# OOAD

## Java



### Compilation Process

javac

→ class file → loaded into Classloader → check for bytecode → interpreter

→ execute

→ send it to the

CPU/hardware

class file = bytecode

### terminologies

JVM

JDK

software kit

JRE

Java Runtime Environment

JVM

### program I/O

System.out.println()

Scanner(System.in )

nextInt()

nextLong()

…

nextLine() string

## IDEs

IntelliJ

VScode

Eclipse

Visual Studio

### Online env:

replit

### Java as a language

Statically typed Language

Strongly Typed Language

### keywords

boolean byte char

false true

int short long

float double

## Data types

### Primitive Data Types

int 4

short 2

long 8

float 4

double 8

char 2

boolean 1 bit

byte 1 byte

float

12.36

1236 x 10-2

### Non Primitive Data Types

Reference Types

Arrays

Strings

Class

Objects

Interfaces

## Operators

### arithmetic

+

-

\*

/

% modulo

### assignment

=

compound

+= a += b

a = a + b

-= a = a -b

…..

### relational

<

>

<=

>=

==

!=

### logical

&& and

|| or

! not

### unary

-

a = -100

++

post

pre

### bitwise

& and

| or

^ ex-or

### shift

<<

>>

>>>

<<

0011 1100 x

0111 1000 y

0001 1110 z

0011 1100 x

1111 0000 x << 2

0000 1111

+1

0001 0000

1010 0000

1010 0000 0000 0001

0001 1111 31

0000 0011 3

0000 0100 -4

### terniray

## Flow control

### if else

if

else if

else

### switch

switch

case

break

default

### while

while

do while

break

continue

### for

for

break

continue

### for-each

loops automatically

assigns too

features:

no index

can not be effectively used to change contents of the array

moves only forward

moves only in single steps

## Arrays

* dynamically allocated
* continuous memory allocation
* objects
* [ ]

int [ ] arr = {10, 20, 30}; //length is 3

arr.length → data

String stra = “atlas” // length is 5

stra = “amazon” // length is 6

stra.length() → function

## Strings

### String

* objects
* immutable

String literal

* String Constant pool
* JVM optimize

new operator

* dynamically allocated
* heap

### String methods

str.length()

str.toUpperCase()

str.toLowerCase()

str.indexOf(“”) the index of that particular sub-string

str.charAt() the character the specified index

str.isEmpty()

### StringBuffer

a t l a s t

01 23 4

### StringBuilder

faster than StringBuffer

not thread-safe

## Recursion

-- function/method calling itself

public static int add(int num){

if (num > 0){

return num + add (num -1);

}

else {

return 0;

}

4 + *add(3)*

4 + *3 +* ***add(2)***

4 + 3 + **2 + add(1)**

4 + 3 + 2 + 1 + add(0)

4 + 3 + 2 + 1 + 0

10



## OOPs

### general concepts

Object class is a superclass

DRY

Do not repeat yourself

#### vocabulary

state

(values)

properties

behaviour

methods

identity

class (memory) division

class has:

* account number
* deposit amount
* type of account
* display()
* static data (interest rate)

objw objx objy



#### class declarations, in general:

modifiers

public

private

constructor

methods

#### declaring member variables

various kinds:

within a class → fields

within a method (or block of code) → local data

in method declaration → parameters

#### method name (conventions)

speak

speakLoudly

getData

runFast

changeRoomTemperature

#### objects

Bank objy = new Bank(234682, 3000, "savings");

Bank objy → declaration

new → instantiate

Bank(xxx, yy) → initialisation

#### garbage collection

JRE deletes objects when it feels that those objects are no longer being used

runs automatically

an object can be deleted, when:

-- no references to that object

#### Features

Encapsulation

Inheritance

Polymorphism

function/methods overloading

C++ Java

operator

C++ Python

Abstraction

## static

### static variable

only one memory location

shared among all objects

if changed for one object, gets changed for another object too

### static methods

can access only static methods & static variables

can not use non static data/method

no this keyword

no super keyword

can be accessed without creating an object

## Polymorphism

4 + 6

“hello” + varx

Static

compile time

go\_for\_dinner(Saturday)

Runtime

Run time

### Function/Method Overloading (points to remember)

* num of parameters
* data types of parameters
* order of type of parameters
* not on the return type
* static method can be overloaded

### Function Overriding (points to remember)

* different (or specific) implementation in a child/subclass
* static methods can not be overridden
  + unless inherited class also makes it static (hides the parent’s implementation)
* method name in parent & child should be same
* parameter list & order should be same in parent & child

## Encapsulation

capsule

hide the data

## Inheritance

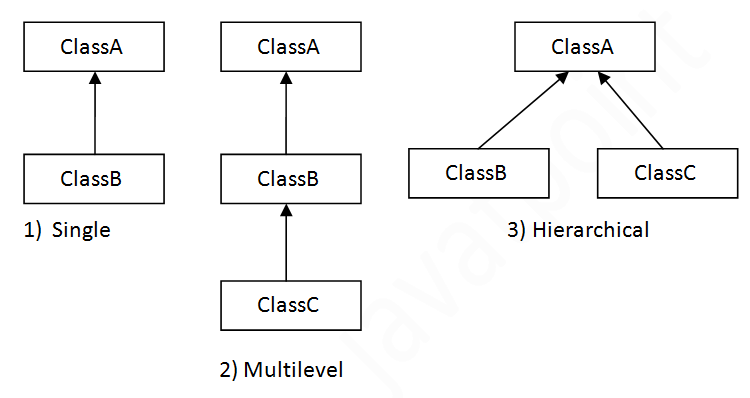
parent class or super class

child class or subclass

Unix()

