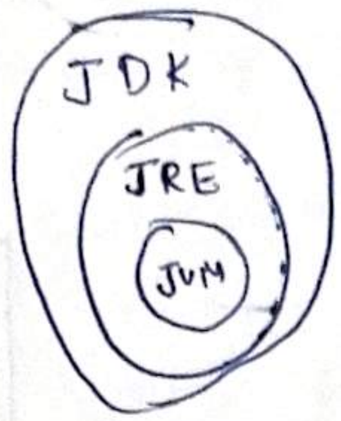
JAVA



JVM + java libraries = JRE

JRE + other development tools = JDK

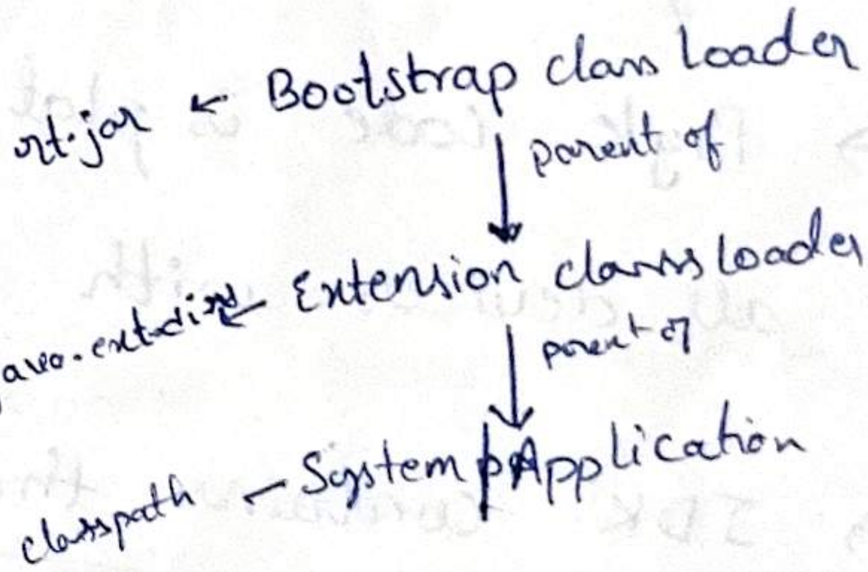
JVM → chef JRE → Kitchen

JDK → hotel

Class Loader:

It is a subsystem of JVM, that loads the class files. There are 3 types

1. Bootstrap class loader
2. Extension class loader.
3. System / Application class loader

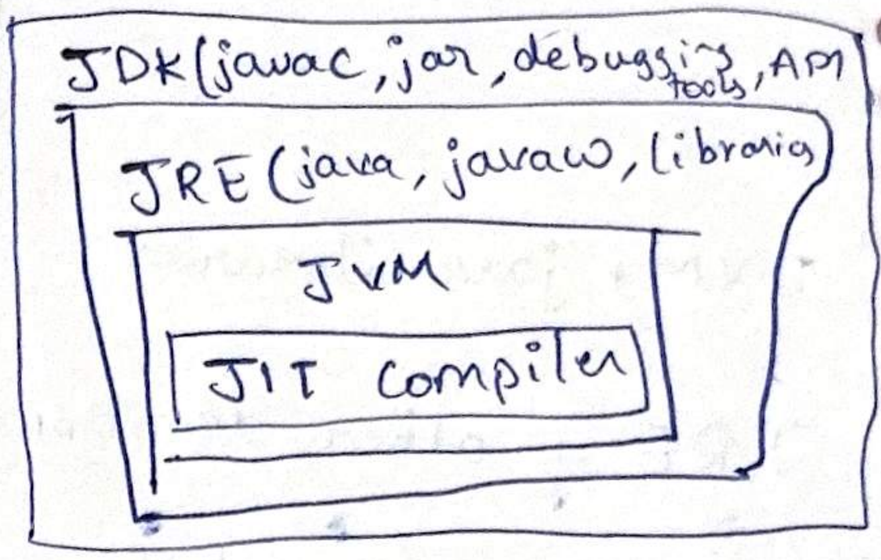


Features of Java

1. Platform independent
2. Object oriented
3. simple
4. Robust
5. Secured
6. Distributed
7. Multi Threading
8. Portable
9. High performance
10. Dynamic Flexibility
11. Sandbox execution
12. Write once, Run anywhere (WORA)
13. Power of compilation & interpretation

JRE consists of below elements

1. Class Loader
2. Byte code verifier
3. Interpreter



Execution of Java program:

→ When a .java file is compiled, a byte code is generated which is platform independent.

