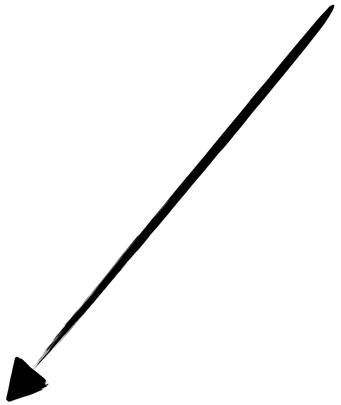
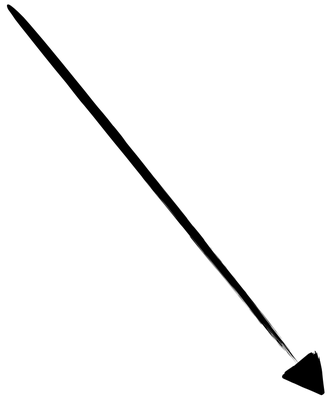
Throwable





Exception

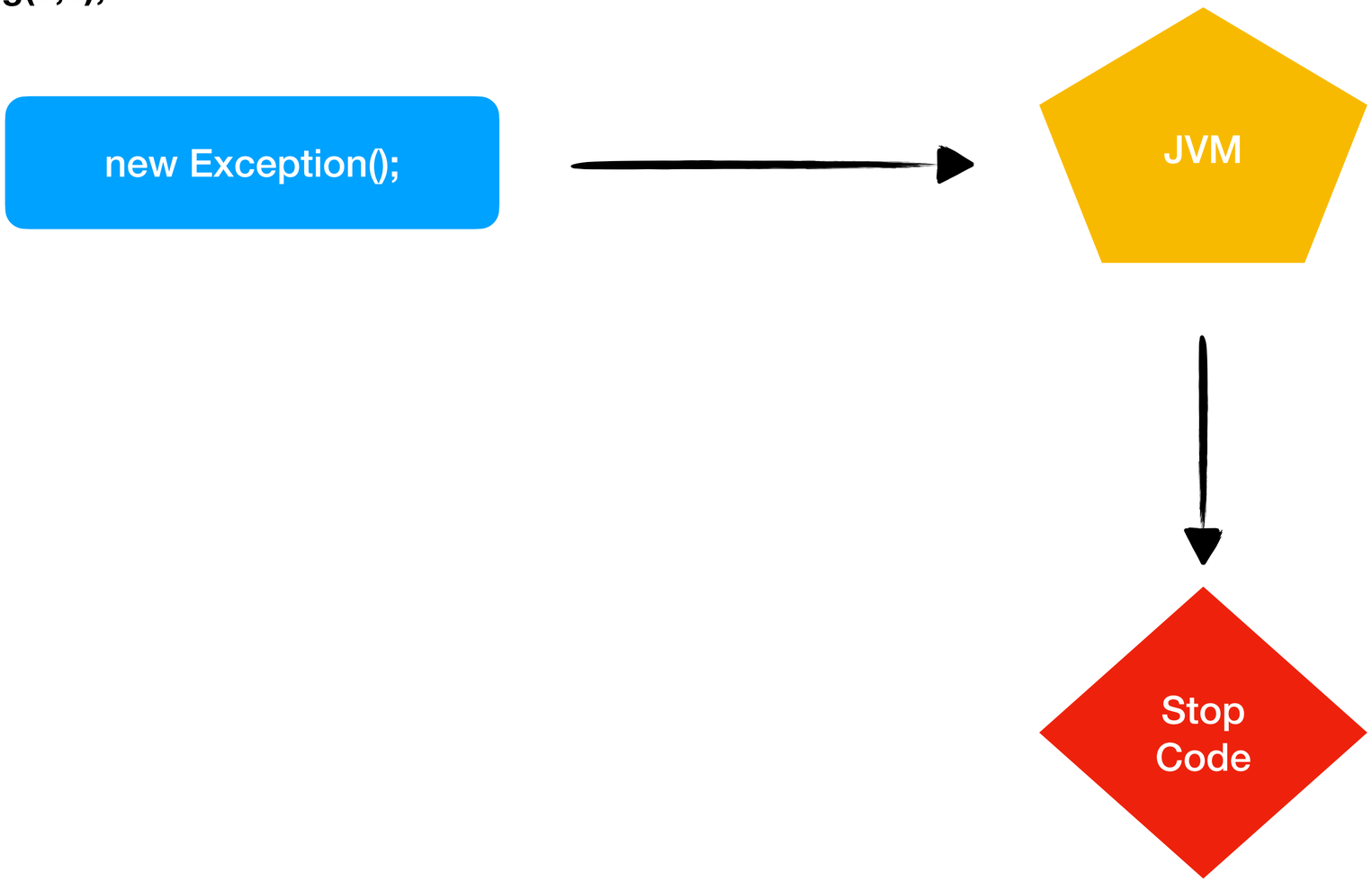
String name = null;
name.substring(2,5);