Linux()

IS-A



### inheritance (points to remember)

private attributes are not inherited

multiple inheritance not supported through class

### super

* used to invoke parent class constructor
* use to invoke parent class method

super.method()

* used to refer parent class data variable

### final

* final variables (data) can not change their value
* final methods can not be overridden
* final class can not be inherited
* final methods can be overloaded

## Object & Class Relationships

Association

Composition

Aggregation

### Association



association manages:

one-to-one

one-to-many

many-to-many

two forms of association:

composition

aggregation

## Example

flight\_one

Seat:

| [empty]  [food] | [empty]  [food] | [empty]  [food] | [empty]  [food] |
| --- | --- | --- | --- |
| [empty]  [food] | [empty]  [food] | [empty]  [food] | [empty]  [food] |
| [empty]  [food] | [empty]  [food] | [empty]  [food] | [empty]  [food] |
| [empty]  [food] | [empty]  [food] | [empty]  [food] | [empty]  [food] |

flight\_number =

single object

Seat obja = new Seat();

Seat(){

// constructor

}

multiple objects (array form)

Seat[] arr = new Seat[2];

arr[0] = new Seat();

arr[1] = new Seat();

Seat(){

// constructor

}

## constructor

same name as that of the class

no return type

can not be called explicitly

gets called implicitly (automatically) when an object is created

if not defined, then JVM will provide a default (empty) constructor

can not be static

can not be abstract

can not overridden

can be overloaded (parameterised)

can not be final

invoked when a new object is created

can have all access modifiers

# abstraction

hiding the details

* abstract
* interfaces (100% abstraction)



### abstract classes

#### points to remember

* abstract keyword for class (mandate)
* abstract & non-abstract methods
* even one abstract method means, class should be abstract
* no instantiation (no objects can be created)
* can have final methods
* can have constructors
* can have static methods

### interfaces

#### points to remember

* it's not a class, its an interface
* interface keyword (mandate)
* all methods have to be abstract & public, unless static
* data variables
  + public
  + static
  + final
* no instantiation (no objects can be created)
* multiple inheritance
  + interface can extend one or more interface
  + interface can extend only another interface
  + class can implement more than one interface
    - abstract
    - non-abstract
* non-abstract class implementing an interface
  + concrete implementation of all abstract methods
  + maintain exact signature of a method
* can have private methods (java 9 onwards)
* can have static methods (java 8 onwards)
* can have overloaded methods
* can not have constructors in interfaces

## inner classes and nested classes

nested classes

inner classes

* inner class
* within method (local classes)
* anonymous inner class

static nested classes

terminology/vocabulary

nested class

outer class

### points to remember

* need outer class for instantiating inner class object
* local inner classes can not be invoked from outside the method
* nested interfaces (static)
  + will not be private
* object of static nested class can be created without outer class object
* anonymous class
  + can be abstract too