→ The compiler generates a .class file which has the byte code

→ Byte code is platform independent which runs on all devices with JRE

→ JDK contains the JRE & compiler, ~~interpret~~

At compile time

SourceCode.java → compiler → Byte Code

At Run time

JIT compiler → Native Machine code

Heap & stack memory in Java:

→ Java heap space is used throught the application

→ stack space is only used for the methods currently running.

Stack Memory

→ Its a temporary memory allocation, accesible only if the method contained them is currently running.

→ It allocates and de-allocates memory as soon as the method finishes execution

→ StackOverflowonError is thrown by JVM if all stack memory is completely filled

→ Accessible only to its own thread

→ Allocation & de-allocation are faster than heap memory

→ Less storage space compared to heap memory

Heap Memory

→ java.lang.OutMemoryError is thrown by JVM when heap memory is entirely full.

→ No automatic de-allocation feature is provided. Need to use garbage collector to remove the old unused objects.

→ Accessing time is slower than stack.

→ Data in heap memory is visible to all threads.

→ size is larger than stack

→ Accessible as long as the whole application runs.

→ Heap memory allocation is further divided into 3 categories to prioritize the data (objects) to be stored in Heap-Memory ~~(or)~~ & in Garbage collection.

1. Young Generation
2. Old & tenured Generation
3. Permanant Generation.

Types of Memory Areas allocated by Java:

1. Class (Method) Area → static methods & variables till java 8 → class code, method code, constructo, class variables
2. Heap → objects. Allocated to Objects at Runtime
   → static methods & variables in later versions.
3. Stack → create along with thread. Stores method data.
4. Program counter registers.
5. Native method stack.

Garbage Collection

→ Destroys objects which are no longer used

→ Free heap memory by destroying un-reachable objects.

→ 2 types of garbage collection

1. Minor (or) Incremental → when unused objects of young generation heap memory are removed
2. Major (or) Full → objects that survive minor garbage collection are copied to old generation & permanant, & these are removed

→ Happens less frequently on older generation

Types of Garbage collectors

1. Serial Garbage collector — single thread
   — pauses Application threads
2. Parallel " " → Default till Java 8
   → multiple GC threads
   → pauses Application threads, but less than parallel
3. CMS " " → concurrently GC thread with application thread
   → so, no/less pause time
4. G1 " " → default in Java 9.
   → ~~Applicat~~ More gaurentee that application Thread won't stop
   → Compaction ie putting all the allocation in one place in memory instead of scattered leaving space for new objects

Memory Leaks in Java

→ In situations where GC does not collect objects as there are references to them.

→ There might be situations where an application creates lots of objects and does not use them.

→ Since all objects had valid references, GC cannot destroy them.

→ Such useless objects are called memory leaks.

Exceptions In Java

→ Exception is an unwanted or unexpected event, which occurs during the execution of a program. ie at run time that disrupts the normal flow of the program's instructions.

→ Errors represent irrecoverable conditions such as JVM running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion etc.

→ Errors are generally beyond the control of the programmer and we should not try to handle errors.

