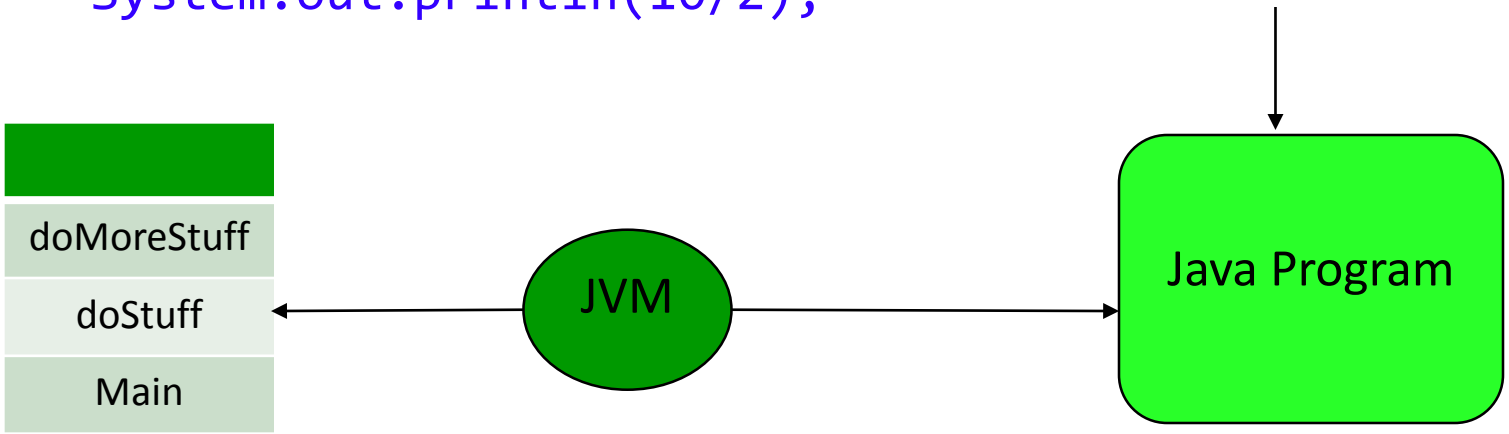


Note: Our goal is to make the program terminate gracefully and avoid any abnormal termination.

Runtime Stack Mechanism

```
public class NoHandling {
        public static void main(String[] args){
                doStuff();
        }
        public static void doStuff(){
                doMoreStuff();
        }
        public static void doMoreStuff(){
                System.out.println(10/2);
        }
}
```

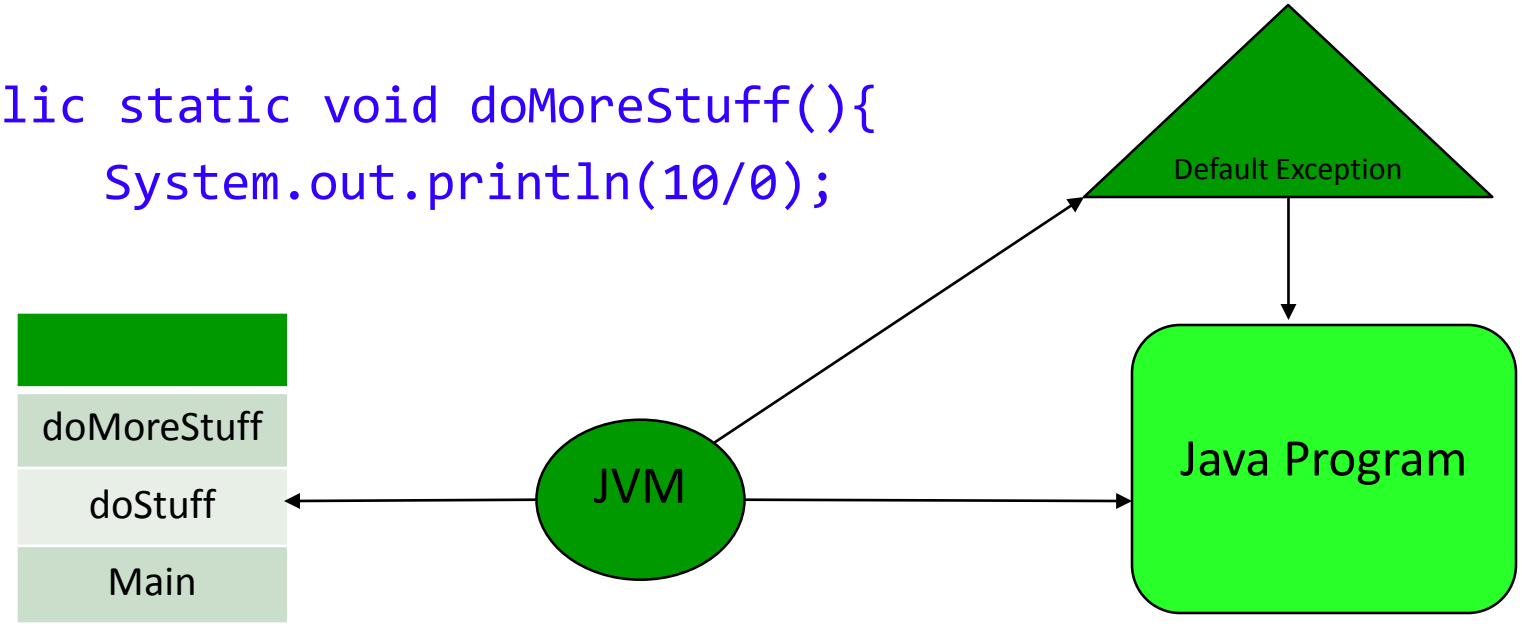
JVM create runtime stack for each method



Runtime Stack Mechanism

```
public class NoHandling {
        public static void main(String[] args){
                doStuff();
        }
        public static void doStuff(){
                doMoreStuff();
        }
        public static void doMoreStuff(){
                System.out.println(10/0);
        }
}
```

Note: If there is at least one method terminate abnormally then the whole program will terminate abnormal termination.

