CHAPTER 12
EXCEPTION HANDLING AND TEXT IO
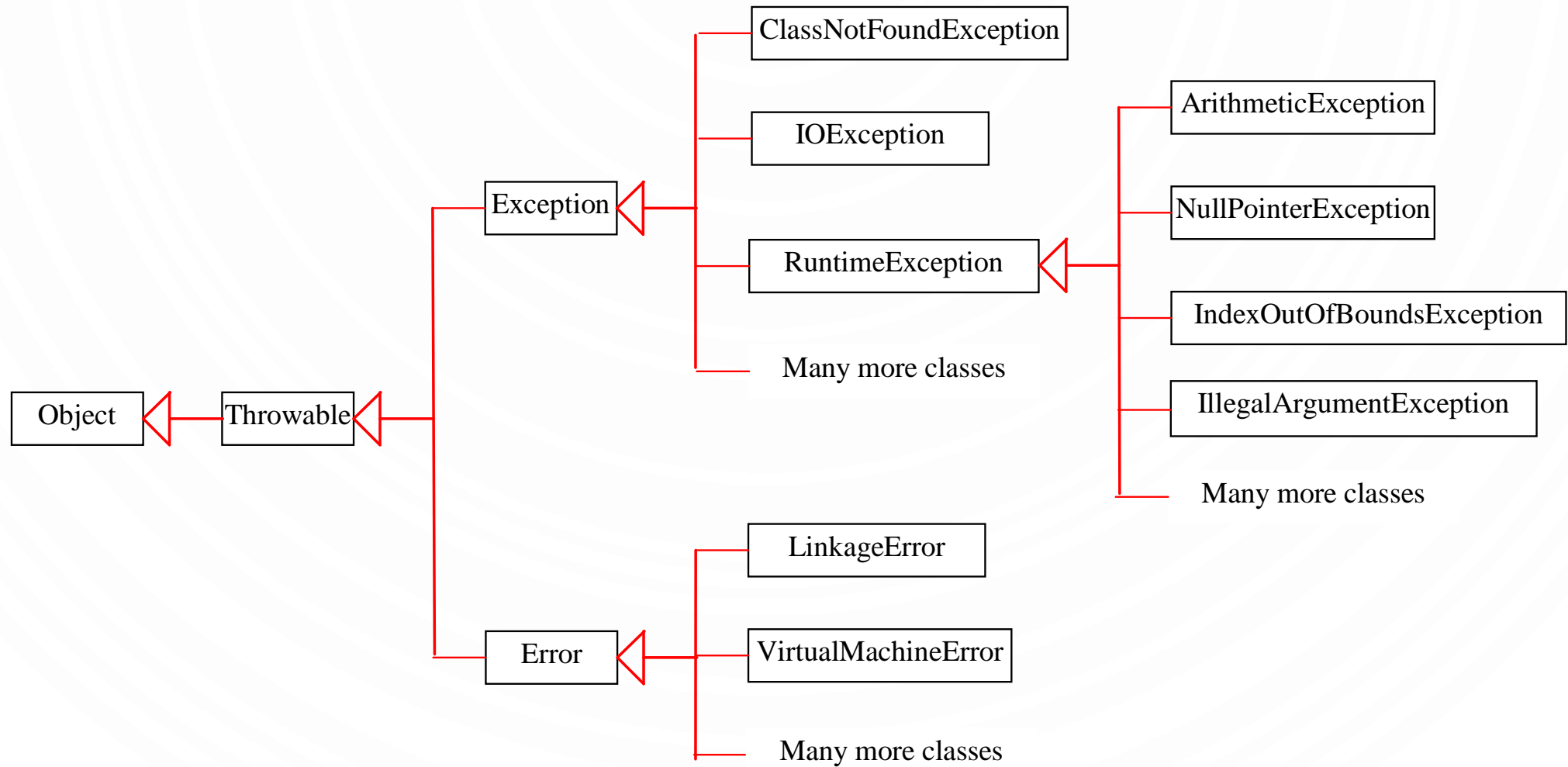