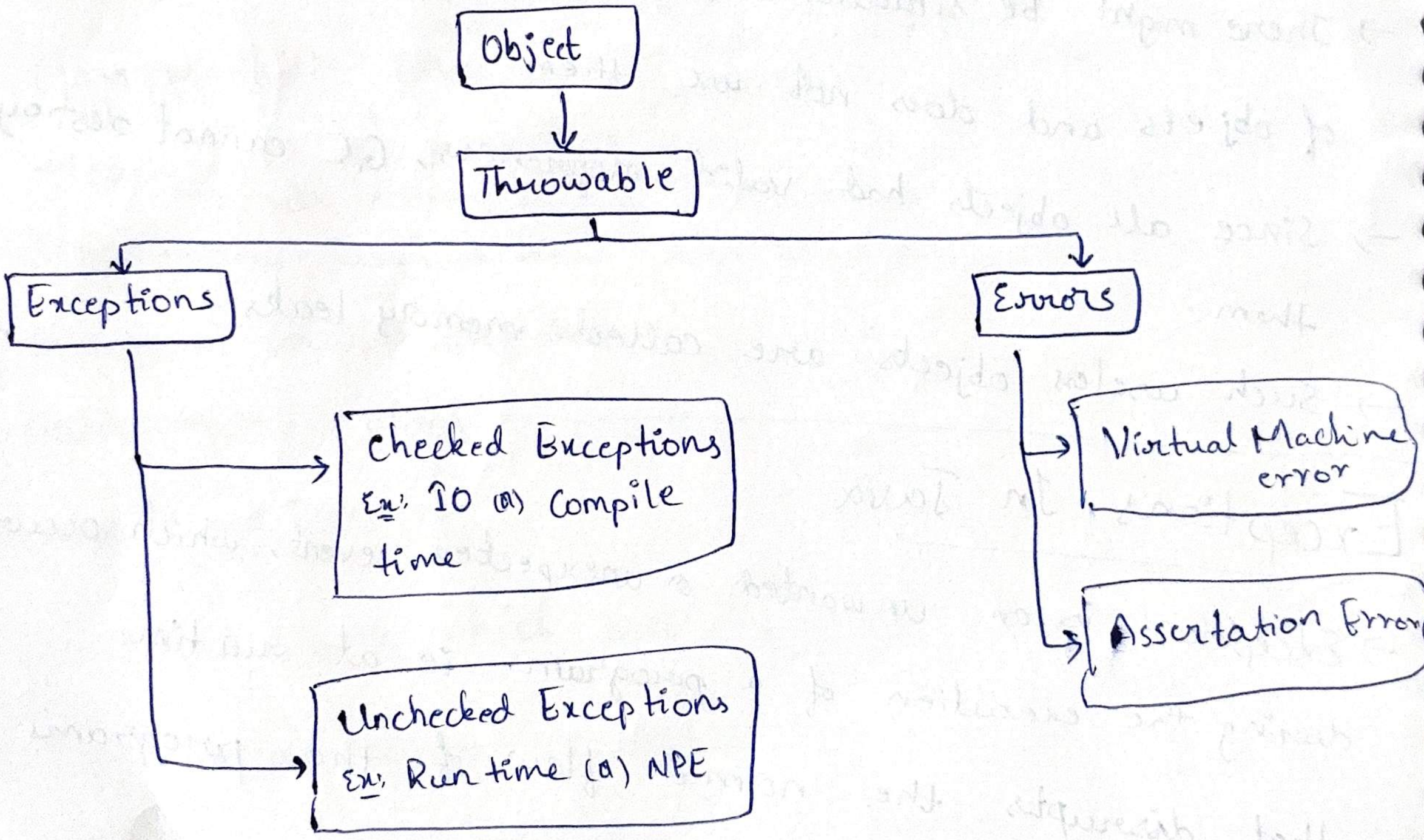
Error:

An error indicates a serious problem that a reasonable application should not try to catch.

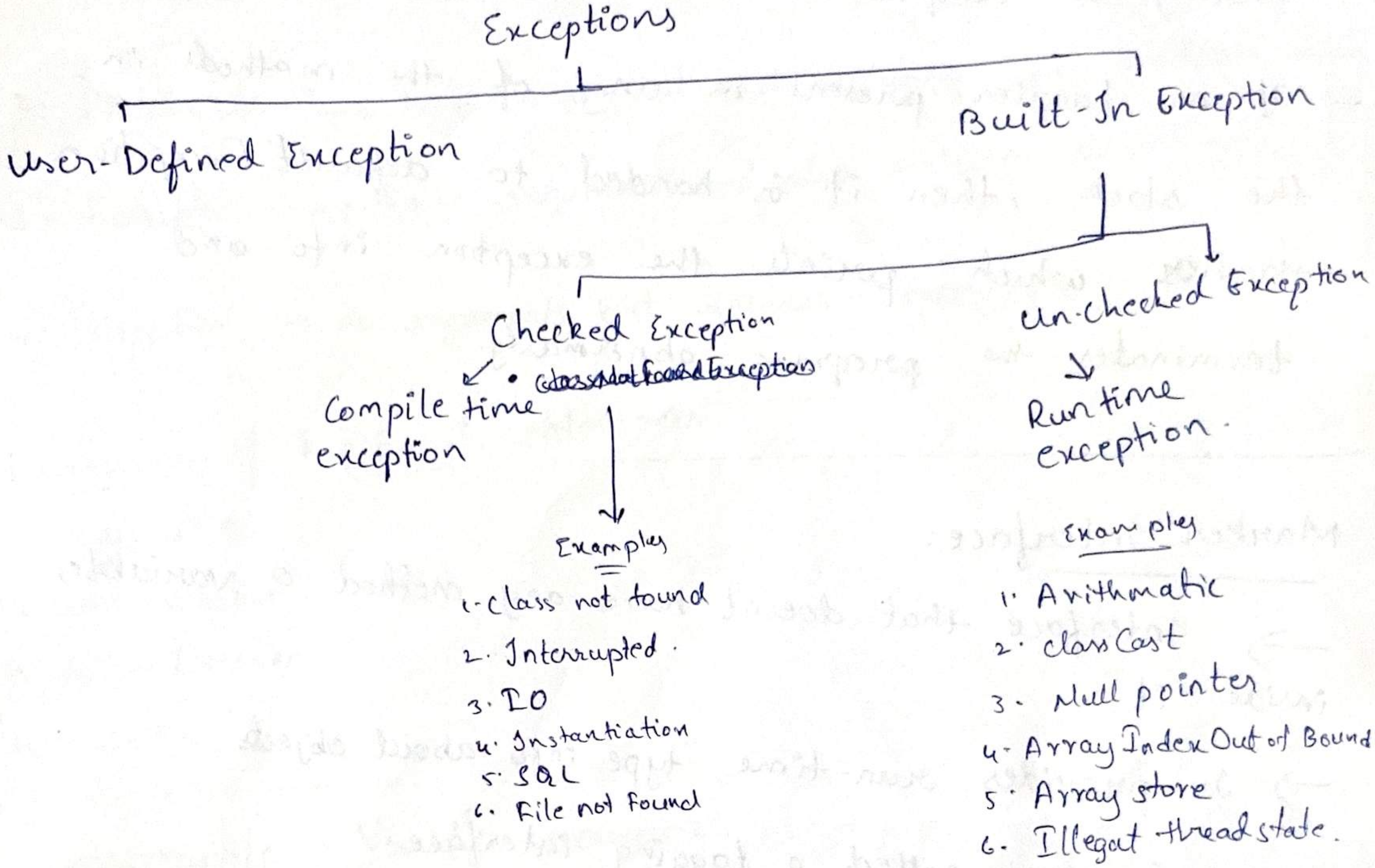
Exception:

Exception indicates condition that a reasonable application might try to catch

Exception Hierarchy



Types of Exceptions:



Advantages of Exception Handling

1. Provision to complete program execution
2. Easy identification of program code and Error Handling code.
3. Propagation of errors -
4. Meaningful error reporting
5. Identifying error types.

How Does JVM handle an Exception:

→ If an exception occurs, the method creates an Exception object and hands it off to the JVM.

→ The exception object contains the name & description of the exception & current state of program where exception occurred

→ Creating an exception and handing it to JVM is called throwing an exception

→ If no handler present in any of the methods in the stack, then it is handed to default Exception Handler which prints the exception info and terminates the program abnormally.

Marker Interface:

→ Interface that doesn't have any method & variables inside it

→ It provides run-time type info about objects.

→ It is also called a tagging interface.

→ Example of marker interface is serializable.

Functional Interface

→ An interface with only one abstract method is called functional interface.

→ Lambda functions can be used to represent the reference of a functional interface.

→ It can have any number of default and static methods.

→ @FunctionalInterface annotation is used to make sure of having only one abstract method.

→ This annotation is optional

Kinds of functional Interfaces:

1. Consumer → have arguments but does not return anything
2. Predicate → have argument and returns only boolean value
3. Function → have arguments and also return something
4. Supplier → no arguments but returns something

Examples of Functional Interfaces:

1. Runnable
2. Comparable.
3. ActionListener
4. Callable.

Comparable Vs Comparata:

Java provides 2 interfaces to sort objects using data members of the class

1. Comparable
2. Comparator.

Comparable:

→ A comparable object is capable of comparing itself with another object.

→ The class itself must implement the comparable interface to compare its instances.

→ should implement the method compareTo
→ can sort based on only ~~1 field~~ one attribute ideally its natural order ex: roll-number

Comparator:

→ Its external to the element type we are comparing
→ Its a separate class which implements comparator interface.
→ ~~Need~~ we can create multiple separate classes to ~~separate~~ compare by different members.
→ Implement Compare. method of the interface.

Design Patterns

→ It is a generic repeatable solution to a frequently occuring problem in software design thats used in software engineering.