## packages

collection of classes, interfaces & sub packages

helps organise classes into a folder structure

easy to find them

reusability

namespace

access specifiers

public

can be accessed from outside/within the package/within the class

private

can be accessed from within the class

protected

can be accessed from within the package/within the class

## Exception Handling

Object

-- Throwable

-- Error

-- stack overflow

-- no memory

-- OS send a signal

-- JRE/JVM error

-- Exceptions

-- Checked Exceptions

discovered during compile time

-- Unchecked Exceptions

-- user defined Exceptions

Throwable

Exception

RuntimeException

NullPointerException

IOException

java.lang

### keywords

try

catch

finally

### finally

1. exception occurs, is handled

catch block executes

finally block executes

code continues

1. exception occurs, is not handled

finally block executes

code crashes

1. exception does not occur

finally block executes

code continues

### throw

can explicitly raise exceptions

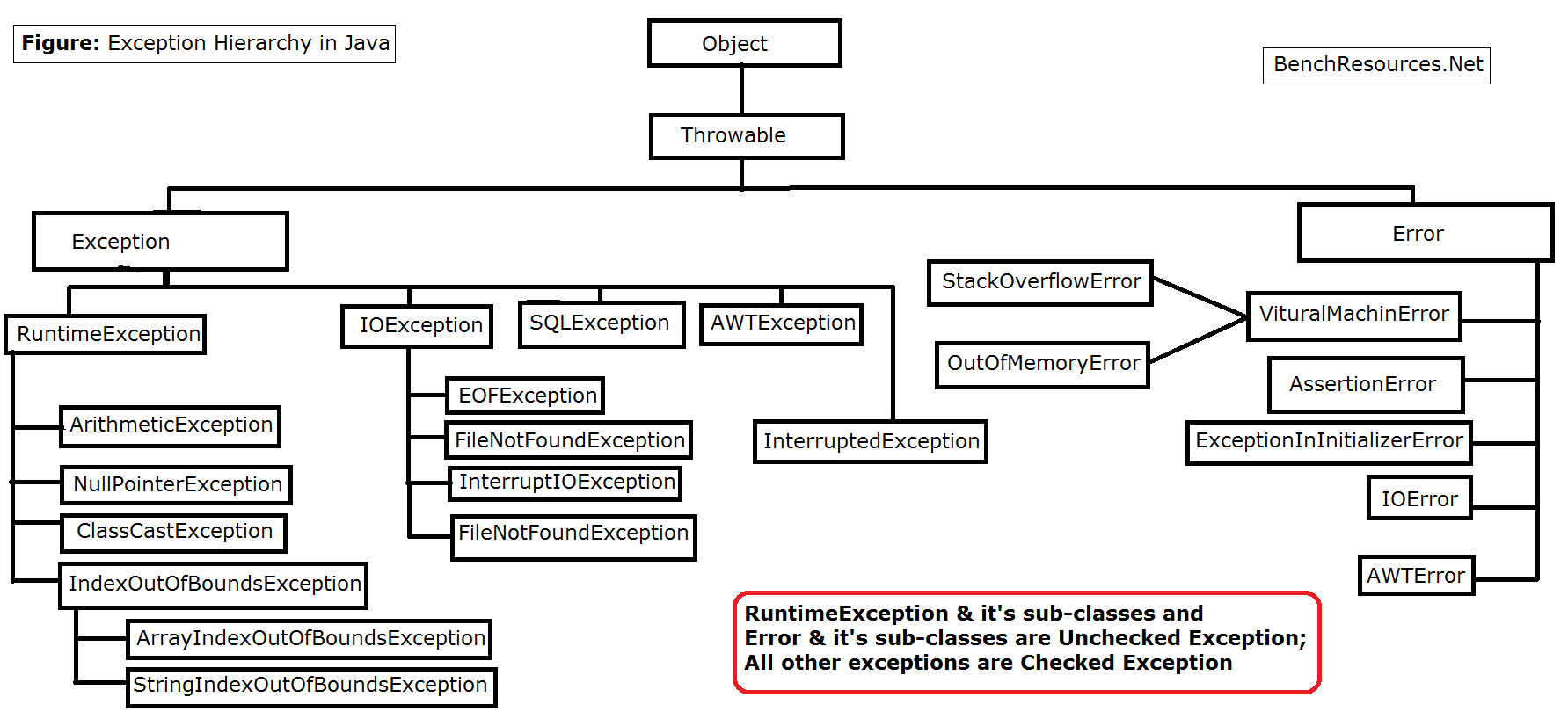
while throwing unchecked exceptions

can either catch it

or let system display the exception & stop the code

while throwing checked exceptions

mandatory to catch the exceptions



## Math

Math.min(x,y)

Math.max(100, 200)

Math.pow()

Math.sqrt()

x = 23 x = -12

y = Math.abs(x)

intervals = [[3,4],[2,3],[1,2], [5,6], [4536789012, 5]

i = 0

intervals[0] = [3, 4]

j = ? (0, 1, 2, 3)

res = [-1, 0, 1]

i = 1

intervals[1] = [2, 3]

j = 0, 3

[3, 4] or [5, 6]

i = 2

intervals[2] = [1, 2]

j = ? (0, 1, 2, 3)

0, 1, 3

i = 3

intervals[3] = [5,6]

j = ? (0, 1, 2, 3)

## Comparable

### equals()

rules:

reflexive → an object must equal itself

symmetric → x.equals(y) must return the same result as y.equals(x)

transitive → if x.equals(y) and y.equals(z) then x.equals(z)

Comparable

index0.compareTo(index1)

Comparator

compare(index0, index1)

## Collections

interfaces

List

Set

Deque

Map

Implementations

ArrayList

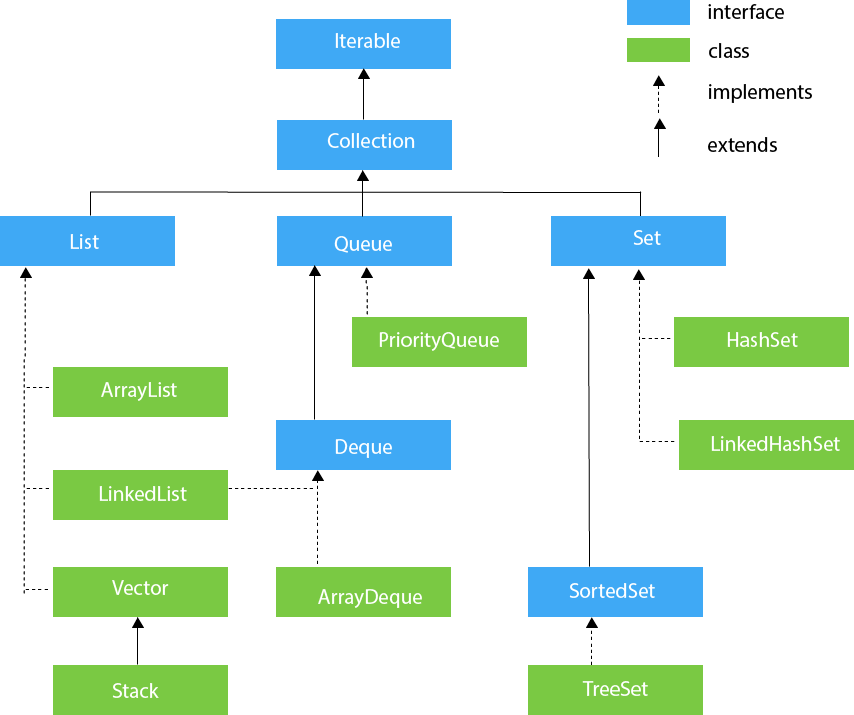
HashSet

TreeMap

Algorithms

sort()

search()



### Collection interface methods

Modification methods

Add

add()

addAll()

Remove

clear()

remove()

removeAll()

retainAll()

Query methods

contains()

equals()

size()

isEmpty()

other methods

iterator()

toArray()

### List

Modification methods

Add

add(Element el)

add(int index, Element el)

addAll(Collection cl )

addAll(int index, Collection cl )

Remove

clear()

remove(int index)

remove(Object obj)

removeAll(Collection cl)

retainAll(Collection cl)

modify elements

set(int index, Element )

Query methods

contains(Object obj)

equals(Object obj)

get(int index)

indexOf(Object obj)

lastIndexOf(Object obj)

size()

isEmpty()

subList(int from, int to)

other methods

iterator()

listIterator()

listIterator(int index)

toArray()

