# Exception Handling in Java

# Errors

- Syntax errors
  - arise because the rules of the language have not been followed.
  - detected by the compiler.
- *Logic errors*
  - leads to wrong results and detected during testing.
  - arise because the logic coded by the programmer was not correct.
- *Runtime errors*
  - Occur when the program is running and the environment detects an operation that is impossible to carry out.

# Errors

- Code errors
  - Divide by zero
  - Array out of bounds
  - Integer overflow
  - Accessing a null pointer (reference)

  - Programs *crash* when an exception goes untrapped, i.e., not handled by the program.

# Runtime Errors

```
1  import java.util.Scanner;
2
3  public class ExceptionDemo {
4    public static void main(String[] args) {
5      Scanner scanner = new Scanner(System.in);
6      System.out.print("Enter an integer: ");
7      int number = scanner.nextInt();
8
9      // Display the result
10     System.out.println(
11       "The number entered is " + number);
12   }
13 }
```

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

Terminated.

# Exception

An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

Exception = Exceptional Event

# Exception Handling

Java exception handling is a mechanism for handling exception by *detecting* and *responding* to exceptions in a systematic, uniform and reliable manner.

Any exceptions not specifically handled within a Java program are caught by the Java run time environment

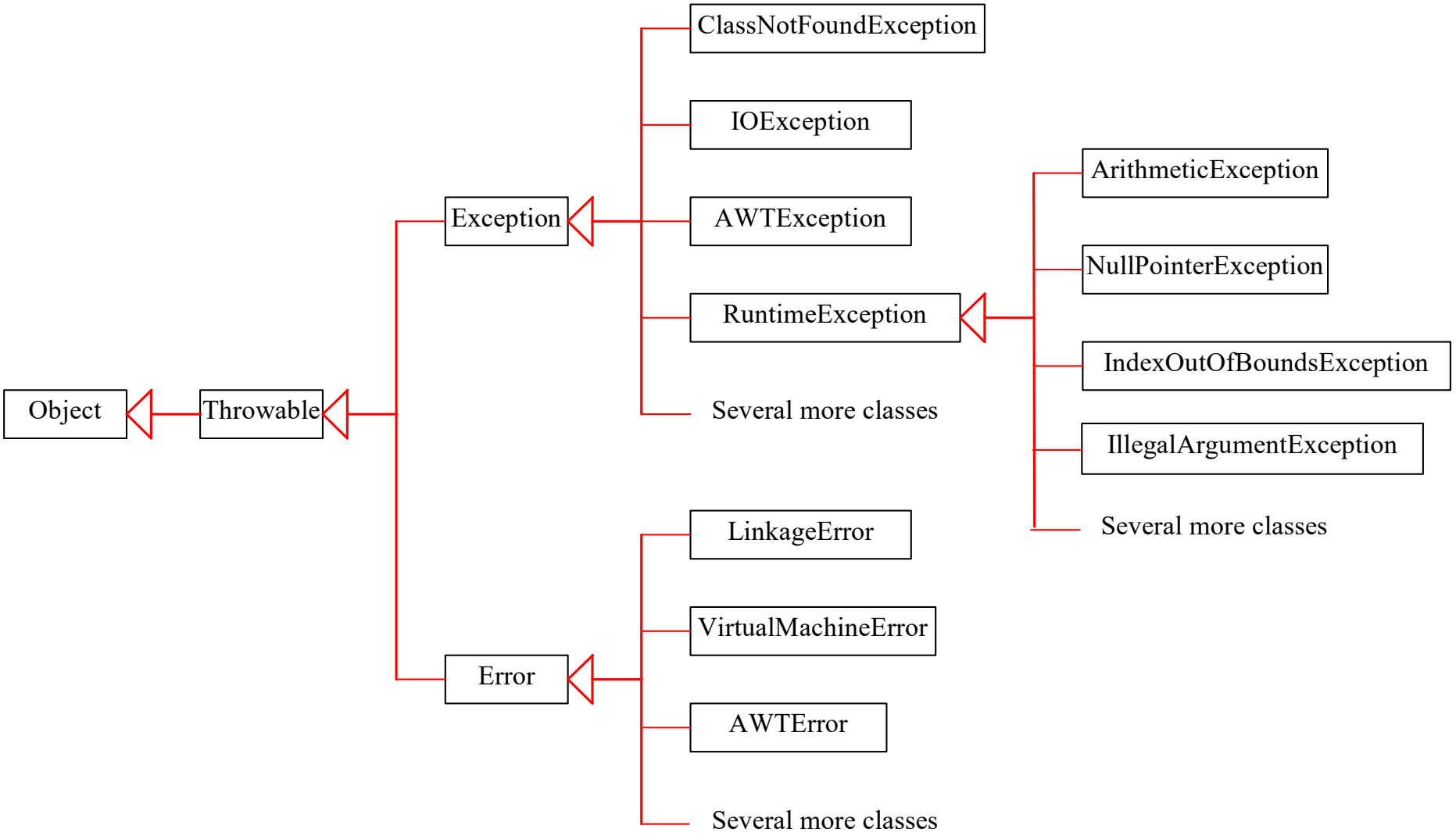
# Exceptions

- A Method in Java throws exceptions to tell the calling code:

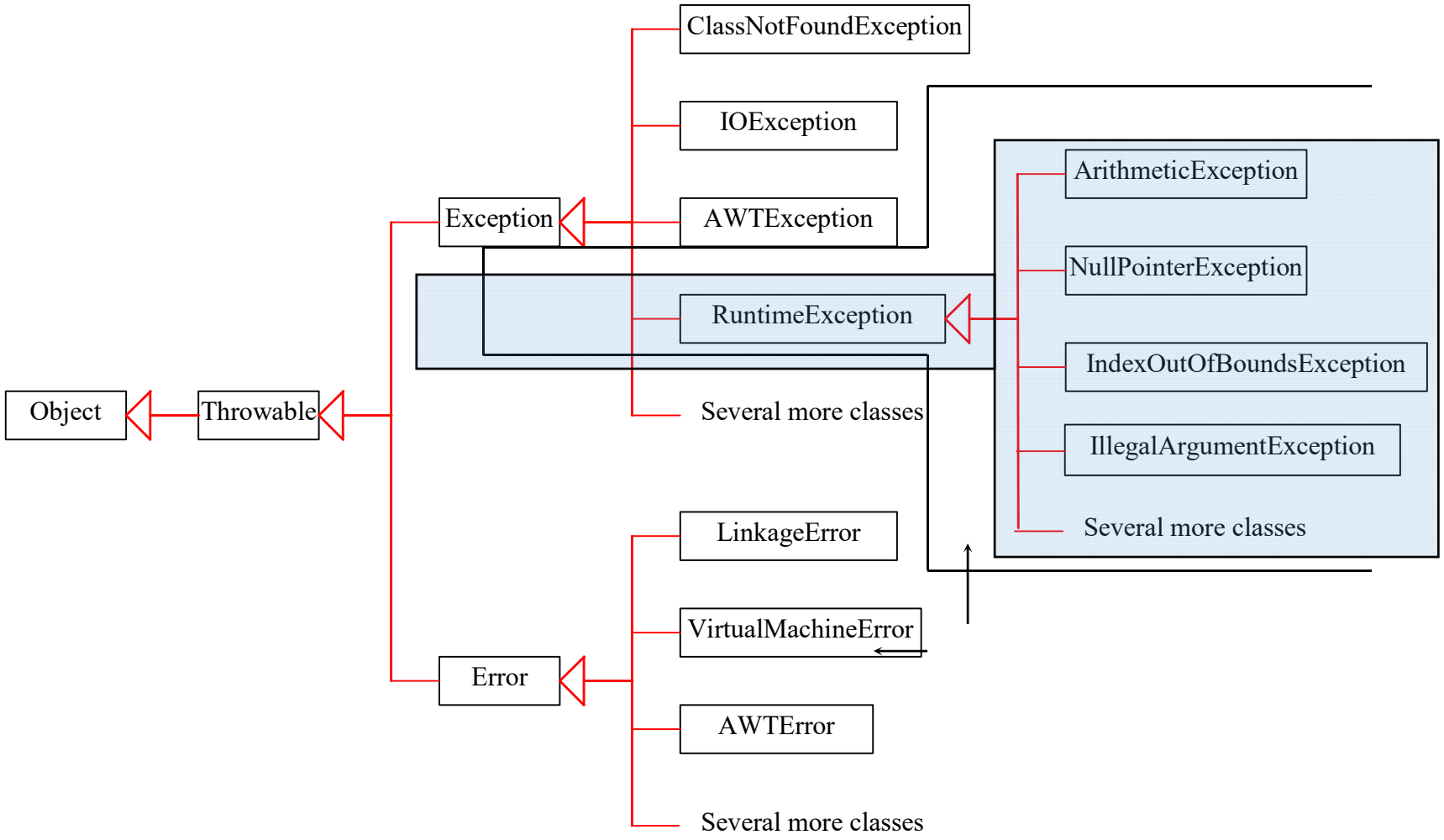
  "*Something bad happened. I failed*."