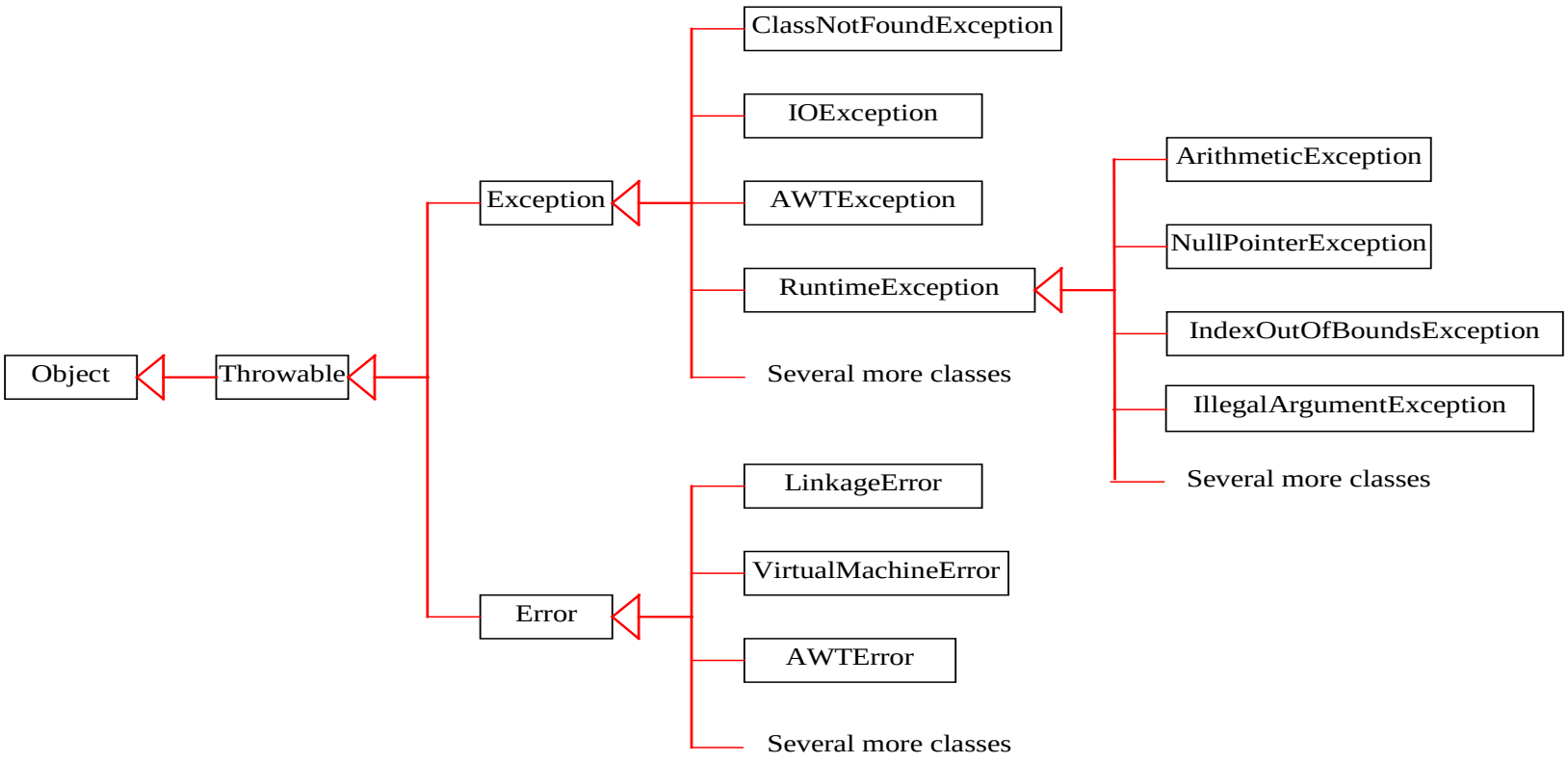


JVM create runtime stack for each method

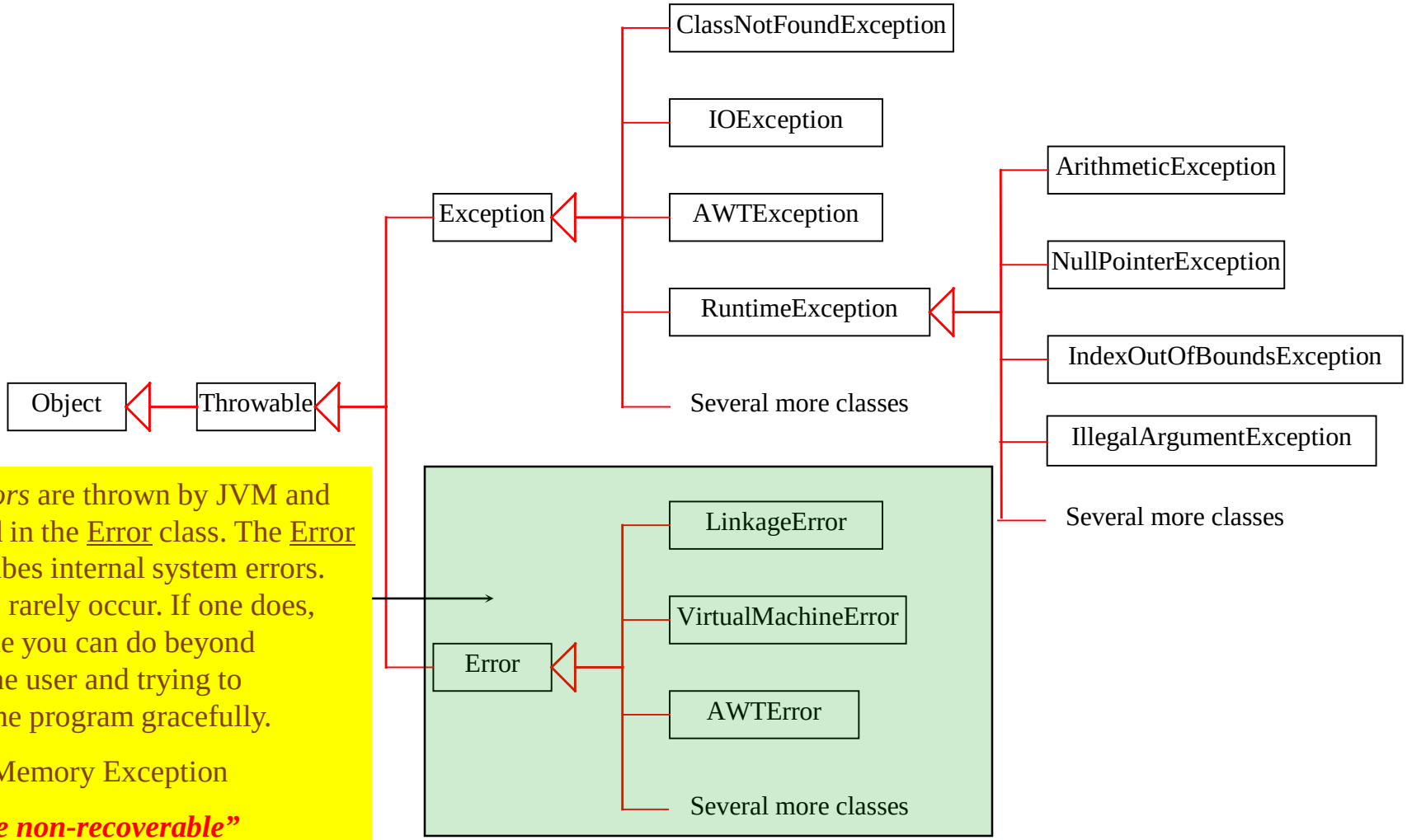
Runtime Stack Mechanism

- If an error happen inside a method, the same method will be responsible to create exception object that is include the following information:
  - Name of exception
  - Description of exception
  - Location at which exception occurs. [stack trace]
- After creating the exception object the method hand over this object to the JVM.
- JVM will check whether the method contains any exception handling code or not.
- If the method doesn't have acceptation handling code, then JVM terminate that method abnormally and remove the corresponding entry from the stack.
- JVM will call the caller method and checks whether the caller method contains any handling code or not.
- If the caller method doesn't have handling code the JVM terminate that caller method abnormally and remove the corresponding entry from the stack.
- This process will continue until the JVM reaches the main method.
- If the main method doesn't contain handling code then JVM will terminate the main method also abnormally and remove the corresponding entry from the stack.
- JVM handover the responsibility of the exception handling to the default exception handler, which is part of JVM.

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.