Types of Design patterns

1. Creational
   a. Factory
   b. Abstract Factory
   c. Builder
   d. Prototype
   e. Singleton.
2. Structural
   a. Adapter → Match interfaces of different classes
   b. Bridge
   c. Composite

d. Decorator → have a plain base and use different decorators to extend it.
   Ex: Create plain pizza
   Add toppings of choice using decorators

e. Facade → A single class that represents the entire system
   Example of hotel having different restraunts & a hotel Keeper will help the customer access the menus without knowing the available restraunt options

f. Proxy → An object representing other Object. Example → debit card represent bank account.

g. Flyweight

3. Behavioral

a. Command Method → encapsulates a command request as an object
   Ex: In a remote every button is associated with a command, which executes when pressed

b. Iterator → sequentially access the members of a collection.

c. Mediator → Defines simplified communication b/w classes
   Ex: Air traffic controller communicating with all flights to decide

d. Memento → Capture & restore an objects internal state
   ex: undo/redo; saving Games.

e. Observer → notify all observers on any info a change in state
   ex: weather station notifies all it observers. Like apps, TV channel
   Its a way of notifying change to number of classes

f. State → stock market changes observed in diff apps & websites
   → Alter an objects behavior when its state changes

g. Strategy → encapsulates an algorithm inside a class

h. template → Defer the exact steps of an algorithm to a subclass
   Ex: foundation, re-info cement, concrete, slab, etc are common for all houses
   variations can be added later in concrete class impl

i. visitor

j. Null object.

SOLID Principles:

→ Solid principles helps in reducing tight coupling

S → single Responsibility
O → Open/closed
L → Liskovs substitution
I → Interface seggregation
D → Dependency Inversion.

1. Single Responsibility Principle:

→ Every class should have only one responsibility or single job
   & single purpose

→ "A class should have only one reason to change"

→ Example

2. Open/closed Principle:

→ This principle states that "software entities should be open for extension, but closed for modification".

→ This means, you should be able to extend a class behavior without modifying it.

3. Liskov's Substitution Principle:

→ According to this "Derrived or child classes must be substitute for their base or parent classes".

→ This principle ensures that any child class should be usable in place of its parent without any unexpected behavior.

4. Interface Segregation Principle:

→ Applied to interfaces not classes.

→ It states. "Do not force any client to implement an interface which is irrelevant to them".

→ The goal is to avoid fat interface and provide multiple client specific interfaces.

5. Dependency Inversion Principle:

→ It states "High level modules should not depend on low level modules. Both should depend on abstractions".

→ Abstractions should not depend on details. Details should depend on abstractions.

OOPS Principles:

1. Abstraction
2. Encapsulation.
3. Inheritance
4. Polymorphism. = Runtime - dynamic method dispatch
   = Compiletime

String Constant Pool:

→ A string acts same as an array of characters.

→ Memory allocation is not possible without string constant pool.

→ Java storing pool is a place in heap memory where all the strings defined in program are stored.

→ The immutability of strings is acheived through the use of a special ~~sp~~ storing constant pool in the heap.

→ String constant pool has the values of strings declared

→ Its a small cache resides within the heap.

→ string constant pool reduces memory usage and improve the reuse of existing instances in memory.

String Vs StringBuilder Vs StringBuffer:

Feature	String	String Buider	String Buffer
	Immutable	mutable	mutable
	Thread safe	Not thread safe	Thread safe
Memory efficiency	High	Efficient	Low efficient
performance	High (no synchronization)	High (no synchronization)	Low (due to synchronization)

Hashing:

→ Hashing is a technique of converting a large string into a small string that represents the same string.

Hash Map:

→ Stores key-value pairs.

→ Can ~~have~~ accept null values in both key and values but only 1 null for key

→ Replaces existing value if duplicate key is found.

→ Uses the technique Hashing hence called as Hashmap

→ Implements cloneable & serializable.

→ It is not thread safe.

→ Order of inversion is not maintained.

Internal structure of HashMap:

It contains an array of Node and Node is represented as a class with 4 fields.

1. int hash
2. K Key
3. V value
4. Node next.

→ Performance of HashMap depends of 2 parameters.

1. Intial capacity
2. Load Factor.

Initial capacity:

→ It is the capacity of hashmap at the time of its creation.

→ Its the number of buckets hashmap can hold when initiated.

→ Default in java is 16 pairs.

Load Factor:

→ Percentage value of the capacity after which the hashmap is to be increased.

→ Default for javas is 75% ie 0.75

Threshhold:

→ product of capacity and load factor.

→ Rehashing takes after size of map reaches the the threshold

→ Default for java is $16 \times 0.75 = 12$

Rehashing:

→ It is the process of doubling the capacity of hashmap after it reaches its threshold.

Interval working of HashMap:

Hashing: Process of converting an object into integer by using the method hashCode().

Index calculation:

index = hashCode (Key) & (n-1)

n is number of buckets.