List

ArrayList

LinkedList (also implements Deque)

### ArrayList

default size = 10

.add()

.add(index)

.set()

.get()

.get(.size() -1 )

.size()

.indexOf()

.lastIndexOf()

.contains()

* re-sizeable
* starts with size 10
* internally uses array to store the elements
* can retrieve elements by index
* allows duplicate values
* allows null values
* ordered collection
* not synchronised

### Set

* unordered
* no duplicate values

### HashSet

* unordered
* unique
* uses hashing for storing the values
* allows null value
* 16 default size
* not synchronised

### LinkedHashSet

* ordered (maintains insertion order)
* unique elements
* null values allowed
* not synchronised

### TreeSet

* Set → SortedSet → TreeSet
* Tree DS
* ordered (maintains ascending order)
* unique elements
* null values not allowed
* not synchronised
* compare() compareTo()

#### TreeSet methods

add

contains

remove

first()

last()

pollFirst() return/remove lowest element

pollLast() return/remove highest element

isEmpty()

clear()

| HashSet | LinkedHashSet | TreeSet |
| --- | --- | --- |
| unordered | insertion order | ascending order |
| internally uses HashMap | internally uses LinkedHashMap | internally uses Tree |
| one null object | one null object | no null objects  NullPointerException |
| equals()  hashCode() | equals()  hashCode() | compare()  compareTo() |
| O(1) | O(1) | O(log(n)) |

### HashMap

key:value

* unordered
* no duplicate keys
* allows null value
* allows null key
* not thread-safe

#### HashMap methods

put()

putIfabsent()

isEmpty()

size()

containsKey()

containsValue()

get(key)

remove(key)

remove(key, val)

keySet()

values()

forEach

iterator()

### TreeMap

* Sort (keys)
* custom sorting (comparator)
* no duplicate keys
* no null key (null values are allowed)
* no thread safe (not synchronised)

pollFirstEntry()

pollLastEntry()

## Java8 Features

Functional Interfaces

Lambda Expressions

forEach()

Method References

Stream API

Stream Filter

Default Methods

Collectors class

### Functional Interfaces

* one abstract method only

Runnable

run()

@FunctionalInterface

### lambda

() -> {

body

}

## File I/O

1. Scanner class
2. BufferedReader
3. File Reader

# Data Structures (Java)

## types of DS

primitive

byte, short, int ….

non primitive

1. Linear Data Structure
   1. Array
   2. Linked List
   3. Stack
   4. Queue

…..

1. Non-Linear Data Structure
   1. Tree
   2. Graph

## Array

linear data structure

collection of data/variables/objects

int, float

Strings

objects

referenced by a common name

contiguous memory location

data type = base type

Arrays are considered as objects

heap area (dynamic memory)

new

size of array = int

length

public final length

Object class is a superclass of array

Linear Data Structure

advantages:

random access of elements (using index)

access is faster

sorting, iteration is easier

modification (of values )

disadvantages:

size is fixed

modification (insertion, deletion) cumbersome

time taking

capacity is huge, used section is less, then memory is wasted

always needs contiguous memory locations

for(int val : arr)

{

// body of the loop

}

### clone

#### while passing to a function:

arr

| 10 | 20 | 30 | 40 | 50 | 60 |
| --- | --- | --- | --- | --- | --- |

brr = arr

brr was referencing back to arr

crr = arr

crr was referencing back to arr

any changes done to crr, effects arr too

#### while passing to a function by clone, or by equating using cloning:

arr

| 10 | 20 | 30 | 40 | 50 | 60 |
| --- | --- | --- | --- | --- | --- |

brr = arr.clone()

brr is a new copied version of arr

separate area in the memory is created

crr = arr.clone()

crr is a new copied version of arr

any changes done to crr, does not effect arr

crr

| 6 | 16 | 26 | 36 | 46 | 56 |
| --- | --- | --- | --- | --- | --- |

### multi dimensional arrays

arr[m][n]

m rows

n columns

m number of n arrays

arr [6][4]

6 one-dimensional arrays

each one dimensional array is of the size 4

ways of creating multi dimensional array:

int arr [ ] [ ] = new int [6] [4]

int [ ] [ ] arr = new int [6] [4]

## Linked List

tiffin box mould

start = 7000

| 8055 |
| --- |
| chole bhature |
| 7054 |

7000

| 7000 |
| --- |
| momos |
| 8000 |

7054

| 7054 |
| --- |
| pasta (red sauce) |
| 9011 |

8000

| 9011 |
| --- |
| shawarma |
| 8055 |

8020

| 8020 |
| --- |
| biryani |
| 7000 |

8055

| 8000 |
| --- |
| chicken masala |
| 8020 |

9011



## Stack

* LIFO
* top

insertion & deletion

### major operations

push

insert element at the top

pop

removes the top element

-------

isEmpty()

return True if it is empty

look() peek()

take the value from the top, but do not delete it

## Queue

* FIFO
* Q

## Hashing

key value (pairs)

### 

319540

→ 42605

→ 665

A

B

C

D

XYZ

-- one-directional process

-- deterministic

-- uniformity

-- fixed size output

-- avoid collision

-- Message Digest

SHA-256

Secure Hash Algorithm

-- large amount of data

smaller table

### Hash Functions

-- Message Digest Functions

340: ironman:suit

341: captain: hammer\_shield

357: thor: hammer\_thunder

### Hash table

Hash Map

0 to n

division method

remainder

| Data | Hash Function | Hash Value |
| --- | --- | --- |
| 23 | 23 % 10 = 3 | 3 |
| 37 |  | 7 |
| 29 |  | 9 |
| 48 |  | 8 |
| 54 |  | 4 |
| 35 |  | 5 |
|  |  |  |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 23 | 54 | 35 |  | 37 | 48 | 29 |

table size = 10

load factor = num of items/ table size

6/10

H(77) = 7

i 105

d 100

l 108

e 101

“abcf” “bcfw” “cvrh” “guio”