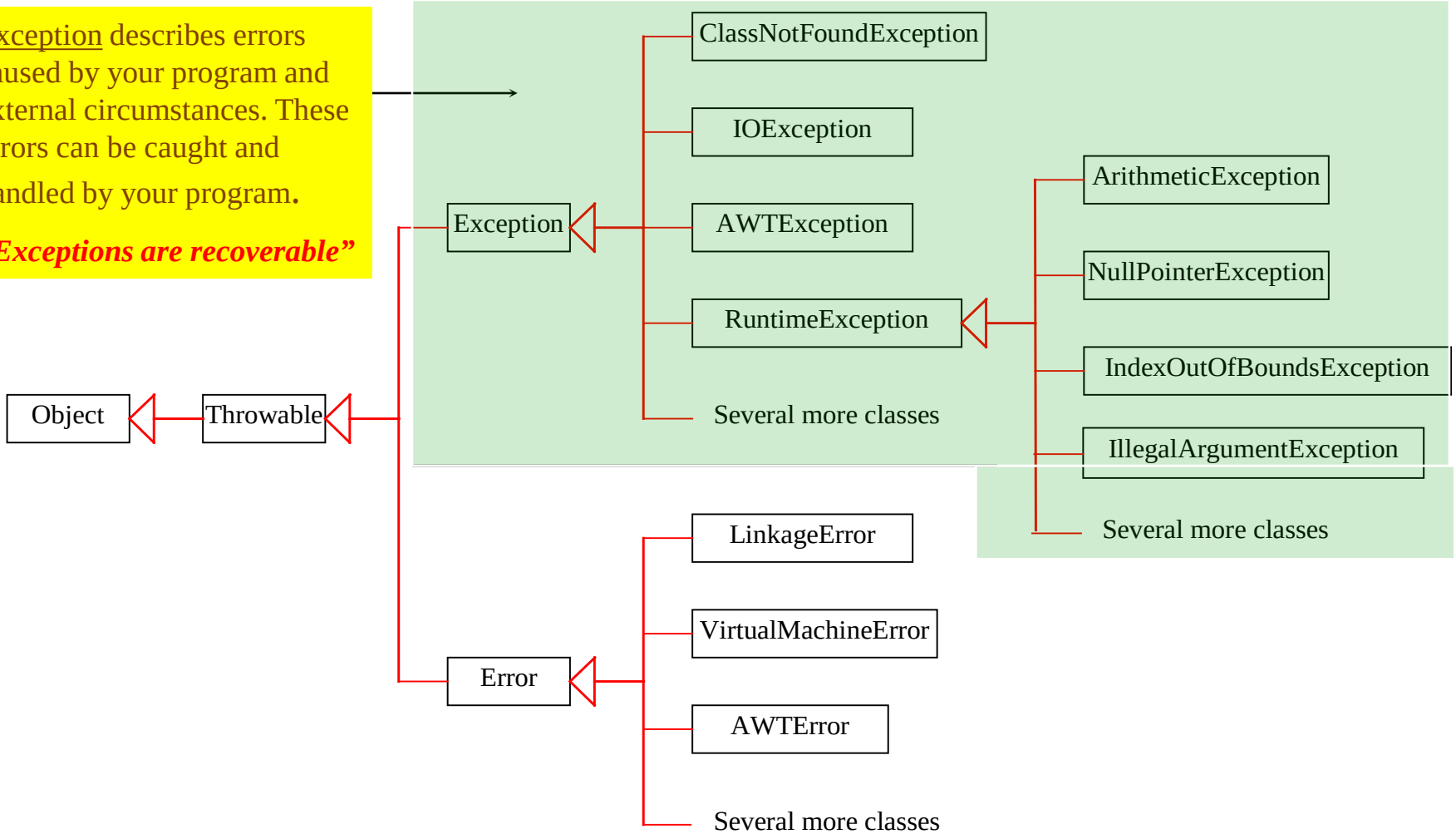
Ex: OutOfMemory Exception

"Errors are non-recoverable"

Exceptions

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

"Exceptions are recoverable"



Checked VS 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.

Checked 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.

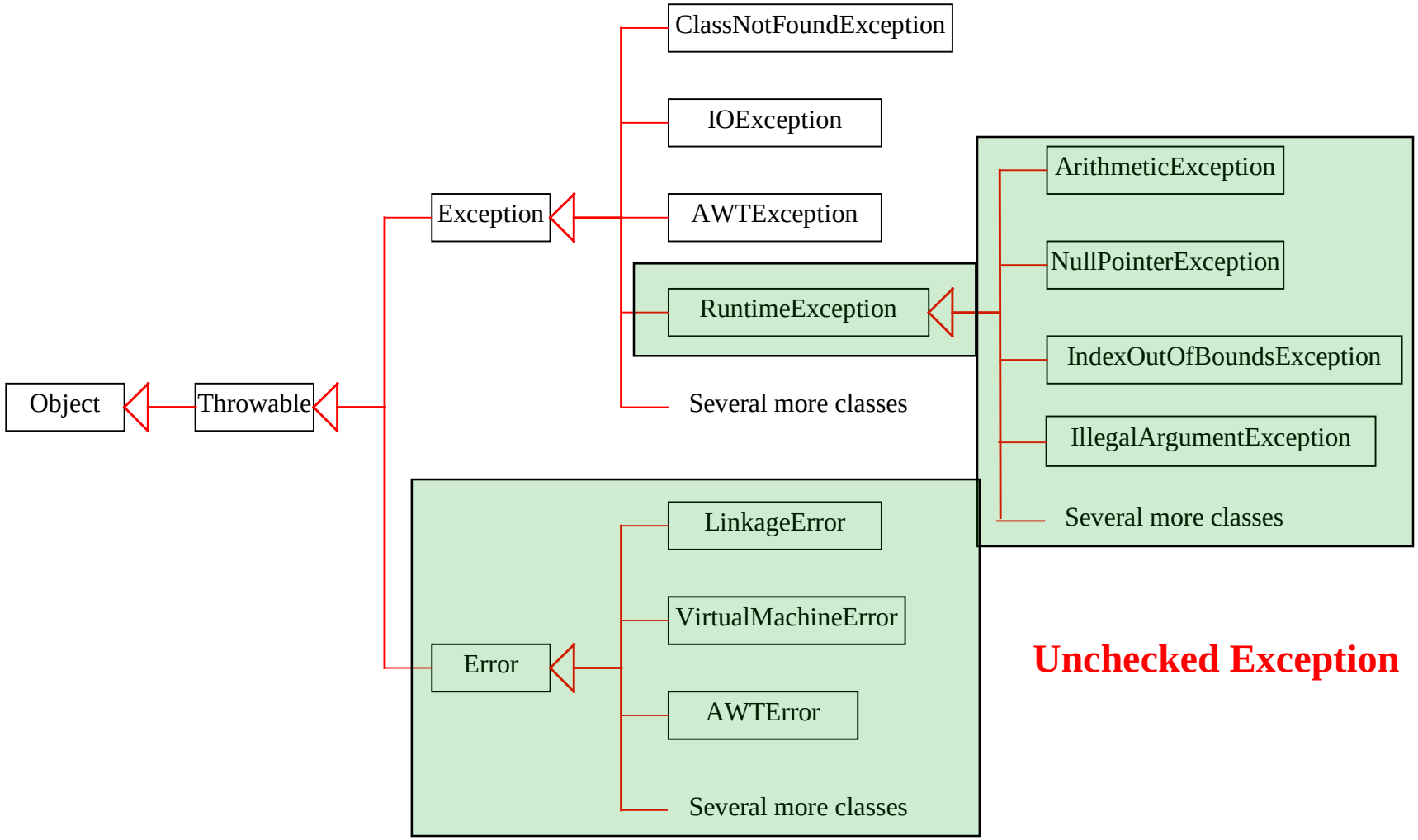
Checked VS Unchecked Exceptions

```
public class Example1 {
        public static void main(String[] args){
                PrintWriter pw = new PrintWriter("abc.txt");
                pw.println("hello");
                System.out.println(10/0);
        }
}
```

This is RunningTimeException but is not detected during the compile time.

