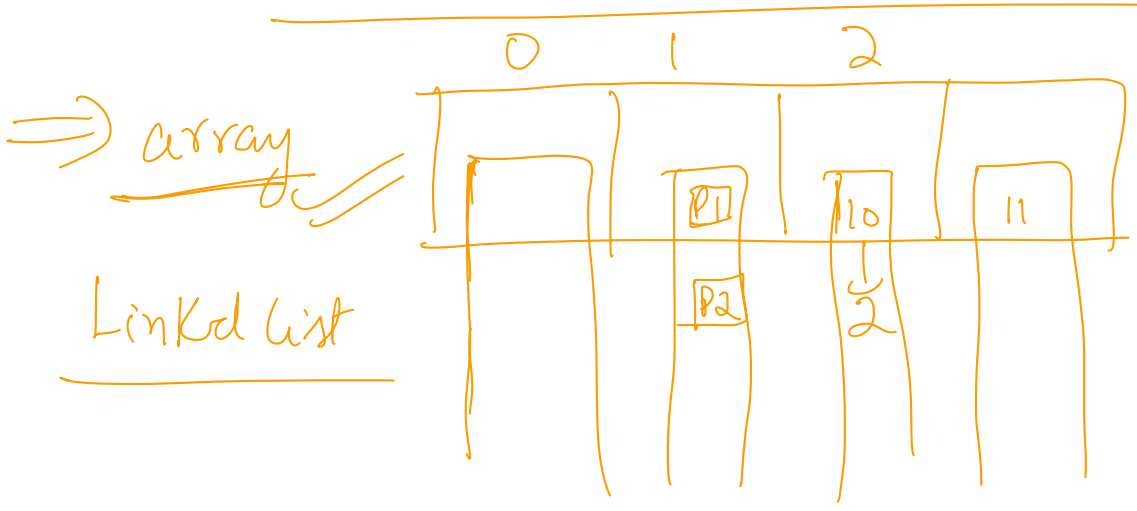
Agenda: (1) Collections Wrap-up
- (1) LinkedHashSet
- (2) PriorityQueue
- (3) TreeSet

(2) Exception Handling
- (1) What are exceptions?
- (2) try catch finally
- (3) Nested try-catch
- (4) Checked vs unchecked exceptions

Class starts at 9:05



set.add(2)

2 % 4 = 2

set.add(10) ⟹ HC(10) ⟹ 10

modulus of |10| ⟹ 10 ⟹ hash function

|absValue of HC| % no. of buckets

$|10| = 10 \Rightarrow 10\%4 = 2$

```
Point {
    int x;
    int y;
}
```

```
HashSet<Point> set;
set = new HashSet<>();
```

1. `Point P1 = new Point(10,20);`
2. `set.add(P1);`
3. `Point P2 = new Point(10,20);`
4. `set.add(P2);`

(10,10)

P1 == P2

address ✗ address

```
Point {
    int x;
    int y;
```

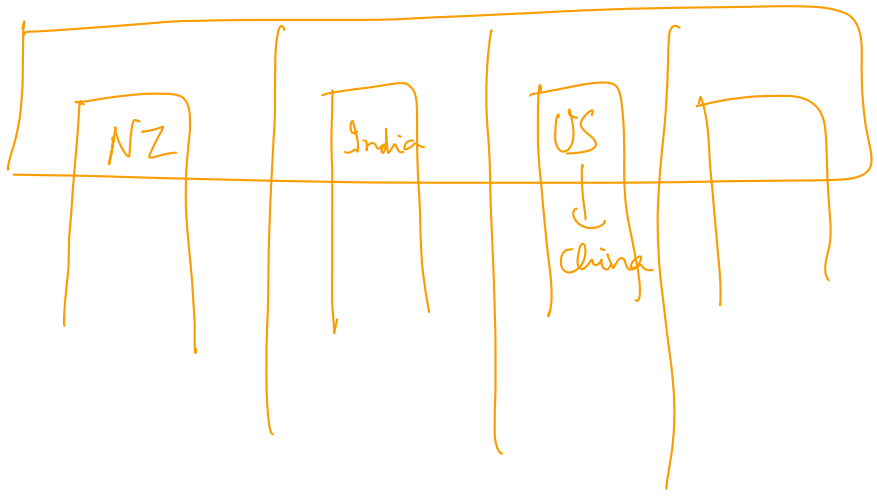
.add(10)
.add(20)

```
boolean equals(Point o) {
    return this.x == o.x
```