396 418

“idle” 414

“idel” 414

“deli” 414

“lied” 414

“idle” (105\*10 + 100\*20 + 108\*30 + 101\*40)%599 =

“idel” 414

“deli” 414

“lied” 414

Collision / Clash

Linear Probing

## Sorting Algorithms

rearrange the positions

### Bubble Sort

{88, 14, 32, 25, 79}

**88 14** 32 25 79

14 **88 32** 25 79

14 32 **88 25** 79

14 32 25 **88 79**

-------------------------------------------

**14 32** 25 79 88

14 **32 25** 79 88

14 25 **32 79** 88

14 25 32 **79 88**

* two loops (inner & outer)
* keep swapping in the inner loop
  + if current element is smaller than the adjacent element (then swap)
* repeat till the outer loop exhausts

### Insertion Sort

13 15 17 18 14 12 11 19 16

**13 15 17 18** 14 12 11 19 16

**13 15 17 14** **18** 12 11 19 16

**13 15 14 17** **18** 12 11 19 16

**13 14 15 17** **18** 12 11 19 16

13 14 15 17 18 12 11 19 16

1. pick a key (an index actually)
2. repeat steps 2 to 4 till the end of array is reached
3. compare the element at current index with the left values, if it is smaller then repeat step 3
4. keep shifting elements from the “sorted” section of the array till the correct location of the key is found
5. increment the loop variable (i)

### Selection Sort

### Merge Sort

divided & conquer algorithm

8 3 1 6 17 14

8 3 1 6 17 14

8 3 1 6 17 14

3 8 1 6 17 14

1 3 8 6 14 17

1 3 6 8 14 17

a b c d e

a b c d e

a b c

mergeSort()

merge()

### Quick Sort

{ 88, 14, 32, 25, 79 }

25

14 25 88 32 79

32 88 79

79 88

14 25 32 79 88

## Graphs

-- Finite Graph

-- Infinite Graph

complete graph

(every node is adjacent to every other node)

vertices = n

each vertex’s degree = n-1

number of edges = (n \* (n-1) )/2

max cut vertex n-2





Digraph

Connected Graph

edge connections

vertex node

degree

out-degree

in-degree

source vertex (in degree = zero)

path

strongly connected

(or has a path, in terms of directed graphs)

Adjacency matrix

adjacency list

cut vertex

### BFS



Visited → A,

Queue → B, C, D

Visited → A, B

Queue → C, D

Visited → A, B , C

Queue → D, E

Visited → A, B , C, D

Queue → E

Visited → A, B , C, D, E

Queue →

O (ver + ed)

### DFS



Visited → A,

Stack → B, C, D

Visited → A, B

Stack → C, D

Visited → A, B, C

Stack → E, D

Visited → A, B, C, E,

Stack → D

Visited → A, B, C, E, D

Stack →



S.visited = true

A.visited = true

D.visited = true

B.visited = true

C.visited = true

stack → S

S A

S A D

S A D B

S A D C

S A D

S A

S

(empty)

Display:

S A D B C

### Colouring



example



0 → 1 → 2 → null

1 → 3 → null

2 → 3 → null

3 → null

## Advanced Algorithms (techniques)

### Greedy Algorithms

intuition

human

Fractional Knapsack problem

Pros:

1. easy
2. time complexity is generally good

Cons:

1. does not guarantee best solution

(not essentially optimised)

n pipes diff lengths

cost = sum of the lengths



Nurse & patient appointments

1. 7:00 7:15 N1
2. 7:30 9:50 N1
3. 7:40 9:10 N2
4. 9:00 9:30 N3
5. 12:00 13:00 N1
6. 14:00 14:30 N1

Nurse & floor

doctor & nurse are not at the same floor

nurse is useless in current floor

nurse is allowed to travel max K floors from the current location (max\_travel)

1 N

2 D

3 D

4 N

5 D

6 D

7 N

max\_travel = 1

num of doctors assisted = 1 + 1 + 1 = 3

scenario B

1 N

2 D

3 D

4 N

5 D

max\_travel = 2

num of doctors assisted = ?

scenario C

1 D

2 D

3 N

4 N

5 D

6 D

7 D

8 N

max\_travel = 2

num of doctors assisted = ?

### Dynamic programming

#### fib (recursion)

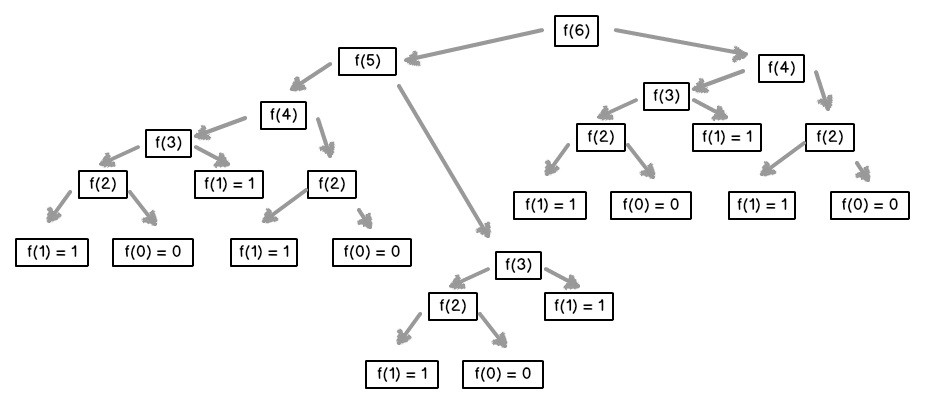
fib(6) →fib(5) → fib(4) → fib(3)