Example:

Inserting a Key-value pair in hashmap.

map.put(new Key("vishal"), 20)

Steps:

1. Calculate hash code of key "vishal". It is 118
2. Calculate index by using index method. it is 6.
3. Create a node object as:

```
{
    int hash = 118
    Key key = "vishal"
    Integer value = 20
    Node next = null
}.
```

4. Place this object at index 6 → if no other object is present.

→ In case of collision (ie same index), ~~replace~~.

* replace if value. is same.
* attach to next node like linked list if different

Concurrent HashMap:

→ underlying data structure used is Hashtable

→ It is thread safe ie. multiple threads can operate without complications

→ At a time any number of threads are applicable for read operation without locking the concurrent HashMap.

→ Object is divided into number of segments according to the concurrency level

→ Default concurrency level is 16

→ Inserting null is not possible for key & value

→ For updation, the thread must lock the particular segment it wants to operate on.

→ This locking is called segment & Bucket Locking

→ Hence, at a time 16 updates can be performed by threads.

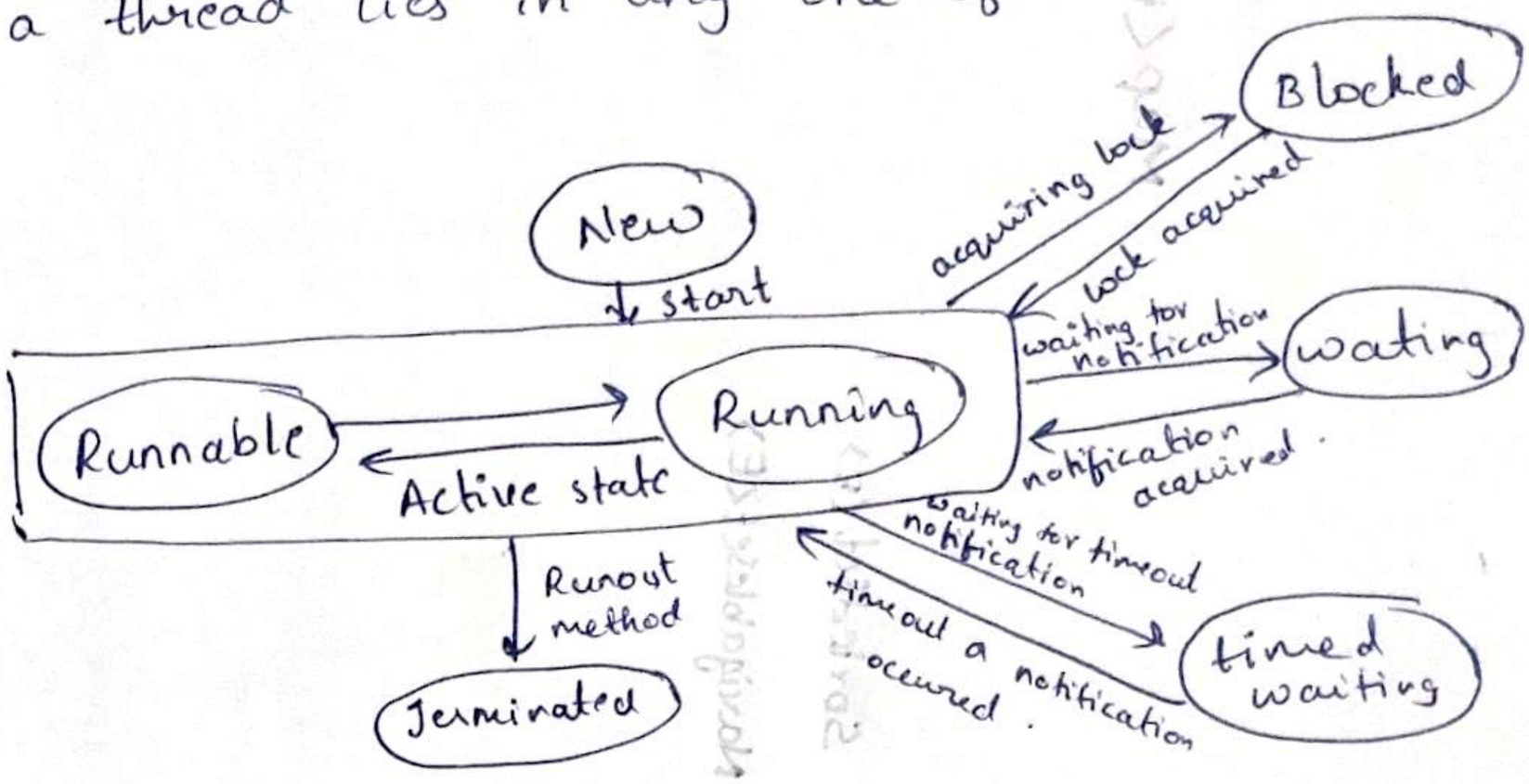
JAVA 8 Features:

1. Lambda Expressions
2. Functional interfaces.
3. Method Reference
4. Streams — intermediate
   — terminal
   — shat circuit.
5. Comparable & Comparator
6. Optional class
7. Date/Time API
8. Miscellaneous.
9. Default Methods in Interface
10. ForEach Method.