- Exceptions are objects of Exception or Error class or their subclasses.

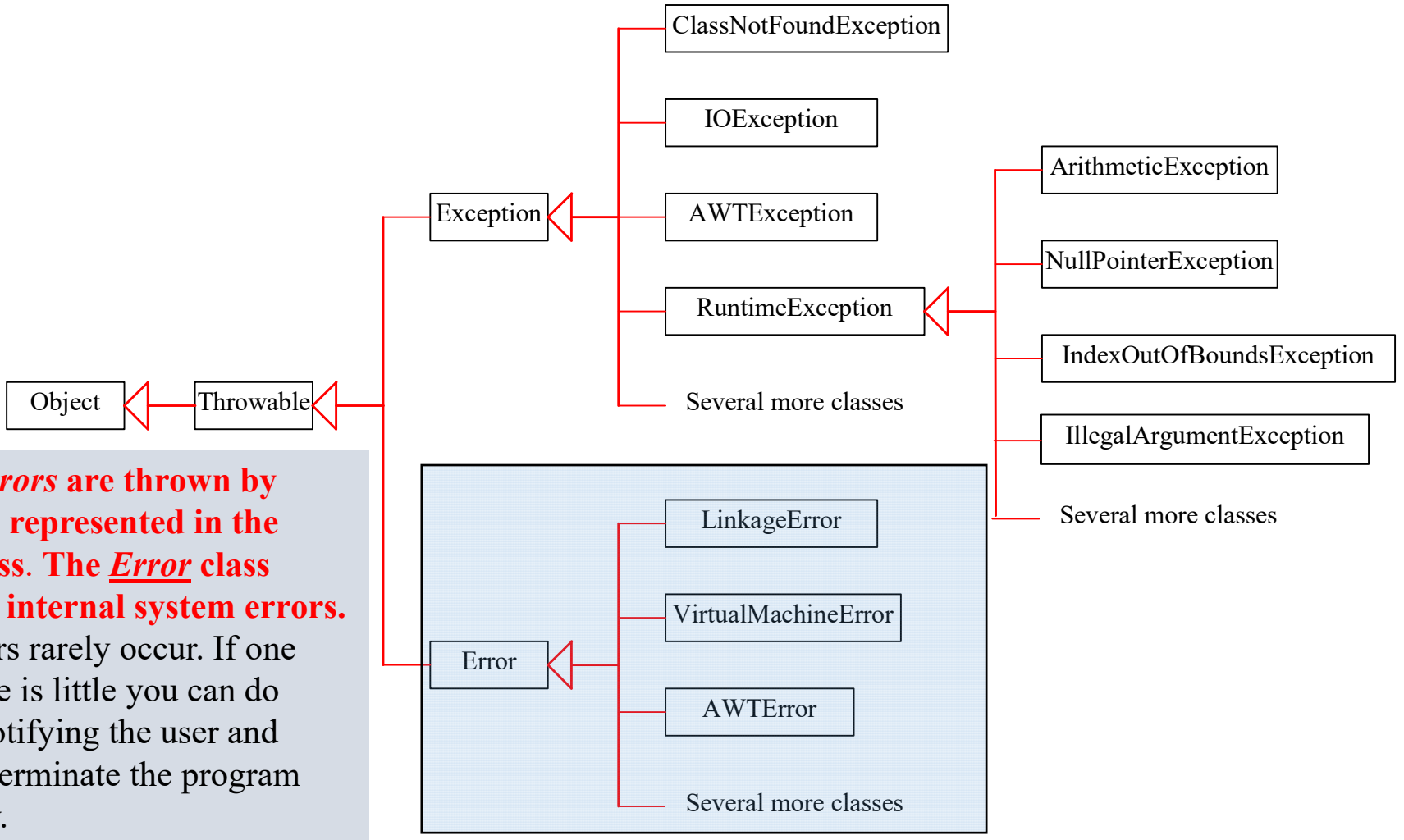
# Exception Classes



# Unchecked Exceptions

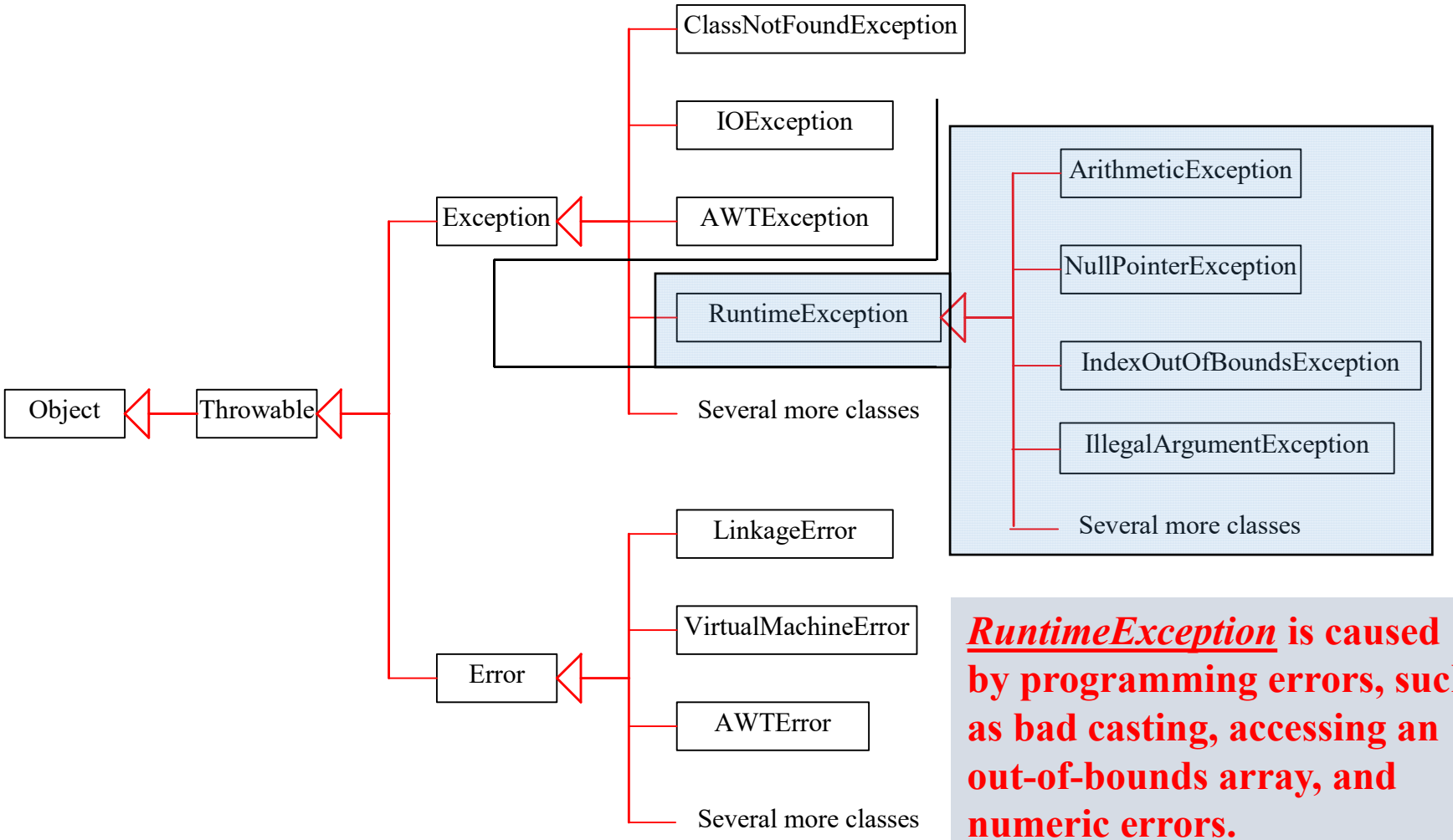


# System Errors



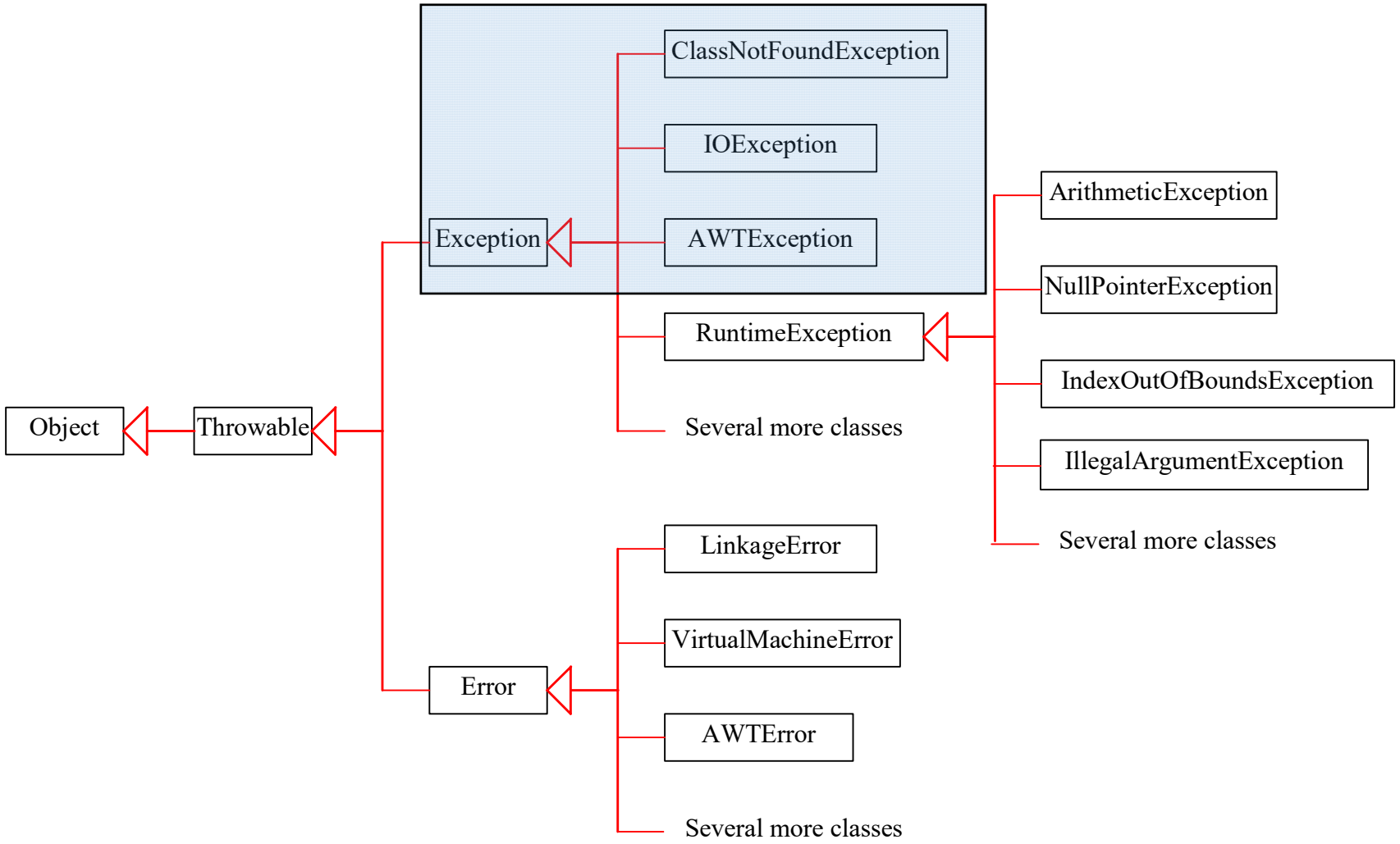