→ fib(1)

→ fib(2)

→ fib(1)

→ fib(1)



O(2n )

optimal substructure property

lookup table for overlapping subproblems

#### dynamic programming patterns

memoization

look for answer of a subproblem in a lookup table

top down

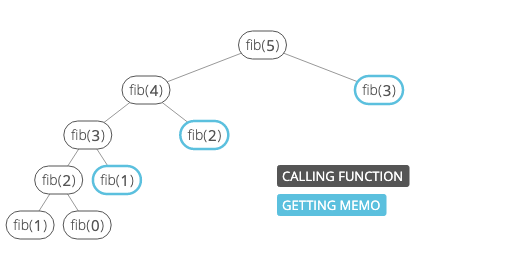
tabulation

fill the look up table with subproblems

compute the solution

bottom up

##### memoization



1 egg at 1 time

2 egg at a time

3 eggs at a time

total num of eggs = 4

3 1 1 way

2 2 2 ways (2 1 1) or (2 2)

1 3left 4ways (1 3) or (1 2 1) or (1 1 1 1) or (1 1 2)

total num of eggs = 3

3

2 1

1 2 (1, 1,1) (1, 2)

*4 ways*

total num of eggs = 2

1 1

2

*2 ways*

total num of eggs = 1

1

*1 way*

| eggs | ways |
| --- | --- |
| 1 | 1 |
| 2 | 2 |
| 3 | 4 |
| 4 | 7 |
| 5 | 13 |

5 eggs

3 2 ( 2ways)

2 3 (4 ways)

1 4 (7 ways)

## Complexity measures - time

1. best case
2. worst case
3. average case

Asymptotic analysis

f(n) g(n)

Big O Notation

f(n) is considered O(g(n)) c, and an integer n0 > 0

such that following inequality holds for all n >= n0

f(n) <= cg(n)

constant any constant

logn logarithmic

log2n log square

root of n root-n

n linear

nlogn linearithmic

n2 quadratic

en exponential

n! n-factorial

general:

C x length O(n)

half at every iteration O(logn)

common scenarios

1. simple for loop (in of size 1)

for(int i=0; i<num;i++){

sum++; // sum = sum + 1 3n

}

O(n)

1. simple for loop (in of size k)

for(int i=0; i<num;i++){ //

sum++;

}

O(n)

1. nested for loop

for(int i=0; i<num;i++){

for(int j=0; j<m; j++){

sum++;

}

}

O(nm)

1. nested for loop with dependent variables

for(int i=0; i<n;i++){

for(int j=0; j<i; j++){

sum++;

}

}

O(n2)

1. nested for loop with index modification

for(int i=0; i<n;i++){

i = i\*2;

for(int j=0; j<i; j++){

sum++;

}

}

O(n)

outer loop outer loop times inner loop inner loop times

i=0 1 i = 0\*2 = 0 0

i=1 2 i = 1\*2 = 2 1

i=3 3 i = 3\*2 = 6 3

i=7 4 i = 7\*2 = 14 7

i=15 5 i = 30 15

i = (n-1) i = (n-1) \*2 = 2(n-1)

if i is doubling

1 to n in roughly log2(n-1) steps

outer loop index

1, 2, 4, 8, 16 ……. 2 log2(n-1)

outer loop

2log2(n-1) + 2

20 21 22 23 2 …… 2 log2(n-1)

2 log2(n-1)+1 - 1

= 2 log2(n-1)2 - 1

= 2(n-1)

=2n-3

inner loop

2(2n-3) + 2 + c(2n-3)

= 2n(2 + c) - 3c - 4

2n(2 + c) - 3c - 4 + 2log2(n-1) + 2

O(n2)

while(i<n) {

i\*=k;

// statements that take constant time

}

O(log(n))

n=16, k=2

i=1 count=1

i=2 count=2

i=4 count=3

i=8 count=4

i=16 count=5

log2(16) = 4

### common tips:

#### constant time

O(1)

constant runtime algorithms

* value of a map
* check if item is null
* lookup table

#### linear time

O(n)

* find an item in a collection
* print/assign all values in a list
* get max/min values in an array

### logarithmic

O(log n)

* binary search
* dividing the problem into half

### Linearithmic

O(n log n)

* merge sort
* quick sort

### Quadratic

O(n2)

* if collection has duplicate values
* all possible ordered pairs
* bubble sort, insertion sort, selection sort

#### Exponential

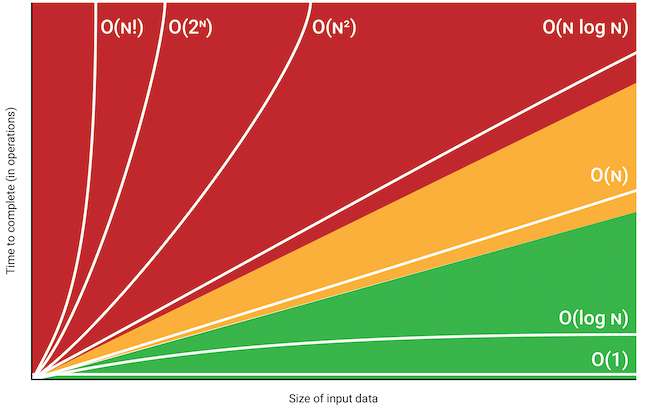
O(2n)

* fibonacci
* travelling salesman
* finding all sunsets (power set)

#### Factorial time

O(n!)