(Id this.y == o.y));

}

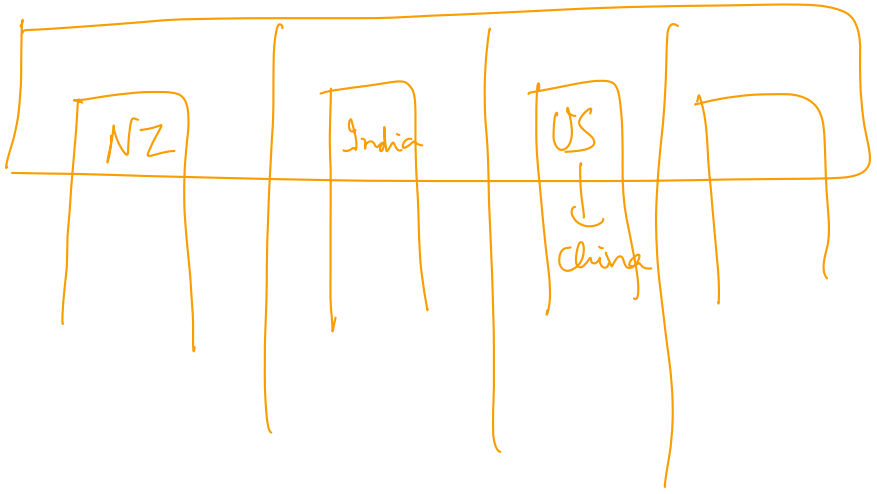
(2) LinkedHashSet ⇒



(1) "India"
(2) US
(3) China
(4) NZ

Ques Do you think, I will get the values from hash set in which I have added them.

LHS ⇒ ensures that we get the elements in the same order as we added them.



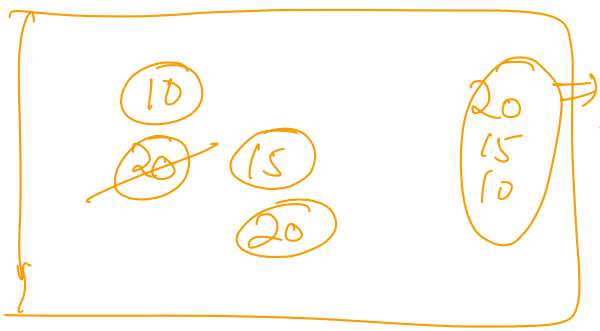
(1) India
(2) US
(3) China
(4) NZ

array

DLL

(3) Priority Queue ⇒ there are certain where you want to maintain the natural ordering of the elements.

```
PQ pq = new PQ();
```




```
pq.add(10);
pq.add(20);
pq.add(15);
```

↳ It internally trees to maintain this.

(4) TreeSet

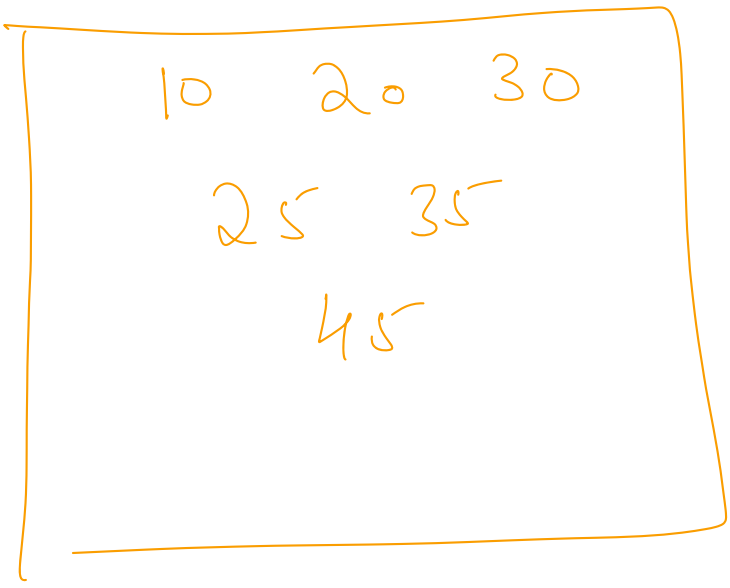
↳ Also implements the Set Interface
↳ add, remove, contains

⇒ O(log N) O(log N) O(log N)

TreeSet ⇒ based on trees only.
⇒ it also maintains the natural ordering of the elements.