Error

Error indicates serious problems in memory.

Throwing an Exception

```
String name = null;
name.substring(2,5);
```



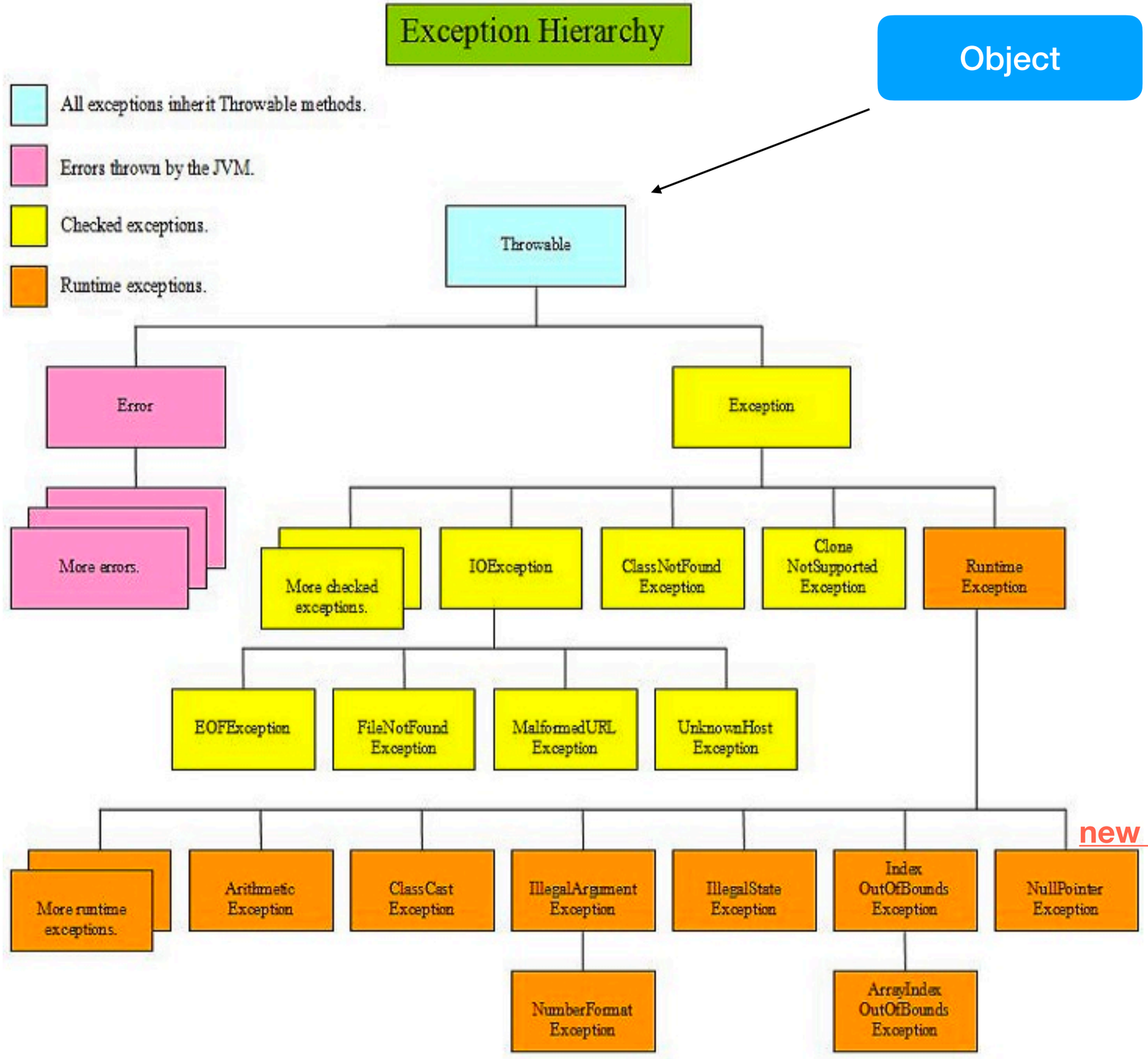
Exceptions

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

- Once exception occurs java creates exception Object and handles it to JVM, and this process is named ***throwing an exception***.

```
Console
<terminated> TestException [Java Application] /Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java (Apr 24, 2020, 3:48:57 PM)
Exception in thread "main" java.lang.NullPointerException
        at ExceptionsErrors.TestException.main(TestException.java:9)
```

new Exception();



Object

Exception

Checked

Runtime / Unchecked

Critical Exception
FileReader fReader =
new FileReader("Exceptions/Test.txt");

normal
Int a = 10;
Int b = 0;
Int c = a/b;

Types of Exceptions

Unchecked Exceptions (Runtime Exception)

- ***Exception occurs at runtime.***
- ***It is Optional to handle unchecked exceptions***

Checked Exceptions

- ***Exception occurs at compile time***
- ***Java requires to handle checked exceptions***

Try Catch Finally block

- ***If in try block Exception is thrown it will not continue to execute next line in try block, it will go to catch statement and check if type the of exception is matched, if it is matched then it will execute that catch block.***

- ***Finally block will be executed in any case, either they will be exception or not.***

- ***Finally block mostly used to close statements like Database connection or FileReader and etc…***

- ***We can skip execution of finally block with System.exit(0);***

- ***'Return' statement will execute finally block but will not continue after finally block.***

```
try {
    int result = 5/0;
    System.out.println("After exception");
}catch (ArithmeticException e) {
    System.out.println(e);
    return;
}finally {
    System.out.println(e);
}
```

Declaring in method signature

- ***We declare it with 'throws' keyword***
- ***It will handle exception, but if there will exception it will throw that Exception.***