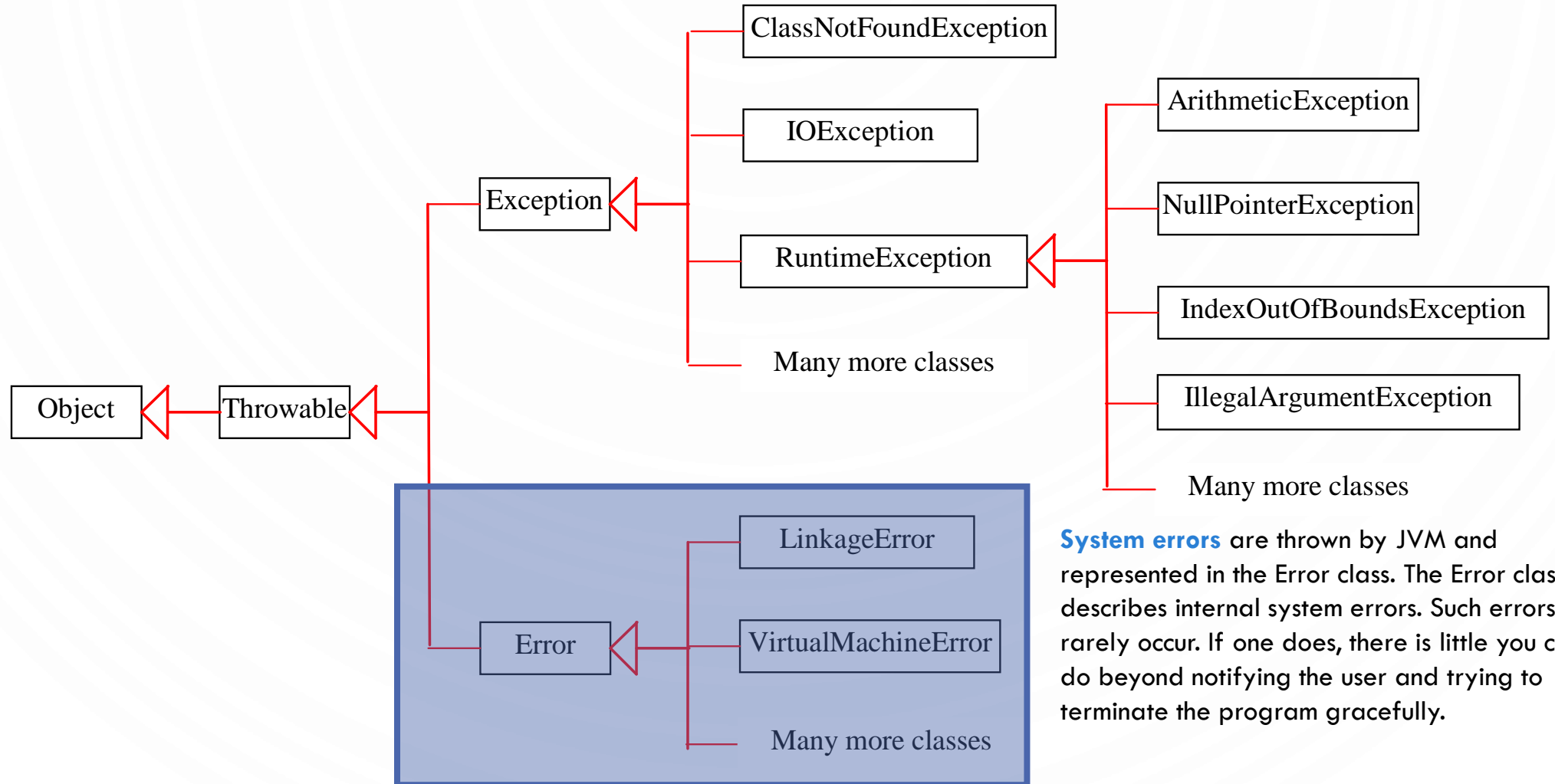
MOTIVATIONS