⇒ Some additional functions of Treeset

```
ts.add(10);
ts.add(20);
ts.add(25);
ts.add(30);
ts.add(35);
ts.add(45);
```



ts.ceiling(35) ⇒ 45
ts.ceiling(25) ⇒ 30
ts.floor(35) ⇒ 30

ts.headset(35) ⇒ 10, 20, 25, 30
ts.tailSet(35) ⇒ 45
ts.subset(20, 35) ⇒ 25, 30
ts.subset(20, true, 35, true) ⇒ 20, 25, 30, 35

(2) Exception Handling

Ques What is an exception in Java program?

→ any type of unexpected situation that happens.

eg: 1

```
void readDataFromAFile (String fpath){
    File f = new File (fpath);
    f.read();
}
```

if filePath passed is incorrect, we will get a fileNotFoundException.

eg2: Scanner sc = new Scanner (System.in);

⇒ int a = sc.nextInt();

"abc"

It will throw an exception if we enter anything other than an integer.

⇒ InputMismatchException.

⇒ We need to handle all these exceptions in order to prevent any unexpected situation

Or a bad surprise for the client.

⟹ try, catch, finally.

```
try {
    L1
    L2
    L3   <- exception raised
    L4
} catch (FileNotFoundException fNf) {
    L5   // send an email to the developer that
    L6   // an exception has happened.
    L7
}
```

⟹ put your suspicious / vulnerable code inside the try block

⟹ write the code that you want to execute once an exception has been caught.

1. FileNotFoundException
2. IOException ⟹ application will crash with the exception if we dont handle it.

```
try {
    L1
    L2
    L3   <- raised exception
    L4
} catch (Exception e) {
    L5
```

⟹ opening a connection to a Database

L6

} finally {

⟹ close the database connection

⟹ write code that should be executed no matter what happens (whether exception is raised or not)

}

boolean fun() {

try {

L1
L2
X L3
✗ L6 } return false;

catch (___) {

L4
L5

} finally {

L6
L7

}

}

* finally would be executed even if we have a return statement in try block

Class resumes at 10:35 PM

⇒ Can we have multiple catch blocks for 1 try block.

eg:

```
① Scanner sc = new Scanner(System.in);
② int a = sc.nextInt();
③ int b = sc.nextInt();
④ int c = a / b;
```

```
try {
① Scanner sc = new Scanner(System.in);
② int a = sc.nextInt();          // 6
③ int b = sc.nextInt();          // 0
④ int c = a / b;
}
catch (InputMismatch ex) {
    L1
    L2
}
catch (DividebyZero) {
    X L3
    X L4
}
catch (Exception e) {
    Exception e = new ArrayIndexOutOfBounds();
    X L5
}
```