***System errors* are thrown by JVM and represented in the Error class. The *Error* class describes internal system errors.** Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

# Runtime Exceptions




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

# Checked Exceptions



# Exception Handling

- Keywords:

# Java Library Exceptions

- Most Java routines throw exceptions.

- How do you know that the method you are going to call may throw an exception?
  - You can look up the class documentation to see if a method throws exception

- Example:

  See the Scanner class methods at:

  http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html

# Handling Exceptions

- Java forces you to deal with checked exceptions.

- Two possible ways to deal:

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

(a)

```
void p1() throws IOException {

  riskyMethod();

}
```

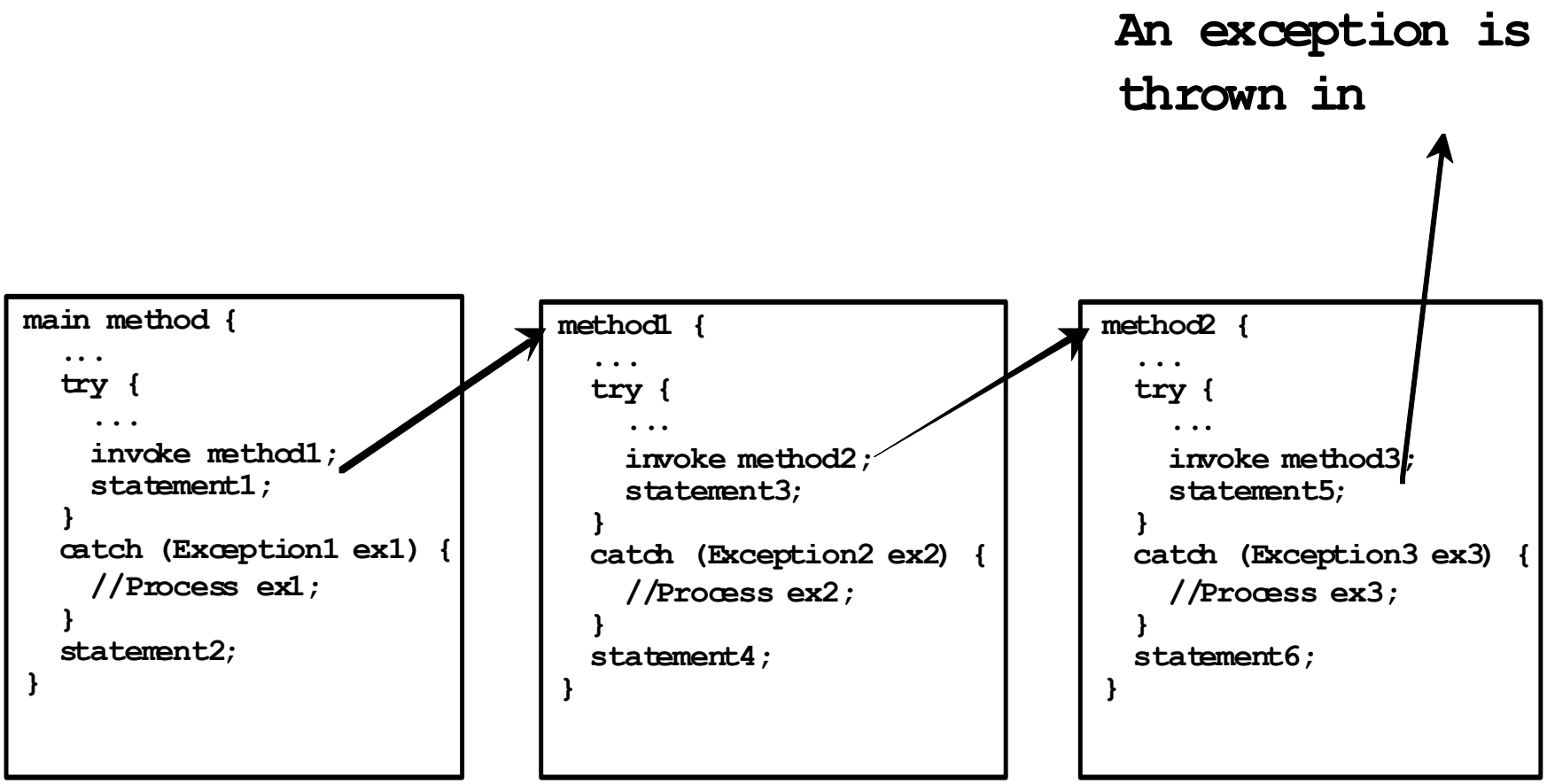
(b)