Unhandled exception type FileNotFoundException

Checked VS Unchecked Exceptions



Fully Checked vs Partially Checked

- A checked exception is set to fully checked if and only if all it's child classes are checked.
  - example: IOException and AWTException .

- A checked exception is set to partially checked if and only if some of it's child classes are unchecked.
  - Example: Exception and Throwable.

Checked, Unchecked, Fully Checked vs Partially Checked

- Describe the behavior of the following exceptions:
  - IOException: checked (checked (fully checked))
  - RuntimeException: (unchecked)
  - InterruptedException: checked (checked (fully checked))
  - Error (unchecked)
  - Throwable (checked (partially checked))
  - ArithmeticException (unchecked)
  - NullPointerException (unchecked)
  - Exception (checked (partially checked))
  - FileNotFoundException (checked (fully checked))

Exception Handling

Keywords:

Try/Catch

```
public class HandlingExceptions {
      public static void main(String[] args){

            System.out.println("statement 1");
            System.out.println(10/0);
            System.out.println("statement 2");
      }
}
```

Risky Code → Abnormal Termination

This line will never be executed

Try/Catch

```
public class HandlingExceptions {
        public static void main(String[] args){
                System.out.println("statement 1");
                try{
                        System.out.println(10/0);
                }
                catch (ArithmeticException e){
                        System.out.println(10/2);
                }
                System.out.println("statement 2");
        }
}
```

The “Risky Code” is now handled and it became not risky

This statement will execute gracefully!

Try/Catch

```
public class HandlingExceptions {
        public static void main(String[] args){
                try{
                        System.out.println("statement 1");
                        System.out.println(10/0);
                        System.out.println("statement 2");
                }
                catch (ArithmeticException e){
                        System.out.println(10/2);
                }
                System.out.println("statement 3");
        }
}
```

This line will never be executed.

Output:

statement 1
5
statement 3

Try/Catch

```
public class HandlingExceptions {
        public static void main(String[] args){
                try{
                        System.out.println("statement 1");
                        System.out.println(10/0);
                        System.out.println("statement 2");
                }
                catch (ArithmeticException e){
                        System.out.println(20/0);
                }
                System.out.println("statement 3");
        }
}
```

Output: ?

Try/Catch

```
public class HandlingExceptions {
        public static void main(String[] args){
                try{
                        System.out.println("statement 1");
                        System.out.println("statement 2");
                        System.out.println("statement 3");
                }
                catch (ArithmeticException e){
                        System.out.println(10/2);
                }
                System.out.println(10/0);
        }
}
```

Output: ?

Try/Catch (Bad Approach)

```
public class HandlingExceptions {
        public static void main(String[] args){
                try{
                        System.out.println("statement 1");
                        System.out.println(10/0);
                        System.out.println("statement 2");
                        System.out.println("statement 3");
                        System.out.println("statement 4");
                        System.out.println("statement 5");
                }
                catch (ArithmeticException e){
                        System.out.println(10/2);
                }
        }
}
```

Try With Multiple Catch Blocks

```
public class HandlingExceptions {
        public static void main(String[] args){

                try{
                        System.out.println("statement 1");
                        System.out.println(10/0);
                        System.out.println("statement 2");
                }
                catch (ArithmeticException e){
                        System.out.println(10/2);
                }
                catch(SQLException se){
                        // do stuff here..
                        se.printStackTrace();
                }
                catch (FileNotFoundException e){
                    // do stuff here..
                    System.out.println("File is not found");
                }
                catch (Exception e){
                    // do stuff here..
                    e.getCause();
                }
        }
}
```

We only use multiple catch block if We don't know the type of exception that may rise

Try/Catch

```
public class HandlingExceptions {
        public static void main(String[] args){

                try{
                        System.out.println("statement 1");
                        System.out.println(10/0);
                        System.out.println("statement 2");
                }
                catch (ArithmeticException e){
                        System.out.println(10/2);
                }
                catch(SQLException se){
                        // do stuff here..
                        se.printStackTrace();
                }
                catch (Exception e){
                    // do stuff here..
                    e.getCause();
                }
                catch (FileNotFoundException e){
                    // do stuff here..
                    System.out.println("File is not found");
                }
        }
}
```

What is the problem In this code?

Try/Catch

```
public class HandlingExceptions {
        public static void main(String[] args){

                try{
                        System.out.println("statement 1");
                        System.out.println(10/0);
                        System.out.println("statement 2");
                }
                catch (ArithmeticException e){
                        System.out.println(10/2);
                }
                catch(SQLException se){
                        // do stuff here..
                        se.printStackTrace();
                }
                catch (Exception e){
                    // do stuff here..
                    e.getCause();
                }
                catch (FileNotFoundException e){
                    // do stuff here..
                    System.out.println("File is not found");
                }
        }
}
```

This code will never be executed!
You have to take the child first and then
the parent, otherwise you will get compiler error

Final, Finally, Finalize

- Final is modifier applicable for classes, methods and variables.
  - If a class declared as final, then we can't extend that class (we can't create child classes for that class).
  - If a method declared as final then we can't overwrite that method in the child class.
  - If a variable declared as final then we can't perform reassignment for that variable.