- 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?

- Example
  - You are working on Microsoft Word, and you try to open a file that does not exist OR is an incorrectly formatted .doc or .docx file (like someone tampered with it). What should happen?
    - (a) Microsoft crashes
    - (b) Microsoft alerts you of the issue
    - (c) Forget Microsoft, Apple is superior!

EXCEPTION TYPES

SYSTEM ERRORS

```
Object ◁— Throwable ◁—┬— Exception ◁—┬— ClassNotFoundException
                      │                │
                      │                ├— IOException
                      │                │
                      │                ├— RuntimeException ◁—┬— ArithmeticException
                      │                │                      │
                      │                └— Many more classes   ├— NullPointerException
                      │                                        │
                      └— Error ◁—┬— LinkageError               ├— IndexOutOfBoundsException
                                 │                              │
                                 ├— VirtualMachineError         ├— IllegalArgumentException
                                 │                              │
                                 └— Many more classes           └— Many more classes
```
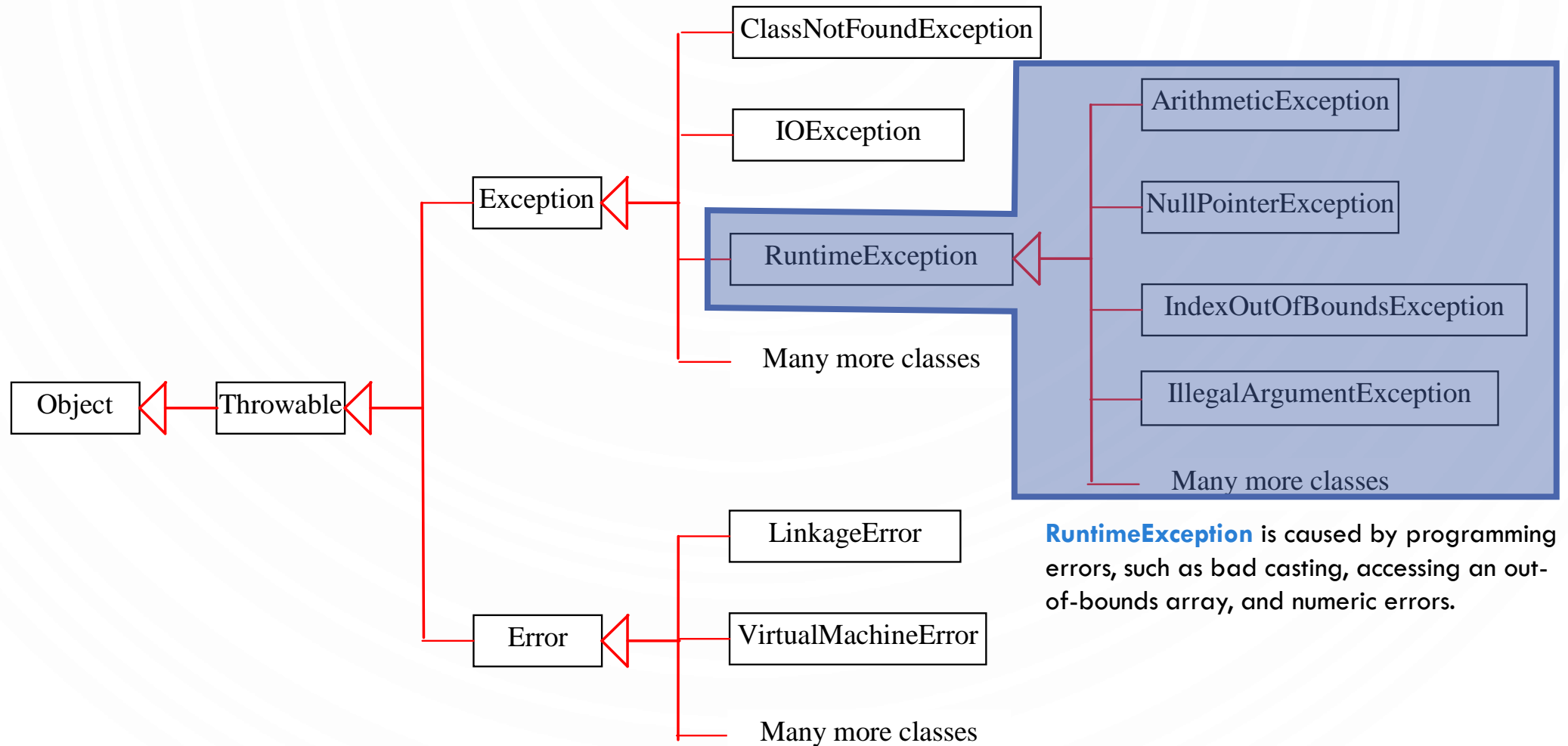
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