# Catching Exceptions

- Install an exception handler with **try/ catch** statement

```
try {
      //Statements that may throw exceptions
}

catch (Exception1 exVar1) {
  //code to handle exceptions of type Exception1;
}

catch (Exception2 exVar2) {
  // code to handle exceptions of type Exception2;
}

...
catch (ExceptionN exVarN) {
  // code to handle exceptions of type exceptionN;
}

// statement after try-catch block
```

# Catching Exceptions



# Getting Information from Exceptions

- Use instance methods of the java.lang.Throwable class
- Some useful methods:

| | |
|---|---|
| String **toString**() | Returns a short description of the exception |
| String **getMessage**() | Returns the detail description of the exception |
| void **printStackTrace**() | Prints the stacktrace information on the console |

- Example of printStackTrace() output

# Example

```
public class Main {
    public static void main(String[] args) {
        java.io.PrintWriter output = null;
        try {
                output = new java.io.PrintWriter("text.txt");
                output.println("Welcome to Java");
                output.close();
        }
        catch(java.io.IOException ex){
                System.out.println(ex.toString());
                ex.printStackTrace() ;
        }
    }
}
```

# Issues

```
public class Main {
   public static void main(String[] args) {
       java.io.PrintWriter output = null;
       try {
              output = new java.io.PrintWriter("text.txt");
              output.println("Welcome to Java");
              output.close();
       }
       catch(java.io.IOException ex){
               ex.printStackTrace() ;
       }
    }
}
```

- Must execute output.close() even if exception happens

# Solution

- Use *finally* clause for code that must be executed "no matter what"

```
try {
      //Statements that may throw exceptions
}

catch (Exception1 exVar1) {
  //code to handle exceptions of type Exception1;
}

catch (Exception2 exVar2) {
  // code to handle exceptions of type Exception2;
}
...
catch (ExceptionN exVar3) {
  // code to handle exceptions of type exceptionN;
}

finally { // optional
   // code executed whether there is an exception or not
}
```

# Use finally block

```
public class Main {
   public static void main(String[] args) {
       java.io.PrintWriter output = null;
       try {
              output = new java.io. PrintWriter("text.txt");
              output.println("Welcome to Java");
       }
       catch(java.io.IOExcetion ex){
               ex.printStackTrace() ;
       }
       finally {
              if (output != null) output.close();
       }
    }
}
```