* permutations of a string



## space complexity

### constant space

O(1)

constant flags, temporary variables etc. , 1 array/collection

in place sorting

* bubble sort
* insertion sort
* selection sort
* heap sort

### O(n)

* recursive array creation
* quick sort
* merge sort

O(k)

* counting sort

|  |  |  |
| --- | --- | --- |
| array |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

check if double linked list is an palindrome

# OOAD

## UML

Unified Modelling Language

* specifications
* construction
* visualisation
* Documentation
* generalised
* pictorial language

#### UML softwares

starUML

lucidchart

diagrams.net

MS visio pro

draw.io

edraw

figma

#### Conceptual model

entities

relationships

OOP

### UML diagram

categories

Structural

* class diagram
* object diagram
* package
* deployment

behavioural

* activity
* state machine
* use case

Interaction

* timing
* sequence
* collaboration

### UML Building Blocks

1. things
2. relationships
3. diagrams

### Structural things

static

(will become actual part of the system)

noun

#### Class Diagram

class name

top compartment

syntax : bold

centre

start with upper case

attributes

compartment below the class name

derived attribute

/

visibility:-

+ public

- private

# protected

~ package

operations (methods)

#### abstract class

prototype

zero/many operations

name syntax: italics

Relationships

1. dependency

dotted arrows

1. generalisation

bold arrow

1. association

simple line

mostly verb or noun

1. aggregation

special association

whole-part

1. composition

special aggregation

ownership is strong

#### multiplicity

0..0 0

0..1

1..1 1

180..180 180

0..\* \* zero or more

1..\* at least one, any number

### Use case diagrams

dynamic behaviour of a live system

actor (and actor’s name) should be relevant to the system

include

extend

### State Machine Diagram

State Diagram

Statechart

State Transition Diagram

### Activity Machine Diagram

### interaction

Lifeline

name

type

messages

communication between lifelines

synchronous

asynchronous

return message

object creation

destruction

self message (reflexive message)

#### Sequence Diagram

notations

## Best Practices

* scalable
* modular
* maintain
* performance

GRASP

General Responsibility Assignment Software Patterns

### Object/class behaviour

#### coupling

Tight Coupling:

one class tightly coupled with another means dependency or creating objects of

class with another

Loose Coupling:

removing dependencies at class level

introducing dependency at final stage of application

making it flexible

#### association

no has-a relationship

#### composition

has-a relationship

strong

(inheritance is-a)

#### aggregation

has-a relationship

weak relationship

#### cohesion

#### polymorphism

### SOLID

Single Responsibility principle

Open-Closed principle

Liskov Substitution principle

Interface Segregation principle

Dependency Inversion principle

#### Single Responsibility

single functionality

single job/purpose

only one reason to change

#### Open Closed Principle

software entities

open for extension

closed for modification

#### Liskov Substitution Principle

subclasses should be substitutable for base classes

class B is a subtype of class A,

B should be replaceable with A without disrupting the behaviour

#### Interface Segregation

larger interfaces should be split into smaller ones

#### Dependency Inversion principle

decoupling software modules

make high-level modules & low level modules depend on abstraction

### GRASP

#### Low Coupling

low dependencies between software artefacts

try to have dependencies via interfaces

#### High cohesion

#### polymorphism

#### information expert

expert

expert principle

#### indirection

how to let objects interact in a manner that the bond among them remains weak



#### DRY

Do not repeat yourself

ATM - 6 menu

ATM - 8 menu

#### Hollywood principle

Don’t call us, we will call you

* similar to DIP

### good practices:

* separate parts of code that vary, from code which remains same
* code to an interface and not an implementation
* encapsulate as much as you can
* polymorph
* composition over inheritance
* interacting components, should be as loosely coupled as possible
* inhibit modification, encourage extension
* DRY
* use patterns

## Design Pattern

Gang of Four

GoF Patterns

### broadly 3 categories

#### Creational

Singleton

Builder

Factory

Abstract Factory

Prototype

#### Structural

Adapter

Bridge

Composite

Decorator

Facade

Flyweight

Proxy Pattern

DAO

#### Behavioural

Chain of Responsibility

Command

Iterator

Mediator

Memento

Observer

State

Strategy

Template Method

Visitor

### Builder

encapsulates the process of building a complex object

separates the representation of object & its construction

helps us to construct different representations of using same construction process

### Singleton

only one single instance of a class exist

global point of access to it exists

example: java.lang.runtime

1. eager initialization
2. static initialization
3. lazy initialization (not thread safe)

### Factory

encapsulate object creation code

provide an interface for object creation

but delegate the actual instantiation of objects to subclasses

max body parts to be created/called by parent class/interface

custom parts to be created by child class

### Abstract Factory pattern

create families of related products

define an interface to create families of related/dependent objects

(without specifying their concrete classes)

### Adapter

allow incompatible classes to work together by converting the interface of one class

into another (expected by the client)