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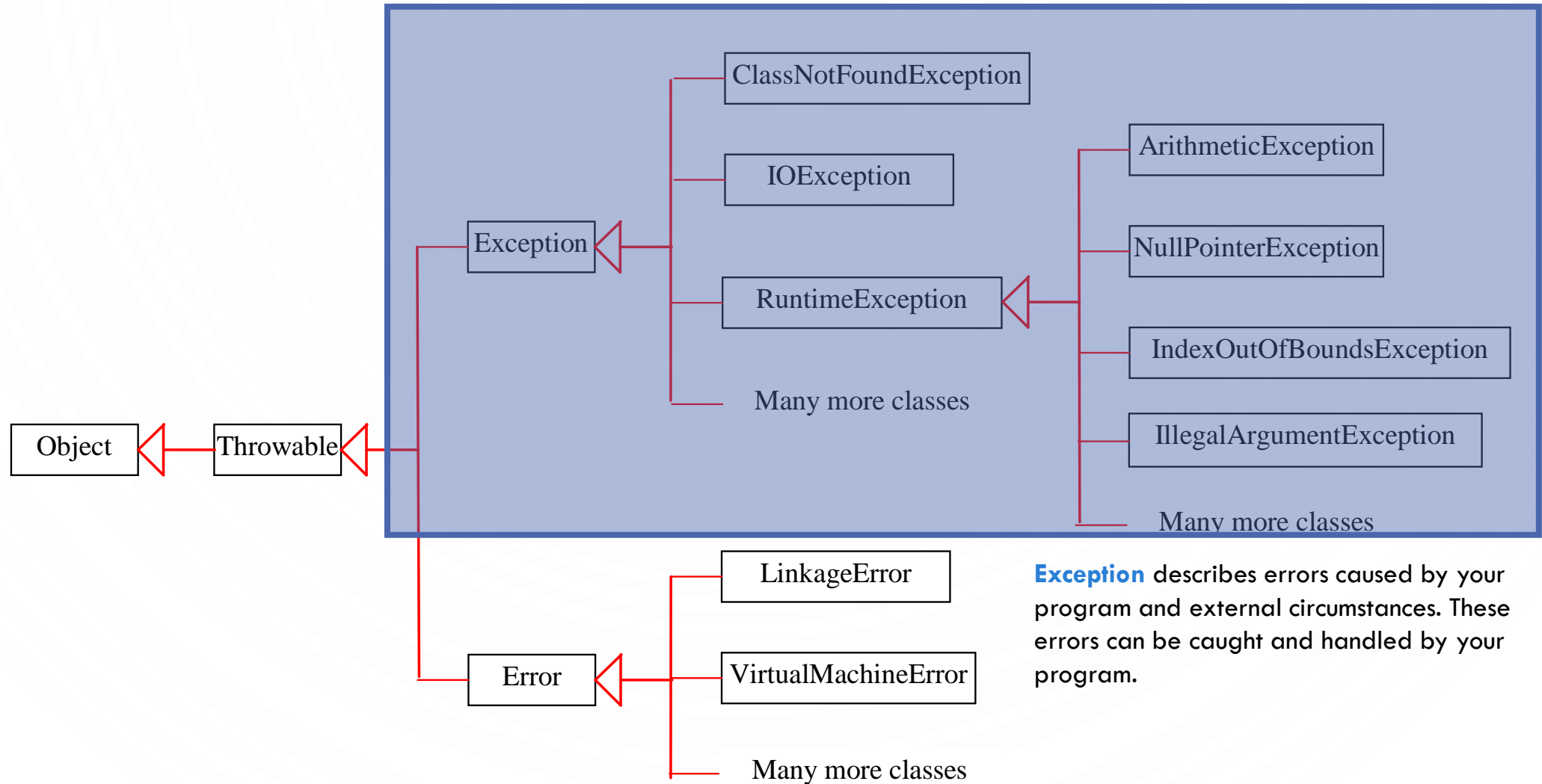
Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.
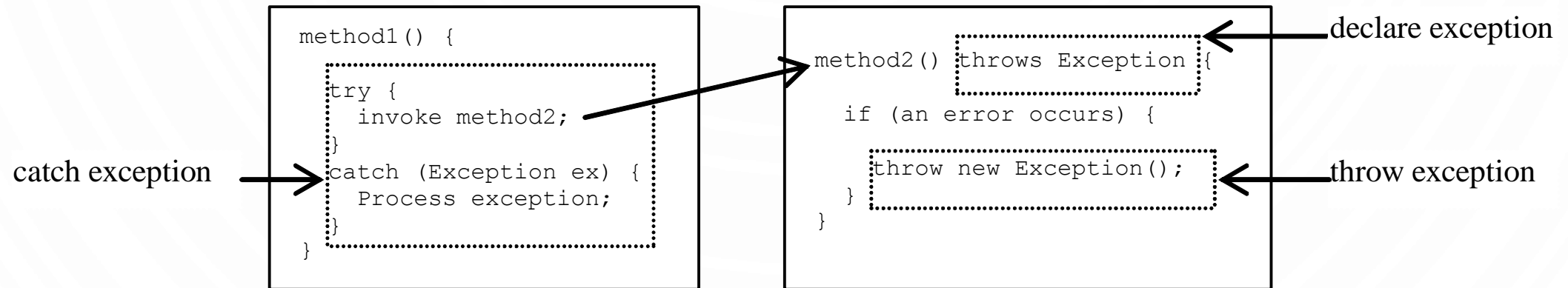
EXCEPTIONS