- ***We can declare multiple Exceptions types in method signature.***

```
public static void main(String[] args) throws InterruptedException,
ArithmeticException, MalformedURLException {

        System.out.println("Sleep for 5 seconds");
        Thread.sleep(5000);
        System.out.println("Wake up...");

}
```

How to handle Exceptions

Try catch block

- ***Try catch block will allow to keep executing our code after Exception is handled.***

Declaring in method signature

- ***It is mostly applied for Checked Exceptions in order to handle them.***
- ***If they will be an Exception it will throw it and will terminate the code even if we declared it in method signature.***

Error

- ***Error*** means something went horribly wrong that you shouldn't try to recover from it.

- **StackOverFlowError** -> Thrown by JVM when there is no memory left in stack.
- **NoClassDefFoundError** -> Thrown by JVM when a class that the code uses is available at compile time but not runtime.

- Error can be handled but not recommended by Java.

Stack

Reference
primitive data types
methods

Heap

Object
String Pool

Throw vs Throws

- Throw is used when we want to throw an Exception explicitly.

- throw new Exception();

- Throws is used when we are handling the exception in method signature

- void method() throws Exception{}

Final, Finalize, Finally

- ***Final*** -> is used to declare the class if the class won't be inherited. **Final method** is declared when method is not meant to be overridden. **Final Variables** is constant variable which meant to be declared to give only once.

- ***Finalize()*** -> is a method which is called by garbage collector to clean unused objects. (System.gc();, System.finalize();)

- ***Finally*** -> is used in try catch block, it will be executed always at the end of try catch block. It is mostly used to close database or file connections.

Lambda Expressions

- Functional Interfaces used for lambda expression.
- Why lambda? // You can do everything without using lambda
  - Enable Functional programming
  - Easy to read
  - Better code