Final, Finally, Finalize

- Finally is a block always associated with try/catch to maintain cleanup code.
- The specialty of finally block is to always execute in respective of if the exception is rise or not and the method handle or not handle.

```
try {
        //Risky Code
    }

Catch(EX e) {
       //Handling Code
    }

Finally {
        //Cleanup Code
    }
```

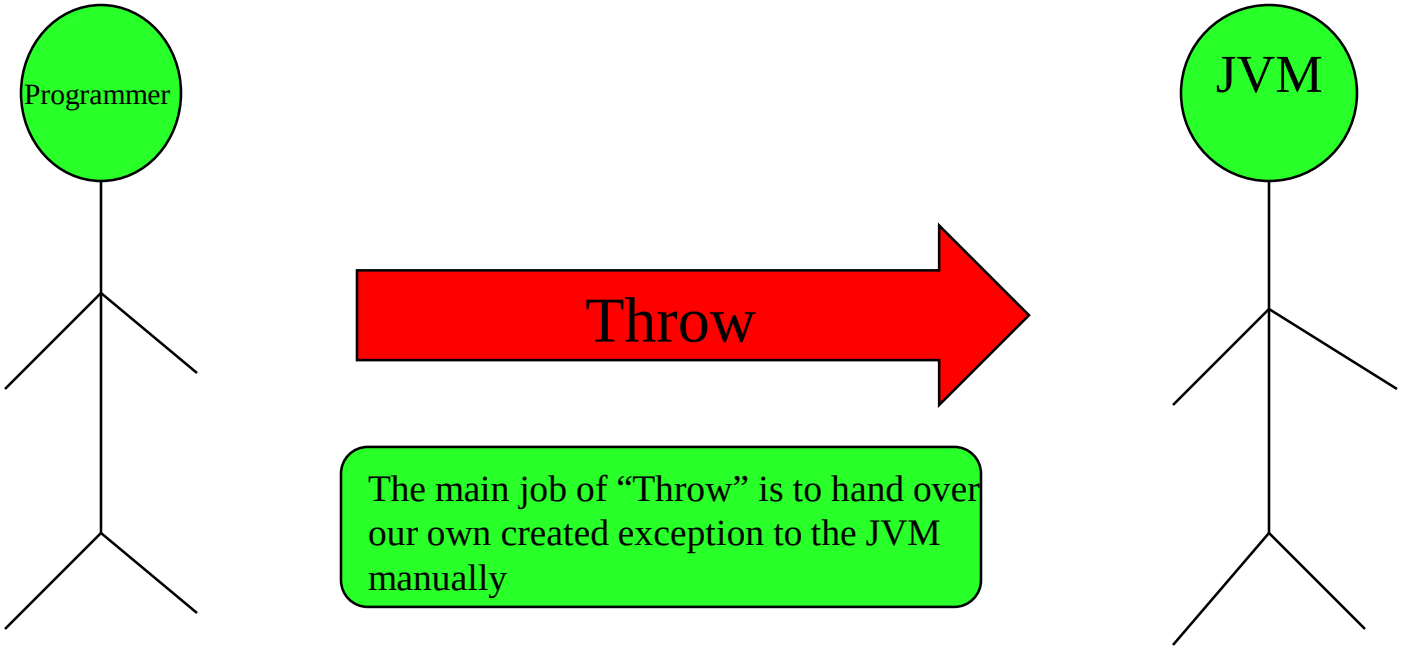
Final, Finally, Finalize

- finalize() is a method presented in the object class.
- The finalize() method is always invoked by the garbage collector just before destroying an object to perform cleanup activates.
- Once finalize() method complete, immediately garbage collector destroy that object

Try/Catch/Finally

- Various combination of try, catch, finally
  - The order is important.
  - Whenever we are writing “try” it’s mandatory that we write “catch” or “finally” otherwise we get compile time error. That is “try” without “catch/finally” is invalid.
  - Whenever we are writing “catch” block the “try” block is mandatory, that is catch without try is invalid
  - Whenever we are writing “finally” block, it’s mandatory to write “try” block that is “finally” without try is invalid.
  - Inside “try, catch, finally” block we can declare try/catch and finally block.

Throw and Throws



Throw

```
public static void main(String[] args){
    System.out.println(10/0);
}
```

The main method is responsible to create exception and hand over to the JVM

Throw

```
public static void main(String[] args){
        throw new ArithmeticException ("/0 is not allowed");
}
```

In this case the program create exception object explicitly and hand over to the JVM manually.

Throw

```
public static withdraw(double amount){
        if (amount > balance )
                Throw new InsufficientResourcesException ();
        }
}
```

Throw is the best option for user defined or customize exception!

Throw

- **Case 1:**

```
public static void test (){
                throw new ArithmeticException("Insufficient Funds");
                System.out.println("hello");
        }
```

We won't be able to write any statement directly after the "throw" statement. The compiler will throw "Unreachable code" error.

Throw

Case 1:

```
public static void test (){
        throw new ArithmeticException("Insufficient Funds");
                System.out.println("hello");
        }
```