X L6

3

Case 1: Input Mismatch. => L1, L2

Case 2: DivideByZero => L3, L4

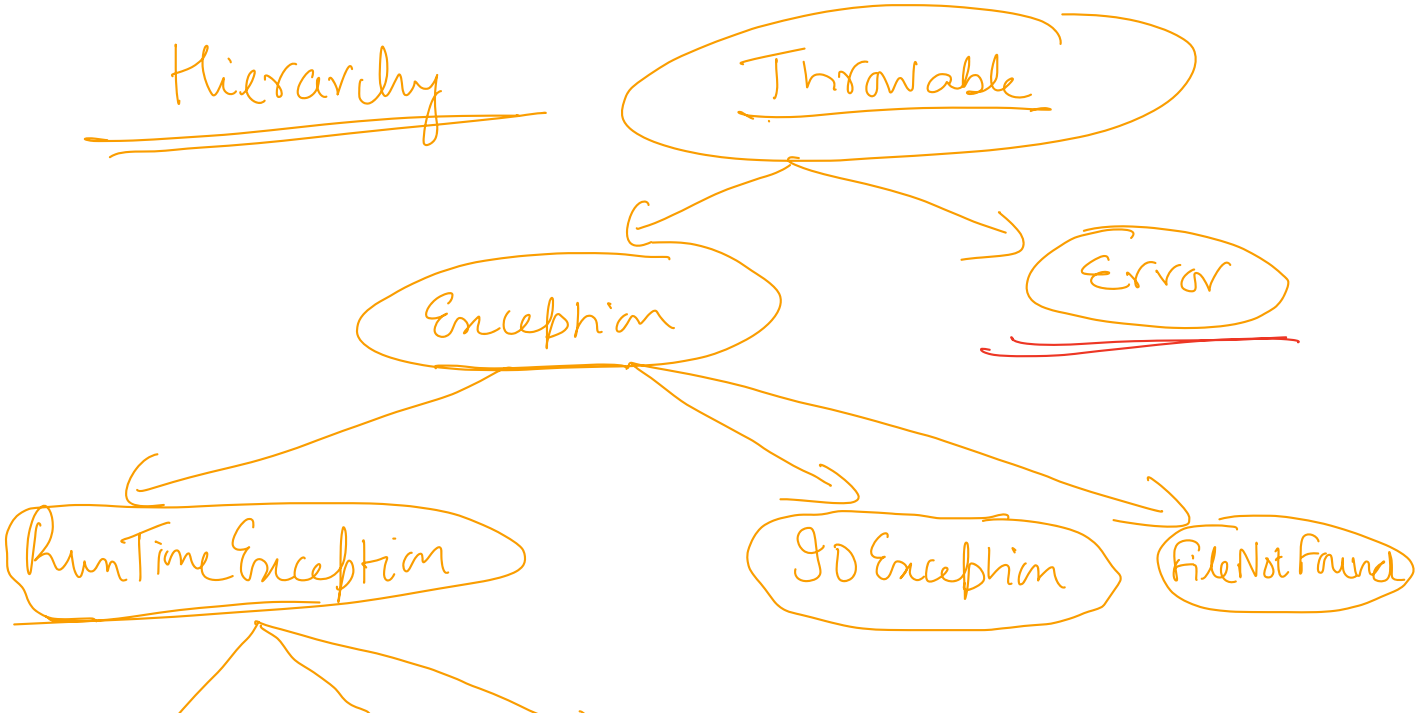
Case 3: ArrayIndexOutOfBounds => L5, L6

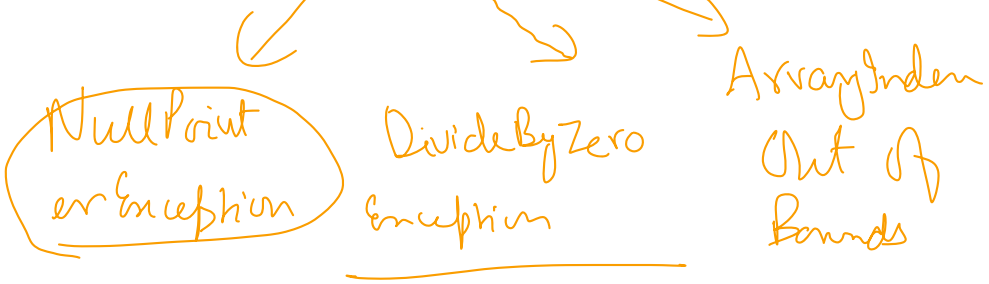
Case 4: ArrayIndexOutOfBounds => if last catch block is not there => Crash.