# *finally* block

- Executed when try block is exited in any of three ways:
  - *After last statement of try block  (success).*
  - *After last statement of catch clause, if this catch block caught an exception.*
  - *When an exception was thrown in try block and not caught*
- *Executed even if there is a return statement prior to reaching the finally block*

# Throwing Exceptions

When somebody writes a code that could encounter a runtime error,

- it creates an object of appropriate Exception class and throws it
- and must also declare it in case of checked exception

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

```
public class Circle {
        private double radius;
        private static int numberOfObjects = 0;

        public Circle() {   this(1.0);  }

        public Circle(double newRadius) throws IllegalArgumentException
        {
            setRadius(newRadius);    numberOfObjects++;
        }

        public double getRadius() {    return radius;   }

        public void setRadius(double newRadius)
                        throws IllegalArgumentException {

                if (newRadius >= 0)
                        radius =  newRadius;
                else
                        throw new IllegalArgumentException(
                                "Radius cannot be negative");
        }

        public static int getNumberOfObjects() {
                return numberOfObjects;
        }
  }
```

```
public class TestCircle {
        public static void main(String[] args) {
                try {
                        Circle c1 = new Circle(5);
                        Circle c2 = new Circle(-5);
                        Circle c3 = new Circle(0);
                }
                catch (IllegalArgumentException ex) {
                        System.out.println(ex);
                }
                System.out.println("Number of objects created: "
                                        + Circle.getNumberOfObjects());
        }
}
```

**Output:**

java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1

# Creating Custom Exception Classes

- Create custom exception classes if the predefined classes are not sufficient.

- To declare custom exception class:
  - Create a class that *extends Exception* or a subclass of Exception.
  - It is good practice to add:
    - An argument-less constructor
    - Another constructor with one string type parameter

```
public class InvalidRadiusException extends Exception {
        private double radius;
        public InvalidRadiusException() { super("invalid radius!"); }
        public InvalidRadiusException(double radius) {
            super("Invalid radius "); this.radius = radius;
        }
        public double getRadius() { return radius; }
}
```

```
public class Circle {
        private double radius;
        private static int numberOfObjects = 0;

        public Circle() {    this(1.0);   }
        public Circle(double newRadius) throws InvalidRadiusException{
            setRadius(newRadius); numberOfObjects++;
        }

        public void setRadius(double newRadius)
                        throws InvalidRadiusException {
            if (newRadius >= 0)       radius =  newRadius;
            else throw new InvalidRadiusException(newRadius);
        }
        public static int getNumberOfObjects() {
                return numberOfObjects;
        }
}
```

```
public class Main {
  public static void main(String[] args) {
    try {
      Circle c1 = new Circle(5);   c1.setRadius(-5);
      Circle c2 = new Circle(0);
    }
    catch (InvalidRadiusException ex) {
        System.out.println("Invalid Radius: " + ex.getRadius());
    }

    System.out.println("Number of objects created: " +
                                Circle.getNumberOfObjects());
  }
}
```

**Output:**

**Invalid radius: -5.0**
**Number of objects created: 1**

# When to create Custom Exception classes

- Use the exception classes in the API whenever possible.
- You should write your own exception classes if you answer ‘yes’ to one of the following:

  ✓**Do you need an exception type that isn't represented by those in the Java platform?**

  ✓**Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?**

  ✓**Do you want to pass more than just a string to the exception handler?**

# When to Use Exceptions

- Use it if the event is truly exceptional and is an error
- Do not use it to deal with simple, expected situations.
- Example:

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

Can be replaced by:

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

# Get more info!

- Java docs: Exception
- http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Exception.html

- Sun Tutorial on Exception Handling

  http://java.sun.com/docs/books/tutorial/essential/exceptions/definition.html

- Exception Handling @mindprod.com
- http://mindprod.com/jgloss/exception.html