Client

Adapter

Adaptee

Target

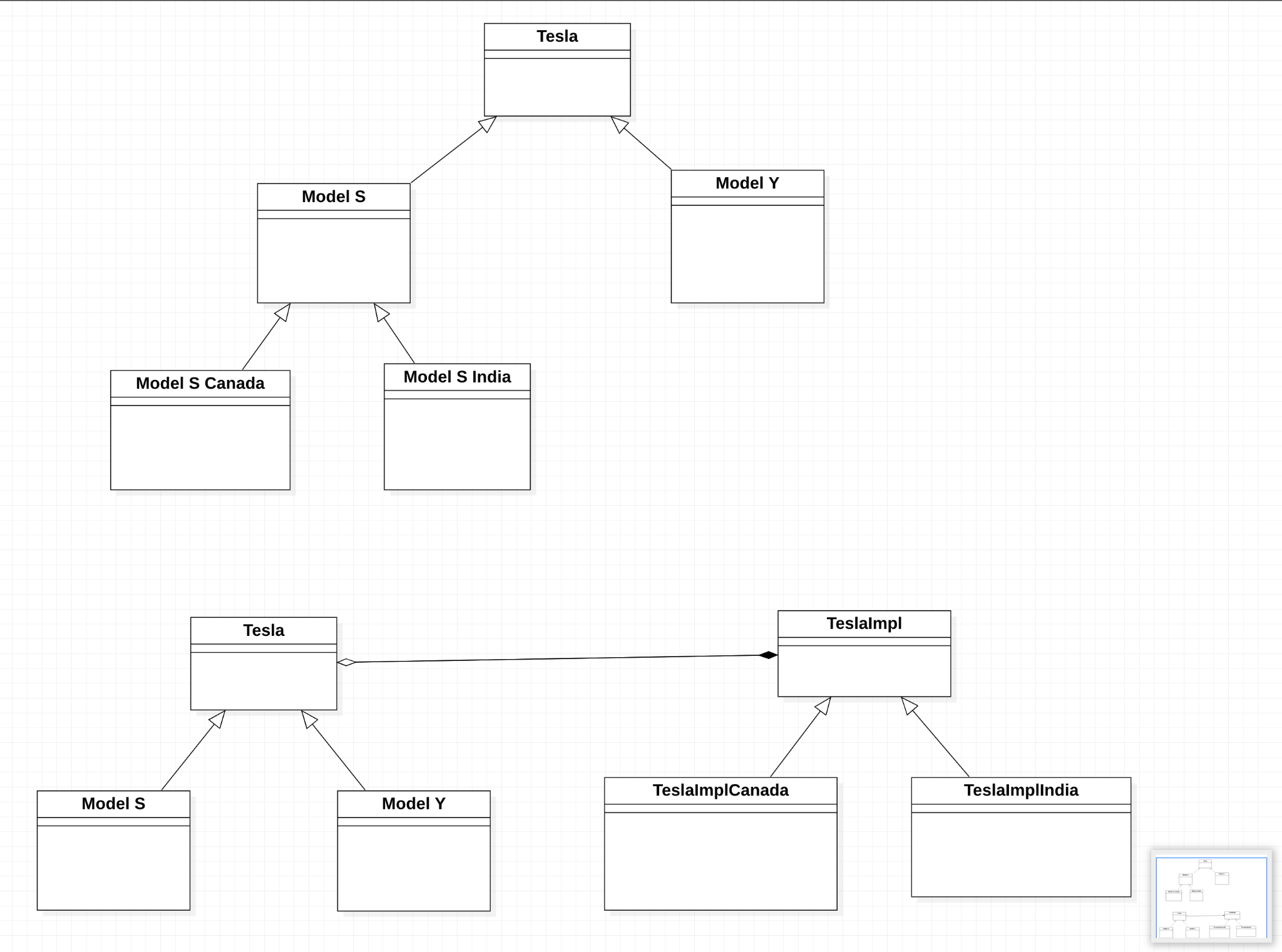
Object Adapter

Class Adapter

java.util.Arrays#asList()

### Bridge

parallel class hierarchy



### Composite

composing objects into tree-like structures

individual objects or compositions of objects (composites)

treating in the same way

component

base interface

leaf

implements default behaviour of base component

composite

will have leaf elements

define child-related operations

client

access the composites by using base component object

List

ArrayList

OtherList

### Decorator

enhanced interface to the original object

add functionalities to a product without disturbing the existing design

instead of subclassing for each decoration, we prefer composition

### Facade

single, simple interface to one or more subsystems

hide the complexity

### Flyweight

sharing state among a large number of objects

for memory efficiency

intrinsic state

extrinsic state

### Proxy

mechanism to provide a surrogate/placeholder/substitute

for another object to control access to it

### Chain of Responsibility

decoupling the sender of a request from it’s receiver

chaining all receiving objects

passing the request along the chain until an object handles it

### Observer

define a subscription mechanism

to notify about any/specific events to other objects

#### actors

observer

subject

### Command

encapsulate a request as an object

parameterize clients with different requests or queue or log requests

support undoable operations

### State

allow object to alter its behaviour when its internal state changes

the object will change its class internally

| current state | possible transitions to other states |
| --- | --- |
| parked | taxi |
| taxi | Airborne or Parked |
| airborne | Land or crash |
| land | Taxi |
| crash | no transition out of this state |

### General info

| Pattern | what is it | when to use |
| --- | --- | --- |
| Builder | construct complex objects | if there are several flavours of an Object  or when lot of steps are involved |
| Singleton | only a single instance of a class exists | when exactly one object is needed to coordinate all the operations/actions |
| Factory | interface for object creation, actual instantiation by subclasses | an object for creating other objects |
| Abstract Factory | factory of factories | system is independent of how its products are created/represented/used |
| Prototype | specify kind of objects & create a prototype, copy other objects from it |  |
| Adapter | allow incompatible classes to work together by converting interface of one class into another as expected | use a reusable class that cooperates with unrelated/unforseen classes |
|  |  |  |

## Problem Statement

1. A lot of concepts to revise
2. Practice coding
3. execute project

Solution:-

breadth of knowledge

depth (in parts)

1. solve brute force coding

simplify coding

1. visit each concept
2. implement project

java 40-50%

DS 35-45%

OOAD 5-10%

best practices

design patterns