VS

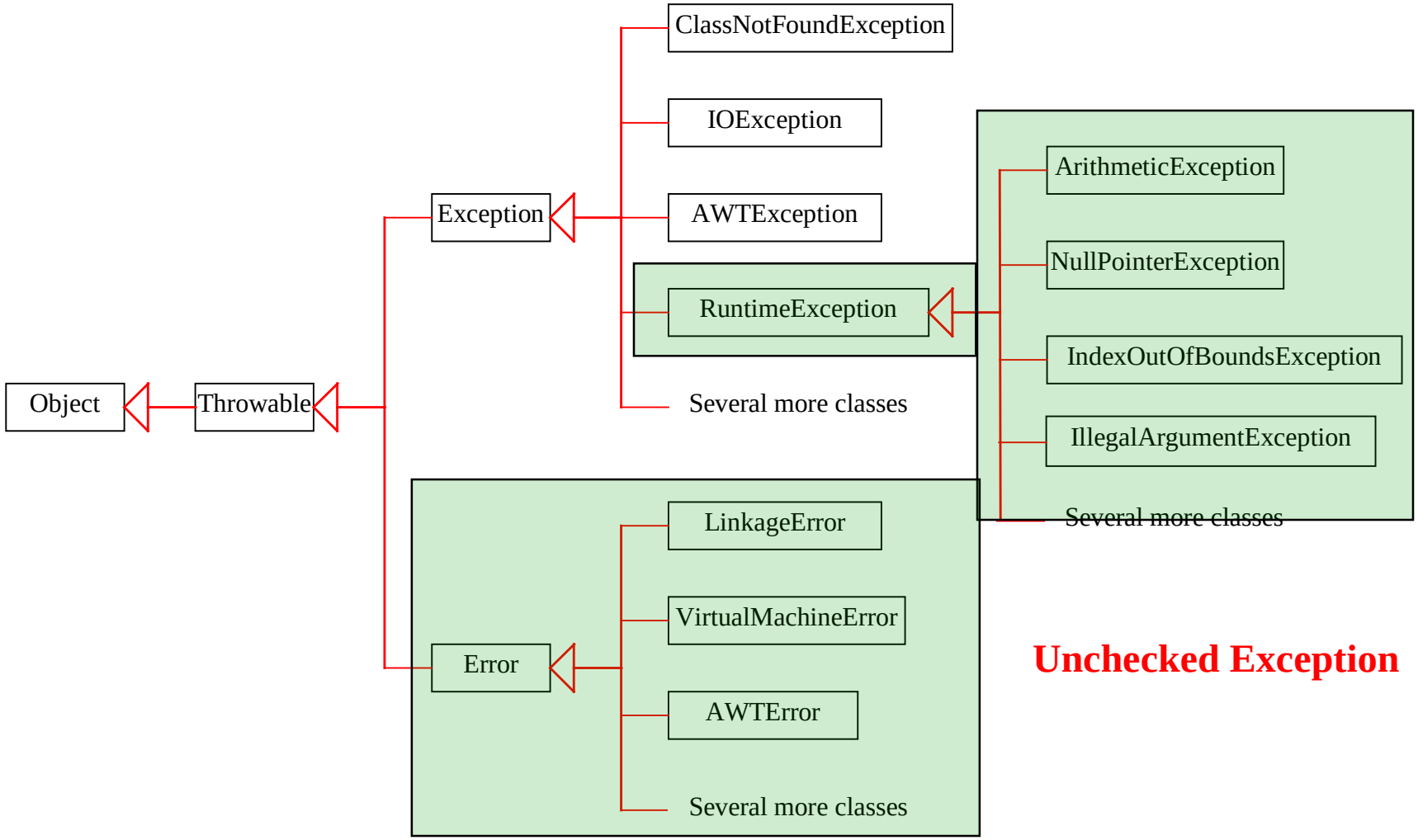
```
public static void test2 (){
                System.out.println(10/0);
                System.out.println("hello");
        }
```

Use “Throw” to develop personalize exceptions

- **Case 2:**

```
public class Test extends RuntimeException{
      public static void main(String[] args){
            throw new Test ();
      }
}
```

Checked VS Unchecked Exceptions



Which one of these is going to compile?

```
public class Test {
        public static void main(String[] args) {
                throw new Error();
        }
}
```

```
public class ThrowsTest {
        public static void main(String[] args) {
                throw new Exception();
        }
}
```

Throws

```
import java.io.*;
public class ThrowsTest {
      public static void main(String[] args){
            PrintWriter pw = new PrintWriter("abc.txt");
            pw.println("hello");
      }
}
```

If there is any possibility to rise checked exception, then it's mandatory to handle that exception, otherwise we will get compile error saying unreported exception must be caught or declared to be thrown

Throws

```
import java.io.*;
public class ThrowsTest {
        public static void main(String[] args){
                try{
                        PrintWriter pw = new PrintWriter("abc.txt");
                        pw.println("hello");
                }
                catch (IOException e){
                        System.out.println("File not found");
                }
        }
}
```

Throws

```
import java.io.*;
public class ThrowsTest {
        public static void main(String[] args) throws IOException {
                PrintWriter pw = new PrintWriter("abc.txt");
                pw.println("hello");
        }
}
```

We can use the “Throws” keyword to delegate the responsibility of handling the exception to the caller. The caller can be a method or JVM. If the

Throws

```
public class NoHandling {
        public static void main(String[] args){
                doStuff();
        }
        public static void doStuff(){
                doMoreStuff();
        }
        public static void doMoreStuff() throws IOException{
                PrintWriter pw = new PrintWriter("abc.txt");
                pw.println("hello");
        }
}
```

Is this going to compile?

Throws

```
import java.io.IOException;
import java.io.PrintWriter;
public class NoHandling {
        public static void main(String[] args) throws IOException{
                doStuff();
        }
        public static void doStuff()throws IOException{
                doMoreStuff();
        }
        public static void doMoreStuff() throws IOException {
                PrintWriter pw = new PrintWriter("abc.txt");
                pw.println("hello");
        }
}
```

Throws

Is this code going to compile?

```
import java.io.IOException;
import java.io.PrintWriter;
public class NoHandling {
        public static void main(String[] args) throws IOException{
                doStuff();
        }
        public static void doStuff(){
                doMoreStuff();
        }
        public static void doMoreStuff() {
                PrintWriter pw = new PrintWriter("abc.txt");
                pw.println("hello");
        }
}
```

Throws/Conclusion

- Throws can be used to delegate the responsibility of exception handling to the caller (it maybe method or JVM)
- The Throws keyword is required only for checked exception and usage of Throws for unchecked exception is not needed.
- The usage of the Throws keyword is only to convince the compiler and it doesn't prevent abnormal termination.
- The Throws keyword cab be used for methods and constructers but not for classes.

Which one of the code below is NOT going to compile?