ClassNotFoundException

IOException

Exception

RuntimeException

Many more classes

Object — Throwable

ArithmeticException

NullPointerException

IndexOutOfBoundsException

IllegalArgumentException

Many more classes

LinkageError

Error

VirtualMachineError

Many more classes

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.

- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. 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.

DECLARING, THROWING, AND CATCHING EXCEPTIONS

```
method1() {

  try {
    invoke method2;
  }

  catch (Exception ex) {
    Process exception;
  }

}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

declare exception

throw exception

catch exception

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

THROWING EXCEPTIONS

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as throwing an exception. Here is an example,

- `throw new Exception();`

- `Exception ex = new Exception();`
  `throw ex;`

THROWING EXCEPTIONS EXAMPLE

```java
1.    /** Set a new radius */
2.   public void setRadius(double newRadius)
3.        throws IllegalArgumentException {
4.     if (newRadius >= 0)
5.       radius =  newRadius;
6.     else
7.        throw new IllegalArgumentException(
8.         "Radius cannot be negative");
9.   }
```

CATCHING EXCEPTIONS

```
1.  try {
2.    statements;   // Statements that may throw exceptions
3.  }
4.  catch (Exception1 exVar1) {
5.    handler for exception1;
6.  }
7.  catch (Exception2 exVar2) {
8.    handler for exception2;
9.  }
10. ...
11. catch (ExceptionN exVar3) {
12.   handler for exceptionN;
13. }
```

CATCH OR DECLARE CHECKED EXCEPTIONS

- Suppose p2 is defined as follows:

```
void p2() throws IOException {
   if (a file does not exist) {
      throw new IOException("File does not exist");
   }

   ...
}
```

CATCH OR DECLARE CHECKED EXCEPTIONS

- Java forces you to deal with checked exceptions. If a method declares a checked exception, 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();

}
```
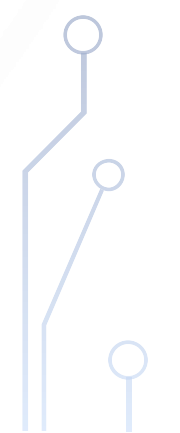(b)

THE FINALLY CLAUSE

```
1. try {
2.     statements;
3. }
4. catch (Exception ex) {
5.     handling ex;
6. }
7. finally {
8.     finalStatements;
9. }
```

CAUTIONS WHEN USING EXCEPTIONS

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

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.

- Often you can handle exception with if-else statements like we have previously seen. In this case, there is no need to throw.

INPUT AND OUTPUT

- Input devices

Keyboard   Mouse   Hard drive   Network   Digital camera   Microphone

- Output devices.