A
↑
B

```
A a = new A();
  = new B();
B b = new A(); X
```

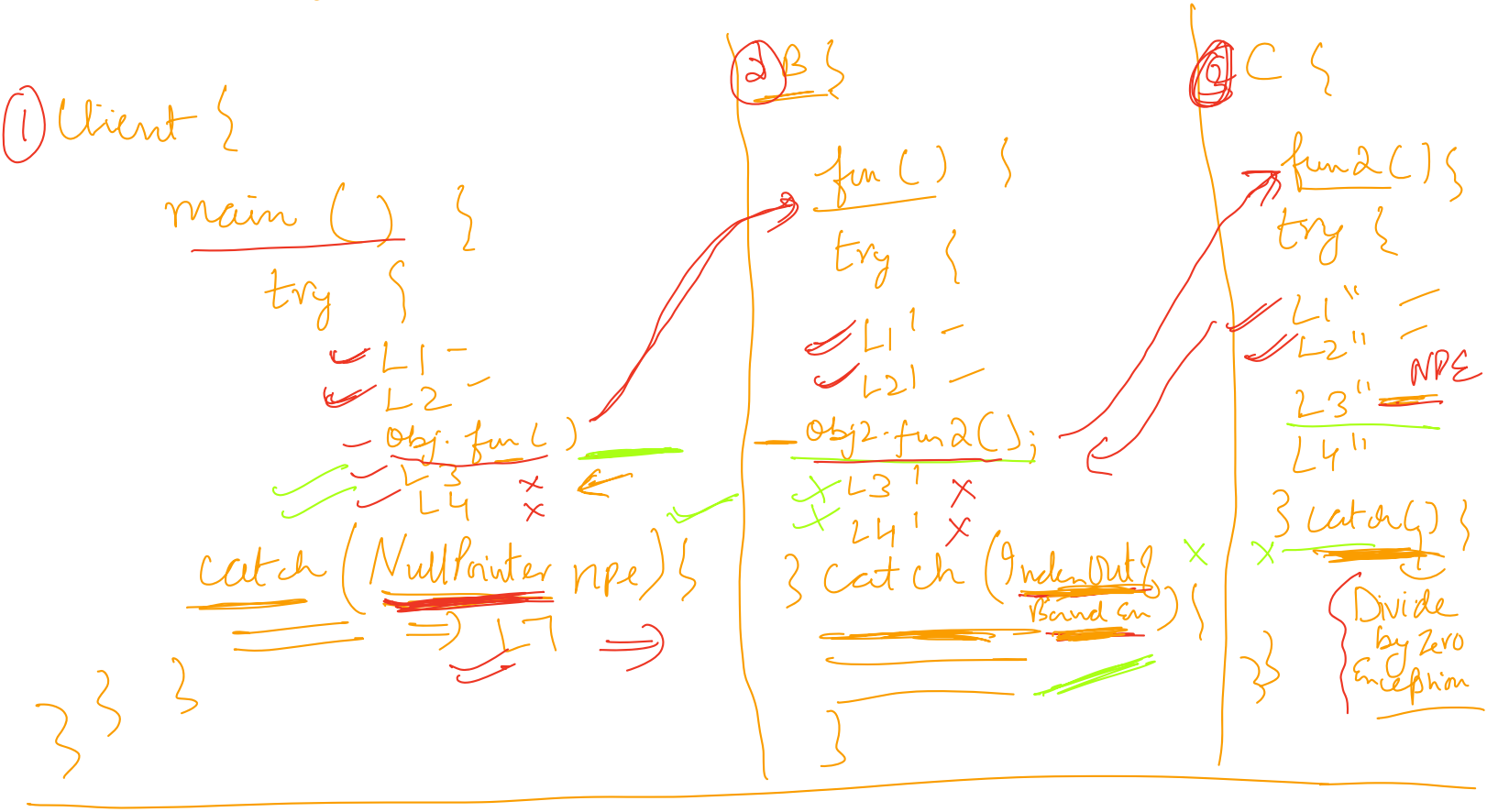
Hierarchy





Error ⇒ There are certain errors in Java from which we cannot recover even by using try catch. eg: Out Of Memory.

Nested Try Catch ⇒ Real use of nested try catch is through functions.



Case 1: L3'' causes a NullPointer exception

L1, L2, L1', L2', L1'', L2'', L7 will be executed

Case 2: L3'' raises an IndexOutOfBoundException

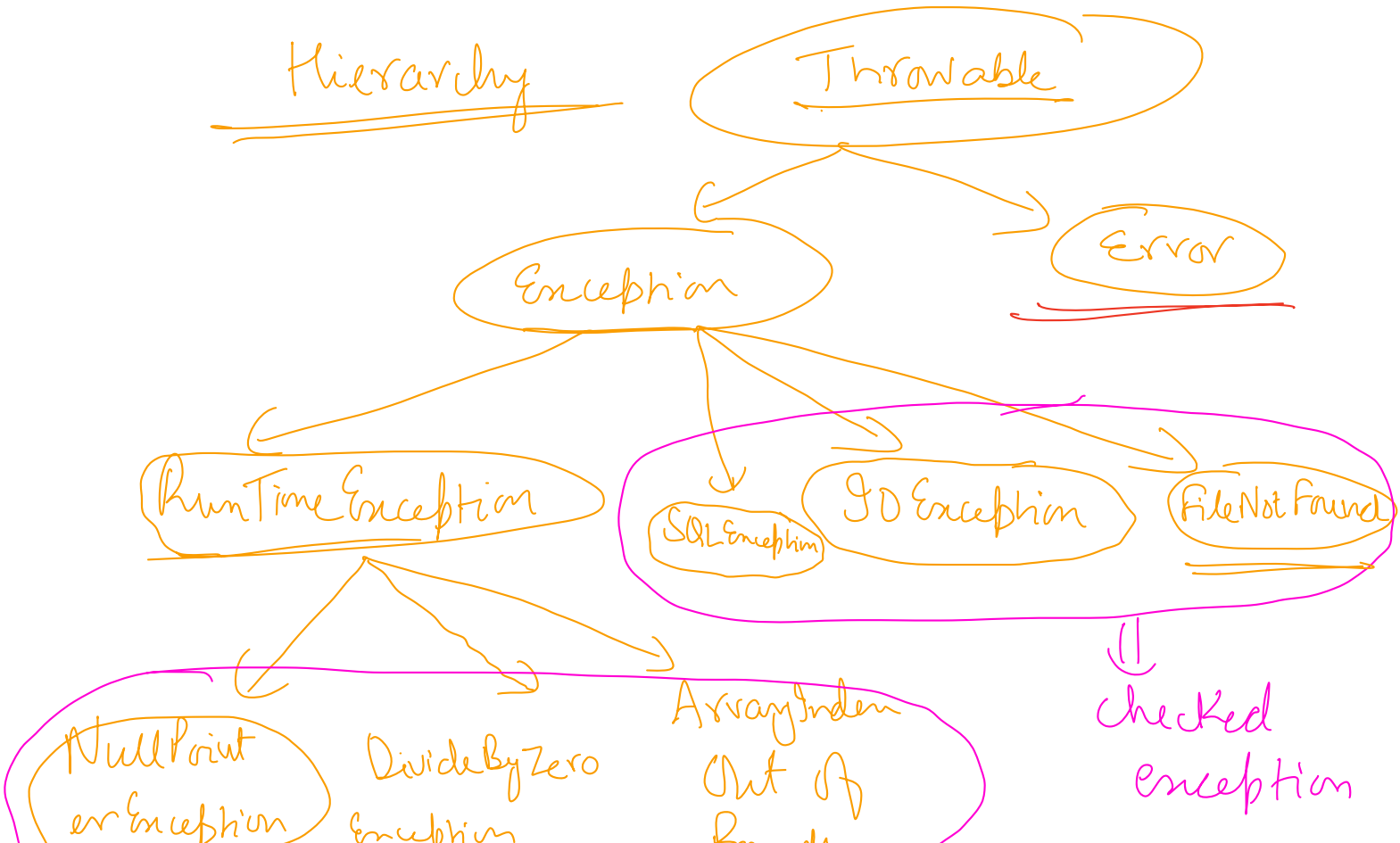
L1, L2, L1', L2', L1'', L2'', L3, L4

Case 3: : L3'' raises an FileNotFound exception

L1, L2, L1', L2', L1'', L2'' ⟹ crash after this.

Checked vs Unchecked

Hierarchy



↳ unchecked exception

(a) Checked exception ⇒ Java forces you to handle certain type of exceptions.

eg: SQLConnection c = new SQLconnection();
trying to open a connection to a database that is sitting on another person's computer in Europe.

⇒ Wi-i lost

eg: FileReader f = new FileReader( )
↳ this can cause a FileNotFound Exception;

⇒ Java forces us to write such code in (1) try catch block or declare your code as something that can cause an exception using (2) throws Keyword.

(2) Unchecked exception
↳ generally they wont happen if we

write good code.

eg: array =

0	1	2	3	4
7	2	1	0	4

array[5] ⇒ Indexout of Bounds

```
if (i>=0&&i< array.size()) {
    array[i];
}
```

this will never result in an index out of bounds

eg:

```
int a = sc.nextInt();
int b = sc.nextInt();
int ans = a/b;
```

⇒ Divideby0 exception

```
if (b!=0) {
    int ans = a/b
}
```

⇒ written good code and Divideby0 exception will never happen.

(1) If we write perfect code, we want

ever get unchecked exceptions

② In case of checked exceptions, no matter how good of a code we write, we can get an exception & thats why Java forces us to handle it.

```
{ DBConnection db = new DBConnection();
  { int ans = a/b
```

Next Class ⇒ Java Interview Questions
⇒ Strings

Next class to be on friday

9-11:30