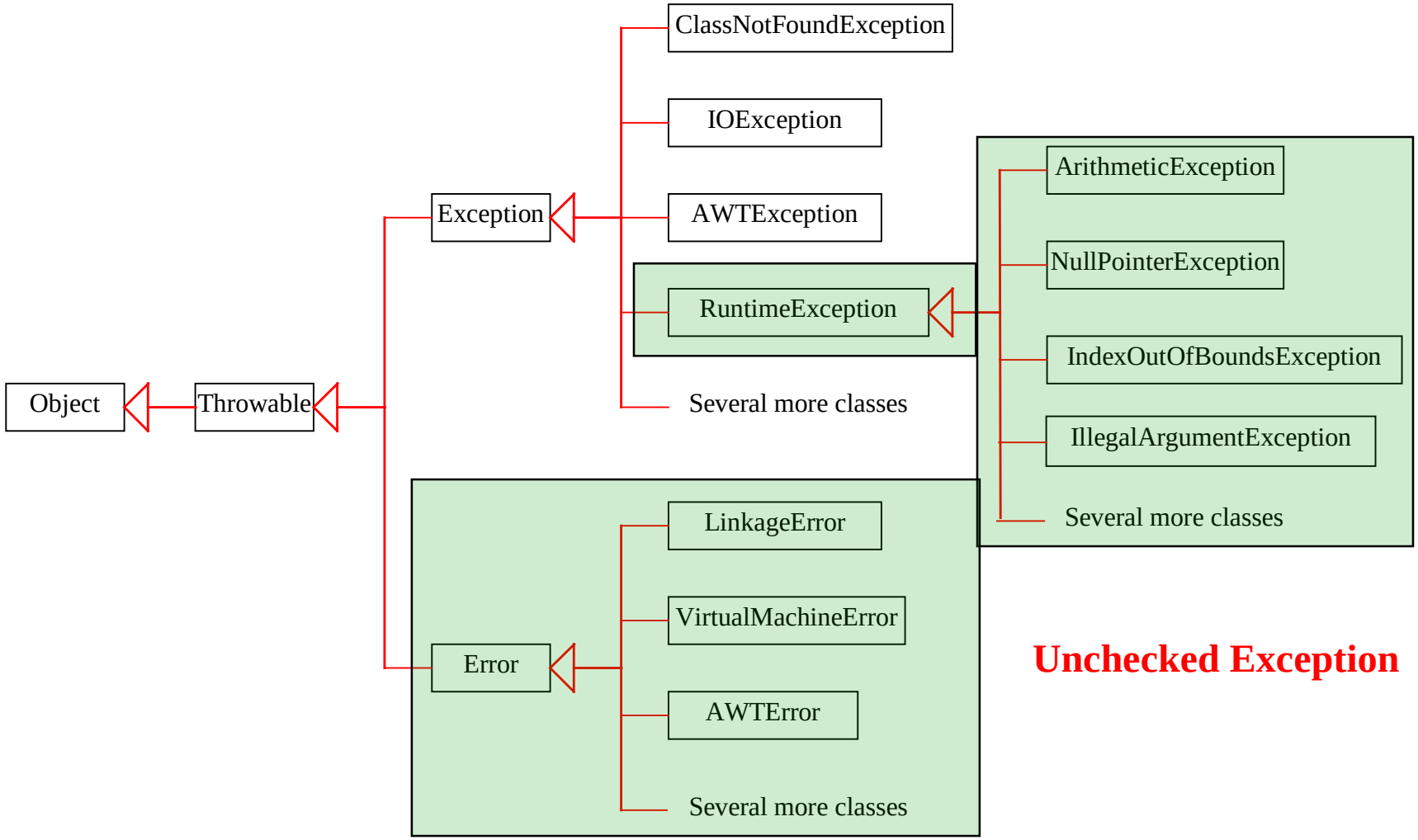
```
import java.io.*;
public class ThrowsTest {
    public static void main(String[] args){
       try{
          System.out.println("hello");
          }
          catch(ArithmeticException e){
                   e.getStackTrace();
              }
    }
}
```

```
import java.io.*;
public class ThrowsTest {
   public static void main(String[] args){
      try{
         System.out.println("hello");
         }
         catch(Exception e){
                  e.getStackTrace();
             }
     }
}
```

```
import java.io.*;
public class ThrowsTest {
   public static void main(String[] args){
      try{
         System.out.println("hello");
         }
         catch(Error e){
                  e.getStackTrace();
         }
   }
}
```

```
import java.io.*;
public class ThrowsTest {
   public static void main(String[] args){
      try{
         System.out.println("hello");
         }
         catch(IOException e){
                  e.getStackTrace();
            }
    }
}
```

Checked VS Unchecked Exceptions



Which one of the code below is NOT going to compile?

```
import java.io.*;
public class ThrowsTest {
    public static void main(String[] args){
       try{
         System.out.println("hello");
         }
         catch(ArithmeticException e){
                  e.getStackTrace();
             }
    }
}
```

```
import java.io.*;
public class ThrowsTest {
   public static void main(String[] args){
      try{
         System.out.println("hello");
         }
         catch(Exception e){
                  e.getStackTrace();
             }
     }
}
```

```
import java.io.*;
public class ThrowsTest {
   public static void main(String[] args){
      try{
         System.out.println("hello");
         }
         catch(Error e){
                  e.getStackTrace();
         }
   }
}
```

```
import java.io.*;
public class ThrowsTest {
   public static void main(String[] args){
      try{
         System.out.println("hello");
         }
         catch(IOException e){
                  e.getStackTrace();
            }
    }
}
```

Conclusion

- Try: To maintain risky code.
- Catch: To maintain exception handling code.
- Finally: To maintain cleanup code.
- Throw: To hand over our created exception object to the JVM manually.
- Throws: To delegate responsibility of exception handling to the caller.