Multi Threading In Java:

At any instant a thread lies in any one of the states.

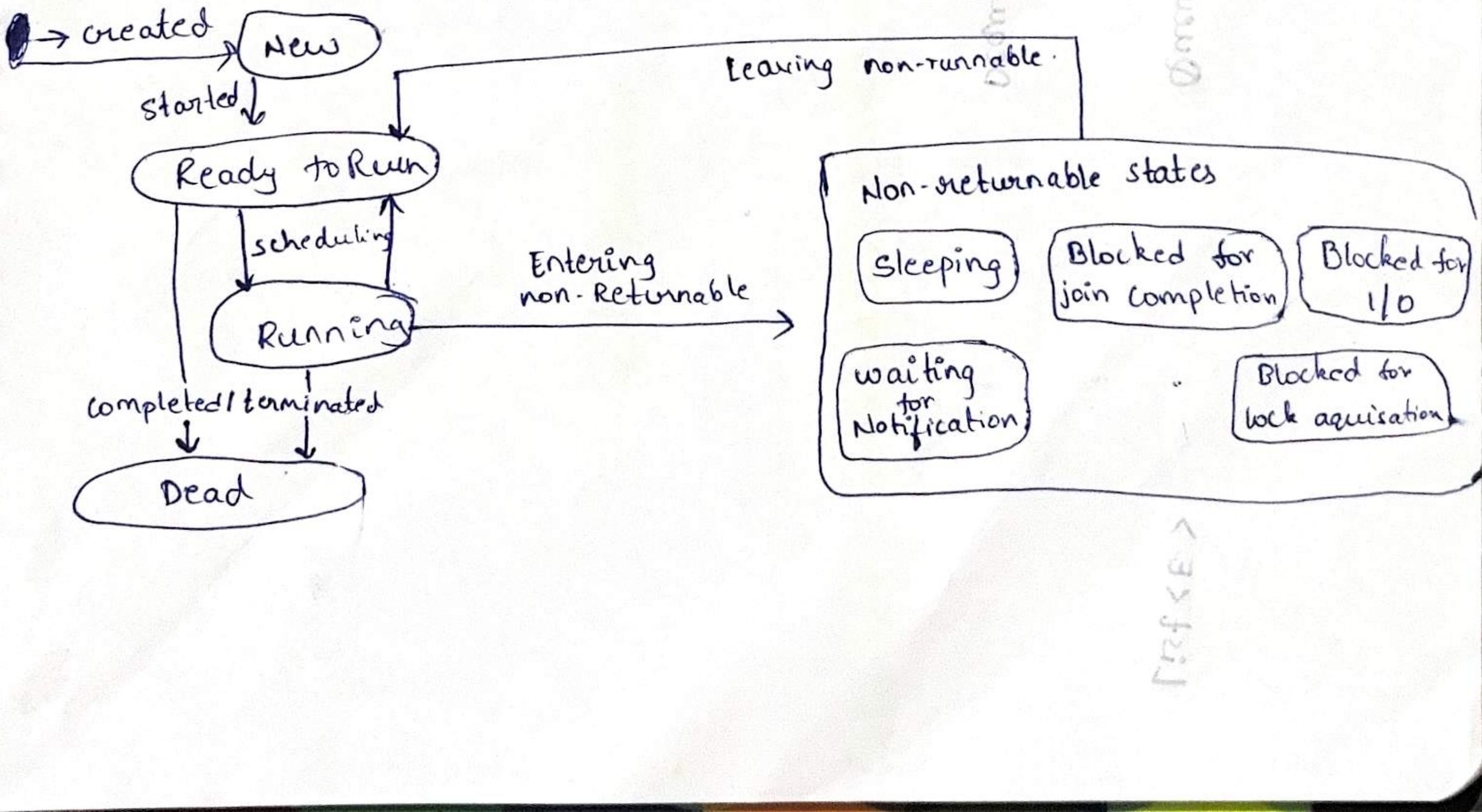
1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed wating
6. Terminated