Display   Speakers   Hard drive   Network   Printer   MP3 Player

- Goal. Java programs that interact with the outside world.
  - Java Libraries support these interactions
  - We use the Operating System (OS) to connect our program to them

WHAT HAVE WE SEEN SO FAR?

- Standard output.
  - The OS output stream for text
  - By default, standard output is sent to Terminal.
  - Example: System.out.println() goes to standard output.

- Standard input.
  - The OS input stream for text
  - By default, standard input is received from the Terminal.
  - Example: Scanner

- "Standard Draw."
  - Really a wrapper for Java's GUI libraries
  - Output to a window instead of a terminal
  - Example: Draw a circle on the screen

FILE INPUT AND OUTPUT

FILE INPUT

- We can reuse **Scanner**!

- Instead of "scanning" **System**.in, we scan a **File**.

- However we must:
    - Import **Scanner**, **File**, and **FileNotFoundException**
    - Modify our main function to handle a **FileNotFoundException**

```
1.  Scanner in = new Scanner(
        new File("myfile.txt"));
2.  in.nextInt();
3.  in.nextDouble();
4.  in.hasNext();
```

```
1.  import java.io.File;
2.  import java.io.FileNotFoundException;
3.  import java.util.Scanner;
4.
5.  public class MyProgram {
6.      public static void main(String[] args)
                throws FileNotFoundException {
7.          //Do something!
8.      }
9.  }
```

- Note – alternative to **throws**, we could have done a **try-catch** block

FILE OUTPUT

- We can use **PrintWriter**

- Offers `print`, `println`, `printf` just like **System**.`out`

1. **PrintWriter** `out` = **new**
   PrintWriter("MyFile.txt");
2. `out.println("Hello FileIO
   World!");`

- Similarly we need to:
  - Import **PrintWriter** and **FileNotFoundException**
    1. **import** `java.io.`**PrintWriter**;
  - Modify main to handle/throws **FileNotFoundException**

FILE INPUT/OUTPUT CAVEATS

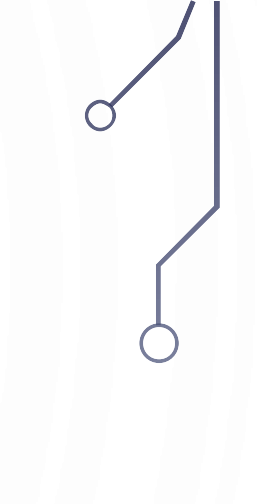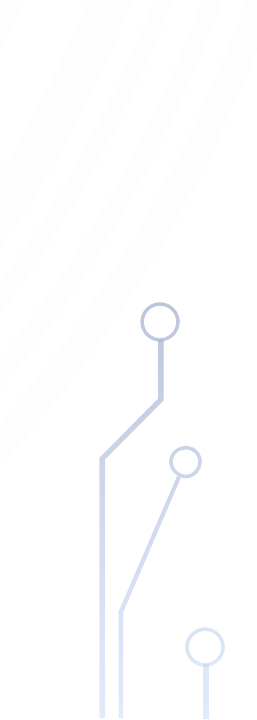- Always call close after you are done using **Scanner** or **PrintWriter**

1. `Scanner` `in =`
   `new Scanner(new`
   `File("MyFile.txt"));`
2. `//Use the Scanner as much`
   `//as you want`
3. `in.close();`

- Call `flush` often on **PrintWriter** to ensure all output gets into the file.

1. `PrintWriter out = new`
   `PrintWriter("MyFile.txt");`
2. `//Use the PrintWriter as much`
   `//as you want`
3. `out.flush(); //Always flush`
   `after use!`
4. `out.close();`

FOR MORE INFORMATION

- Google

- API

- Tutorials

- StackOverflow

- Practice, Practice, Practice!

EXERCISE – WORK IN TRIPLETS

EXERCISE

1. Write a program that will generate N random cirlces, where $N \in [3, 20]$, the center (x, y) points between $[-10, 10)$, and the radius is between $[1, 4)$. Write the circles to a file – first line is N, each line after is the circle defined by x, y, and r

2. Write a program that reads your file and shows it to the user using StdDraw. Use a random color to show the outline and a different random color to fill the polygon.

3. Augment your programs to

   1. Allow random rectangles as well. Randomly select circles/rectangles with 0.5 probability