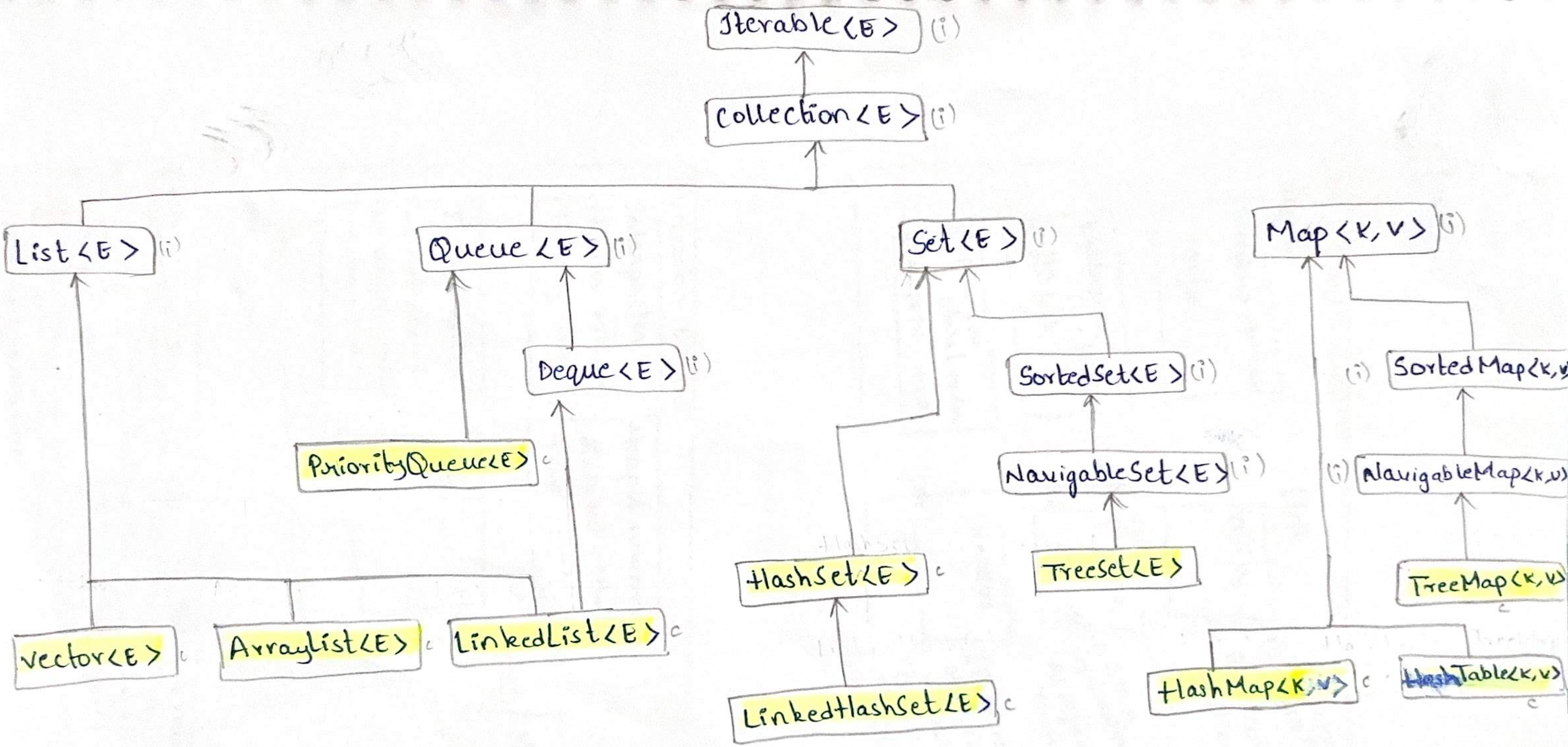


→ Threads are light weight processes within a process

→ Threads can be created using 2 mechanisms

1. Extending the thread class
2. Implementing the Runnable interface.




Iterable<E> (i)
Collection<E> (i)
List<E> (i)
Queue<E> (i)
Deque<E> (i)
Set<E> (i)
SortedSet<E> (i)
NavigableSet<E> (i)
Map<K,V> (i)
SortedMap<K,V> (i)
NavigableMap<K,V> (i)
PriorityQueue<E> c
HashSet<E> c
TreeSet<E>
TreeMap<K,V> c
Vector<E> c
ArrayList<E> c
LinkedList<E> c
LinkedHashSet<E> c
HashMap<K,V> c
HashTable<K,V> c

~~Class~~

Class Loader: