Java and DSA

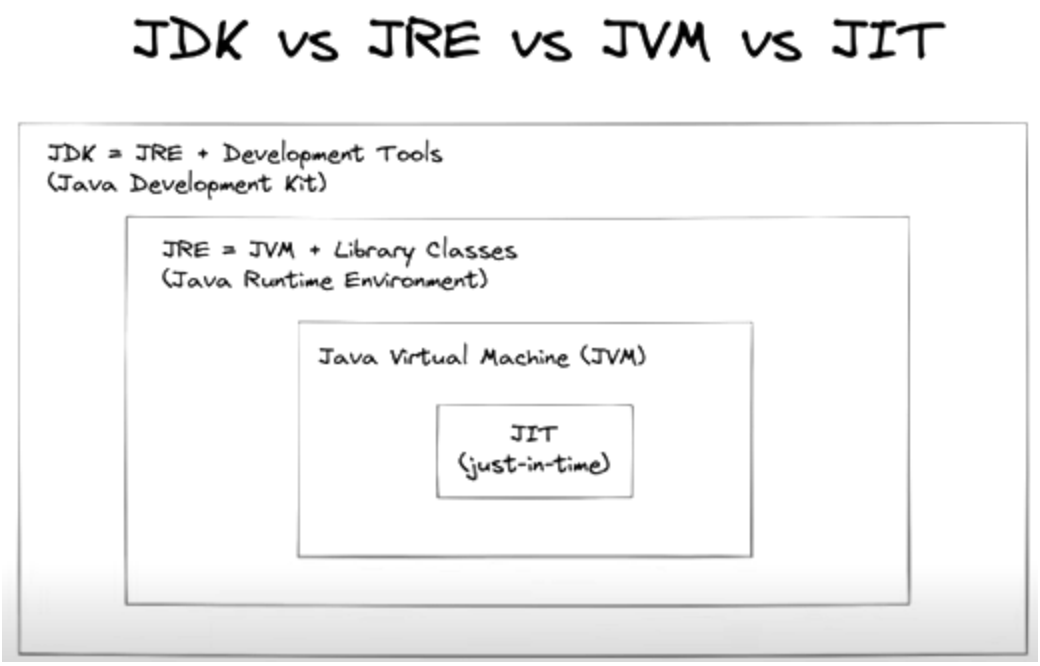
I .Java architecture

1.How Java code executes:

- Reason why Java is platform independent
The source code will not directly run on a system we need JVM to run this.

More about platform independence
- It means that byte code can run on all operating systems.
- We need to convert source code to machine code so computer can understand
- Compiler helps in doing this by turning it into executable code
- this executable code is a set of instructions for the computer
-After compiling C/C++ code we get .exe file which is platform dependent
-**In Java we get bytecode, JVM converts this to machine code**
- Java is platform-independent but JVM is platform dependent

2.JDK V/S JRE V/S JVM V/S JLT



2.1 JDK

- Provides environment to develop and run the Java program
- It is a package that includes:

1. development tools to provide an environment to develop your program
2. JRE to execute your program
3. a compiler - javac
4. archiver jar
5. docs generator - javadoc
6. interpreter/loader

2.2 JRE

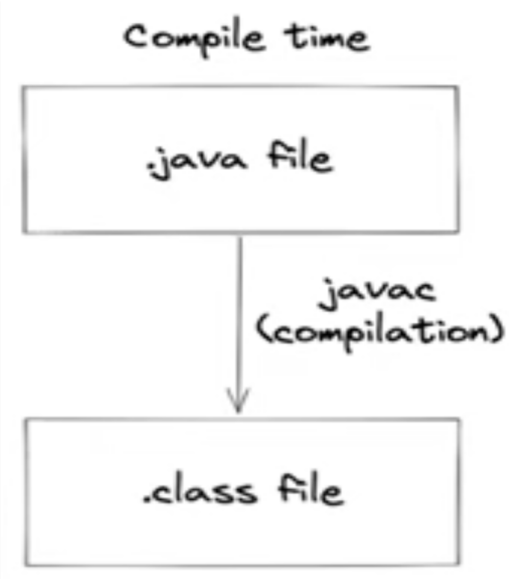
-It is an installation package that provides environment to only run the program
- It consists of:

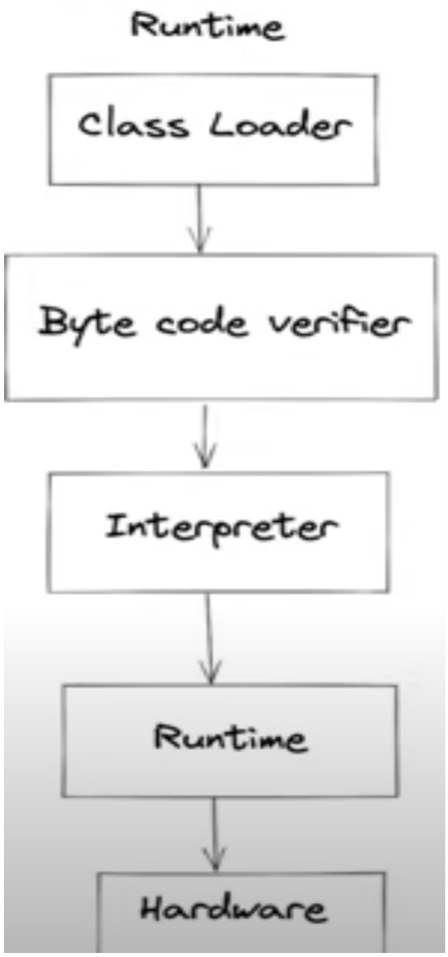
1. Deployment technologies
2. User interface toolkits
3. Integration libraries
4. Base libraries
5. JVM

- After we get the .class file, the next things happen at runtime:

1. Class loader loads all classes needed to execute the program.
2. JVM sends code to Bytecode verifier to check the format of code

2.3 Compile time V/S Runtime





2.4 How JVM works:

1.Class Loader: reads class file and generate binary data
- an object of this class is created in heap

2. Linking
- JVM verifies the class file
- allocates memory for class variables & default values
- replace symbolic references from the type with direct references
- Initialization: all static variables are assigned with their values defined in the code and static block
-JVM contains the Stack and Heap memory allocations.

3.JVM Execution

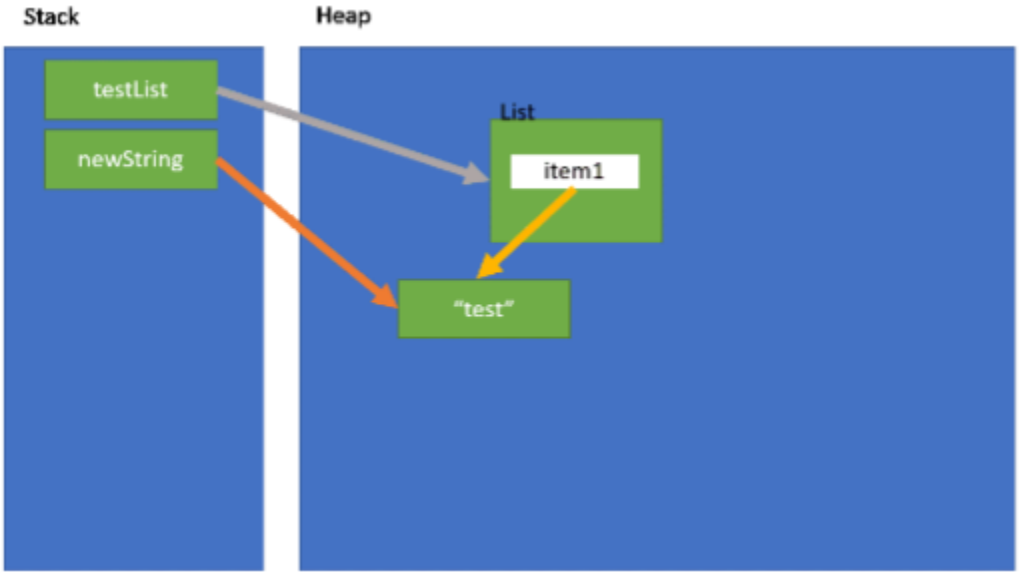
- **Interpreter:Line by line execution when one method is called many times,it will interpret again and again**
- **JIT: those methods that are repeated,JIT provides direct machine code so re-interpretation is not required.**
    - **makes execution faster**

II Memory Management

How Java Memory Works?

Stack v/s heap

```
public List<String> test() {
String newString = "test";
List<String> testList = new ArrayList<>();
testList.add(newString);
return testList;
}
```



Stack:

- Not only just one but also **every thread has its own stack region.**
- In stack, instantiated fields are added to memory one on another just like its name stacking.
- As you can see, this area is not big enough to store objects so what is happening is **primitive types and object pointers can be stored** directly instead of storing a whole object.

Heap:

- Heap size is bigger than stack because **heap is the main region for holding objects.** Every created object is held in heap and its reference is helding in stack.
- There is just **one heap that the application** has.
- That sounds quite reasonable because we might have more than one big object that passes one method to another.
- So basically, stack is used for local variables and there could be many stacks but they all use one heap to store the objects.
- Those pointers are held in stack so when we want to pass objects between methods, we are not copying objects like in stack, we are passing its reference.
- part:
    - Young Generation(Nursery): All the new objects are allocated in this memory. Whenever this memory gets filled, the garbage collection is performed. This is called the Minor Garbage Collection.
    - Old Generation: All the long lived objects which have survived many rounds of minor garbage collection is stored in this area. Whenever this memory gets filled, the garbage collection is performed. This is called as Major Garbage Collection.

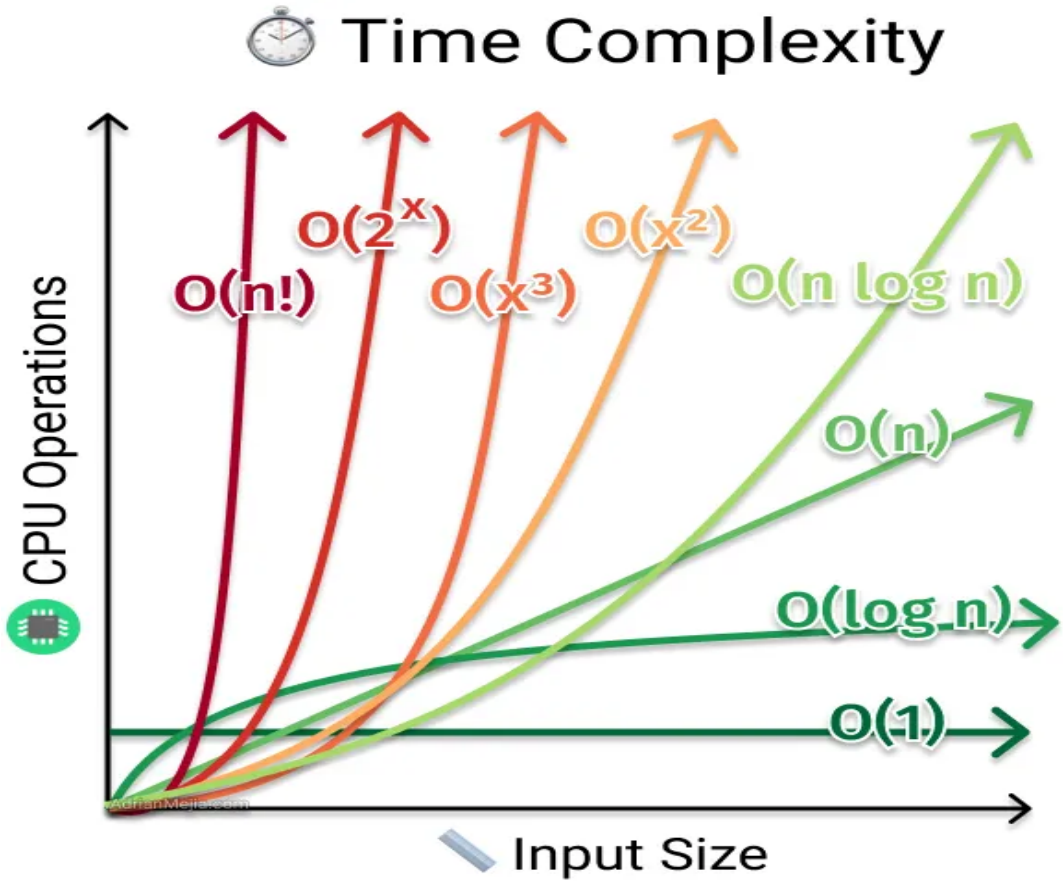
Metaspace v/s PermGen Memory

PermGen Memory:

- This is a special space in the java heap which is separated from the main memory where **all the static content is stored** in this section. Apart from that, this memory also stores the application metadata required by the JVM.
- Metadata is data which is used to describe the data. Here, garbage collection also happens like any other part of the memory.
- String pool was also part of this memory before Java 7. Method Area is a part of space in the PermGen and it is used to store the class structure and the code for methods and constructors. The biggest disadvantage of **PermGen is that it contains a limited size which leads to an**

Major types of complexities (time and space):

- Constant: O(1) - Excellent/Best
- O(log n) - Good
- Linear time O(n) - Fair
- Logarithmic time: O(n log n) - Bad
- Quadratic timeO(n^2), Exponential time O(2^n) and Factorial time O(n!) - Horrible/Worst



Sorting algorithm and their time and space complexity:

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst

Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Bucket Sort	Ω(n +k)	θ(n +k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n + k)
Count Sort	Ω(n +k)	θ(n +k)	O(n +k)	O(k)
Shell Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(1)
Tim Sort	Ω(n)	θ(n log(n))	O(n log (n))	O(n)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Cube Sort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

III Java Basics

1.Meaning of public static void main(String[] args):

1. **public**: The main method is **accessible from outside the class**.
2. **static**: The **main method belongs to the class itself rather than an instance of the class**. This allows it to be called without creating an object of the class.
3. **void: The main method does not return any value.**
4. **main**: The name of the method, which is recognized by the **JVM as the entry point.**
5. **String[] args:** The parameter passed to the main method. It is an array of strings that can be used to pass **command-line arguments to the program**. These arguments are useful when you want to provide some inputs to the program during execution.

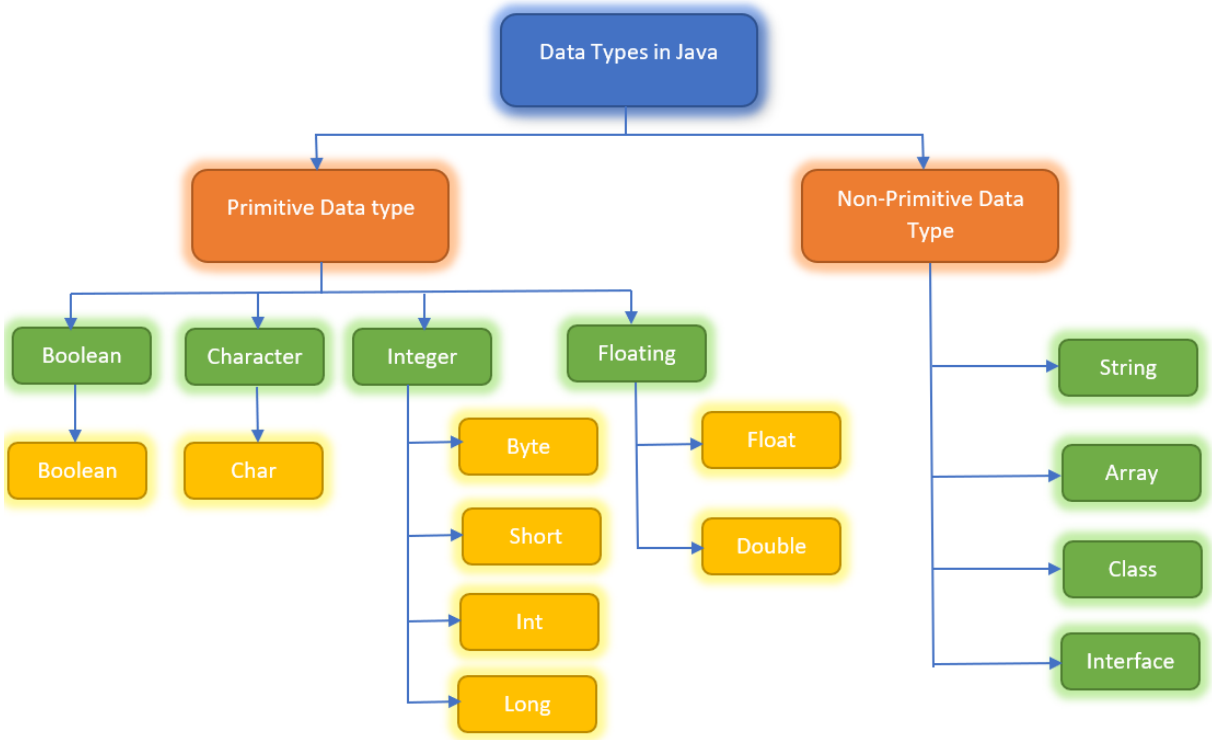
Example:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

2. Meaning of System.out.println():

- The **System class is a built-in class** in Java that provides access to system-related resources and methods.
- **The out field of the System class is an instance of java.io.PrintStream,** which represents the standard output stream.
- The **println() method is an overloaded method** that comes in different forms to handle different data types. It prints the argument(s) followed by a newline character (\n), which moves the cursor to the next line after printing the output.
- ***By default the System.out and System.in will be null, when it is null System.out will print in console and System.in will take the input from keyboard***

3. Data Types:



Type	Description	Default	Size	Example Literals	Range of values
boolean	true or false	false	1 bit	true, false	true, false
byte	twos-complement integer	0	8 bits	(none)	-128 to 127
char	Unicode character	\u0000	16 bits	‘a’, ‘\u0041’, ‘\101’, ‘\\’, ‘\’, ‘\n’, ‘β’	characters representation of ASCII values 0 to 255

short	twos-complement integer	0	16 bits	(none)	-32,768 to 32,767

int	twos-complement intger	0	32 bits	-2,-1,0,1,2	-2,147,483,648 to 2,147,483,647
long	twos-complement integer	0	64 bits	-2L,-1L,0L,1L,2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f , -1.23e-100f , .3f ,3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d , -123456e-300d , 1e1d	upto 16 decimal digits

4. Conditional Statement

If -else

5.Loops

1. for loop:

```
for(initialize;condition;increment){
  body
}
```

2. for each:

There is also a "for-each" loop, which is used exclusively to loop through elements in an array

```
for (type variableName : arrayName) {
}
```

3. while:

```
initialize;
while(condition){
  body;
  increment;
}
```

4. do while:

```
initialize;
do {
  // code block to be executed
  increment;
}
while (condition);
```

*The `break` statement can also be used to jump out of a loop.
*The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

6.Switch

6.1 **Old Switch**

Old Switch Syntax:
switch (expression) {
// cases
case one:
// do something
break;
case two:
// do something
break;
default:

```
// do something
}
```

- cases have to be the same type as expressions, must be a constant or literal
duplicate case values are not allowed
-break is use to terminate the sequence
-if break is not used, it will continue to next case
-default will execute when none of the above does
- if default is not at the end, put break after it

6.2 Enhanced switch

7.Pass-by-Value

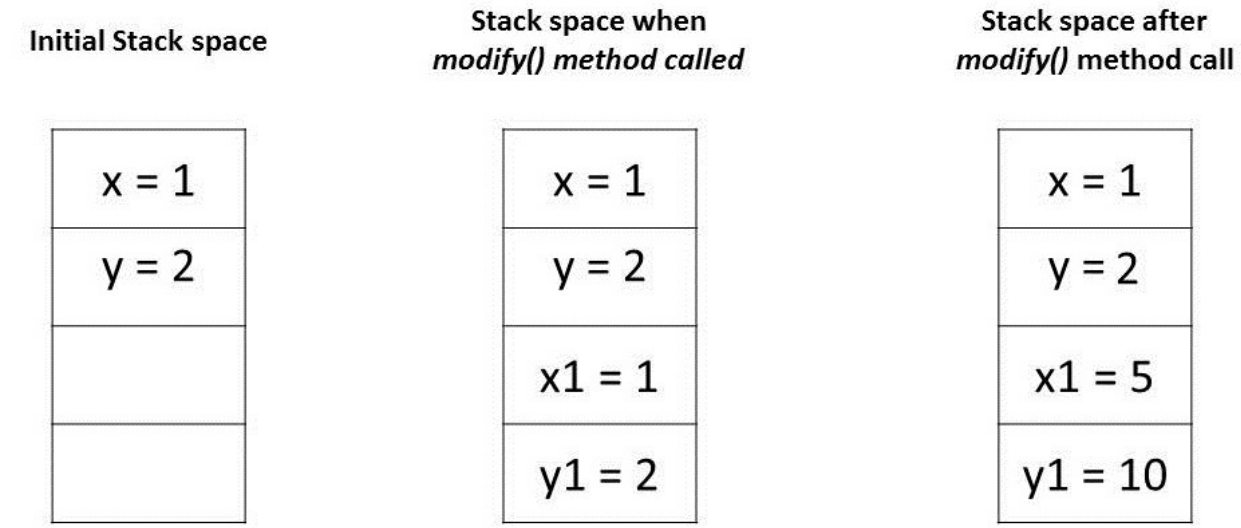
Refer: https://www.baeldung.com/java-pass-by-value-or-pass-by-reference

-When a parameter is pass-by-value, **the caller and the callee method operate on two different variables which are copies of each other.**
- Any changes to one variable don't modify the other.
-It means that while calling a method, parameters passed to the callee method will be clones of original parameters.
-Any modification done in the callee method will have no effect on the original parameters in the caller method.
-In Java, **Primitive variables store the actual values, whereas Non-Primitives store the reference variables which point to the addresses of the objects they're referring to.**
-Both values and references are stored in the stack memory.
-In case of primitives, the value is simply copied inside stack memory which is then passed to the callee method;
-In case of non-primitives, a reference in stack memory points to the actual data which resides in the heap. When we pass an object, the reference in stack memory is copied and the new reference is passed to the method.

7.1 Primitives pass by value:

```
public class PrimitivesUnitTest {

  @Test
  public void whenModifyingPrimitives_thenOriginalValuesNotModified() {

    int x = 1;
```

```
        int y = 2;

        // Before Modification
        assertEquals(x, 1);
        assertEquals(y, 2);

        modify(x, y);

        // After Modification
        assertEquals(x, 1);
        assertEquals(y, 2);
    }

    public static void modify(int x1, int y1) {
        x1 = 5;
        y1 = 10;
    }
}
```

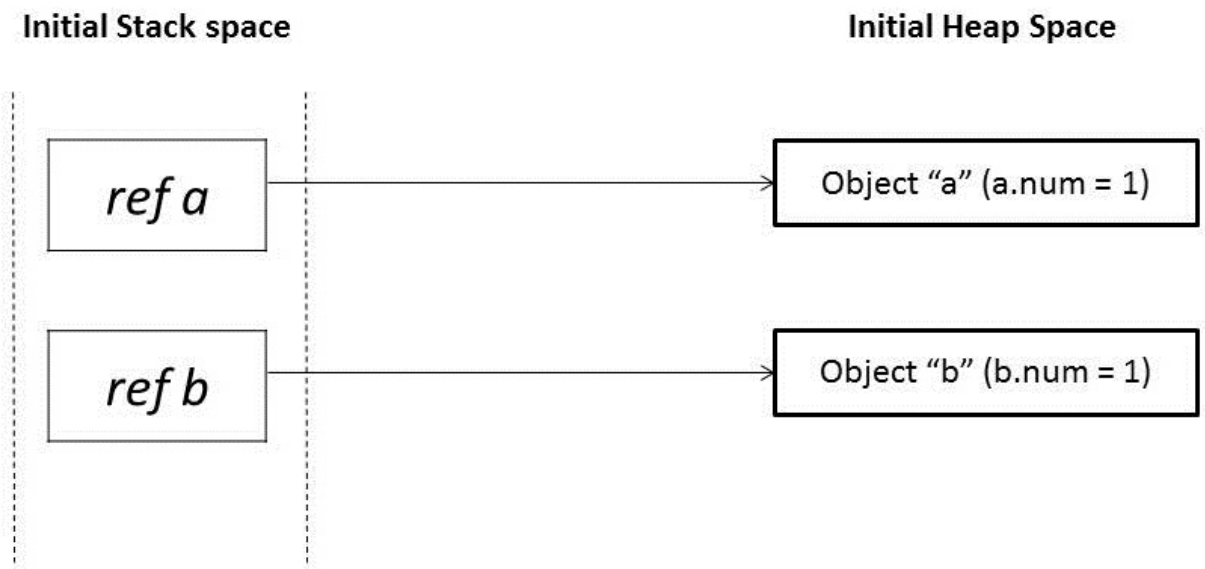


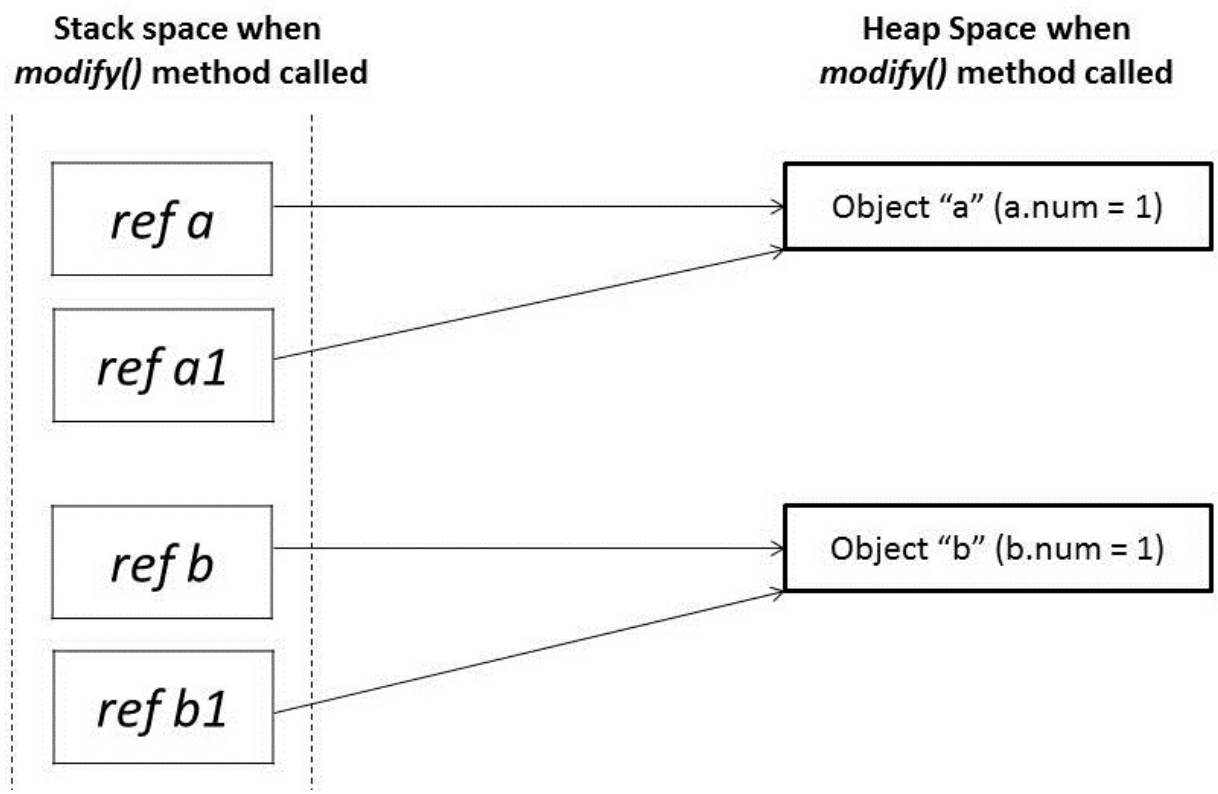
7.2 Passing Object References

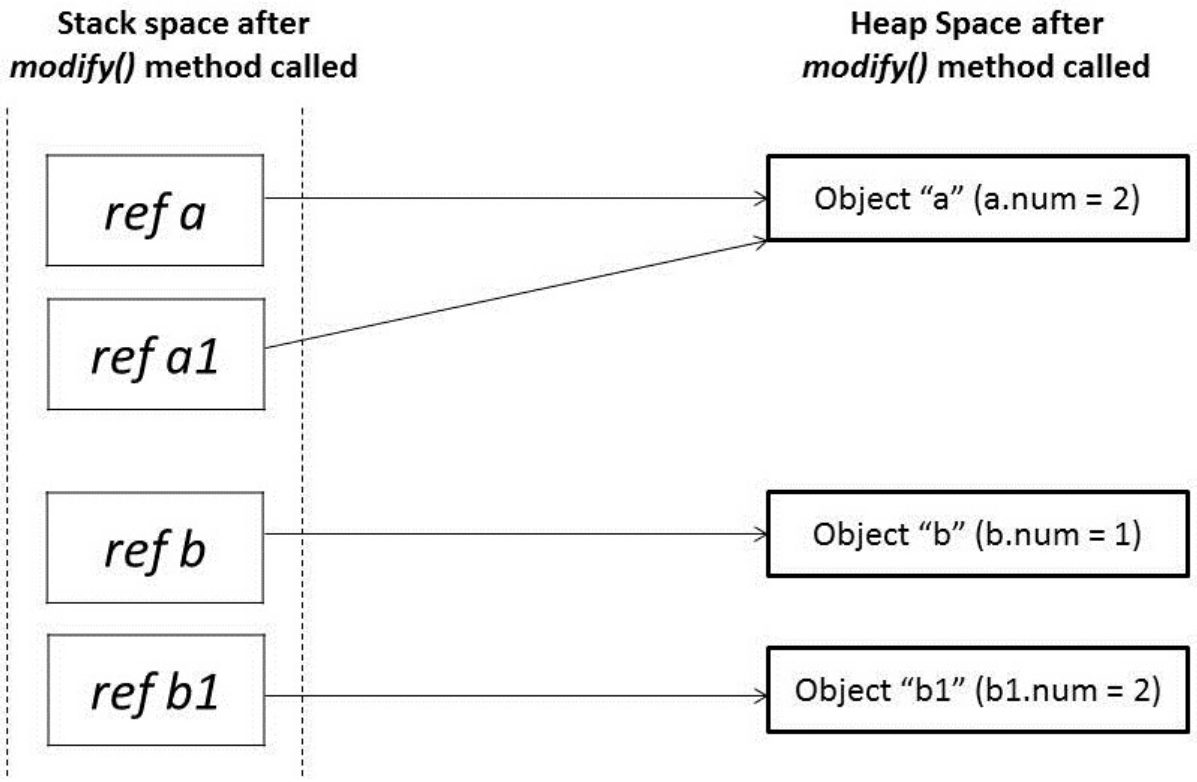
-In Java, all objects are dynamically stored in Heap space under the hood. These objects are referred from references called reference variables.
-A Java object, in contrast to Primitives, is stored in two stages.
-The reference variables are stored in stack memory and the object that they're referring to, are stored in a Heap memory.

-**Whenever an object is passed as an argument, an exact copy of the reference variable is created which points to the same location of the object in heap memory as the original reference variable.**
-As a result of this, whenever we make any change in the same object in the method, that change is reflected in the original object.
-However, if we allocate a new object to the passed reference variable, then it won't be reflected in the original object.

```
public class NonPrimitivesUnitTest {

    @Test
    public void whenModifyingObjects_thenOriginalObjectChanged() {
        Foo a = new Foo(1);
        Foo b = new Foo(1);

        // Before Modification
        assertEquals(a.num, 1);
        assertEquals(b.num, 1);

        modify(a, b);

        // After Modification
        assertEquals(a.num, 2);
        assertEquals(b.num, 1);
    }

    public static void modify(Foo a1, Foo b1) {
        a1.num++;

        b1 = new Foo(1);
        b1.num++;
    }
}

class Foo {
    public int num;

    public Foo(int num) {
        this.num = num;
    }
}
```


Initial Stack space
Initial Heap Space
ref a
Object “a” (a.num = 1)
ref b
Object “b” (b.num = 1)


Stack space when
modify() method called
Heap Space when
modify() method called
ref a
Object “a” (a.num = 1)
ref a1
ref b
Object “b” (b.num = 1)
ref b1



8.Array

To handle reversing the odd and even length array iterate till **array.length+1**

2D array

ArrayList

9.Linear search:

A linear search is a simple searching algorithm that sequentially checks each element in a list or array until a match is found or the entire list has been searched. Here's the algorithm for a linear search along with its pseudocode:

Algorithm: Linear Search

Start at the beginning of the list.

For each element in the list:
a. Compare the current element with the target element.
b. If they match, return the current element's index (or position).
c. If they don't match, move to the next element.
If the end of the list is reached without finding the target element, return a special value (e.g., -1) to indicate that the element was not found.

Pseudocode:

```
function linearSearch(list, target):
    for i from 0 to length(list) - 1:
        if list[i] equals target:
            return i  // Element found, return its index
    return -1  // Element not found
```

Time Complexity:

Best case: O(1)

- when the target element is in the beginning of the array.

Worst case: O(N)

- where N is the size of array
- when the target element is at the end.

To get number of digit : (int)(Math.log10(num))+1

10. Binary Search

Binary search is a highly efficient algorithm for finding a specific target element in a sorted array or list.

Algorithm: Binary Search

- Initialize two pointers, left and right, to the start and end of the sorted list, respectively.
- Calculate the middle index as (left + right) / 2.
- Compare the middle element with the target element.
    - a. If the middle element equals the target, return the middle index (element found).
    - b. If the middle element is less than the target, set left to middle + 1, and repeat step 2.
    - c. If the middle element is greater than the target, set right to middle - 1, and repeat step 2.

- Continue this process until left is greater than right, indicating that the target element is not in the list. Return a special value (e.g., -1) to indicate that the element was not found.

Pseudocode:

```
{
 function binarySearch(sortedList, target):
    left = 0
    right = length(sortedList) - 1
    while left <= right:
        middle = (left + right) / 2
        if sortedList[middle] equals target:
            return middle  // Element found, return its index
        else if sortedList[middle] < target:
            left = middle + 1
        else:
            right = middle - 1
    return -1  // Element not found
}
```

- Best case: O(1)
    - when the target element is in the middle of the array.

-Worst case: log(N)
    - where N is the size of array
    - when the target element is at the end.

- mid = (start+end)/2 this may exceeds the int value;
- so use this:
        -mid = start+(end-start)/2

-How to check whether the it is desc or asc order sorting
Check the 1st and last element, this will even work if there is a sequential element like [99,99,99,88,77,66]

-Most time it is applied to sorted array

11.Bubble / sinking / exchange sort:

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, indicating that the list is sorted.

Algorithm: Bubble Sort

- Start at the beginning of the list.
- Compare the first two elements.
    - a. If the first element is greater than the second element, swap them.
- Move to the next pair of elements and repeat step 2 until the end of the list is reached.
- After the first pass, the largest element will have "bubbled up" to the end of the list.
- Repeat steps 1-4 for the remaining unsorted portion of the list (i.e., excluding the last element).
- Continue this process until no swaps are made during a pass, indicating that the list is sorted.

Pseudocode:

```
function bubbleSort(arr):
    n = length(arr)
    swapped = true

    while swapped:
        swapped = false
        for i from 0 to n - 2:
            if arr[i] > arr[i + 1]:
                // Swap the elements
                swap(arr[i], arr[i + 1])
                swapped = true

// Swap function to exchange two elements
function swap(a, b):
    temp = a
    a = b
    b = temp
```

-Space complexity: O(1)
- also known has in place sorting

- Best case: O(N)
    - when the array is sorted.

-Worst case: O(N^2)
- when sorted in opposite order

-it is unstable sorting algorithm
- because when 2 element have same value original order is not maintained

12.Selection sort:

-Space complexity: O(1)
- also known has in place sorting

- Best case: O(N)
  - when the array is sorted.

-Worst case: O(N^2)
  - when sorted in opposite order

-it is unstable sorting algorithm
- because when 2 element have same value original order is not maintained

13.Insertion sort:

-Space complexity: O(1)
- also known has in place sorting

- Best case: O(N)
  - when the array is sorted.

-Worst case: O(N^2)
  - when sorted in opposite order

14.Cyclic Sort

When a range of numbers from [0,N] or [1,N] is given, use cyclic sort.

- ***If starts from 0, index = value***
- ***If starts from 1, index = value -1***

15. String and String buffer

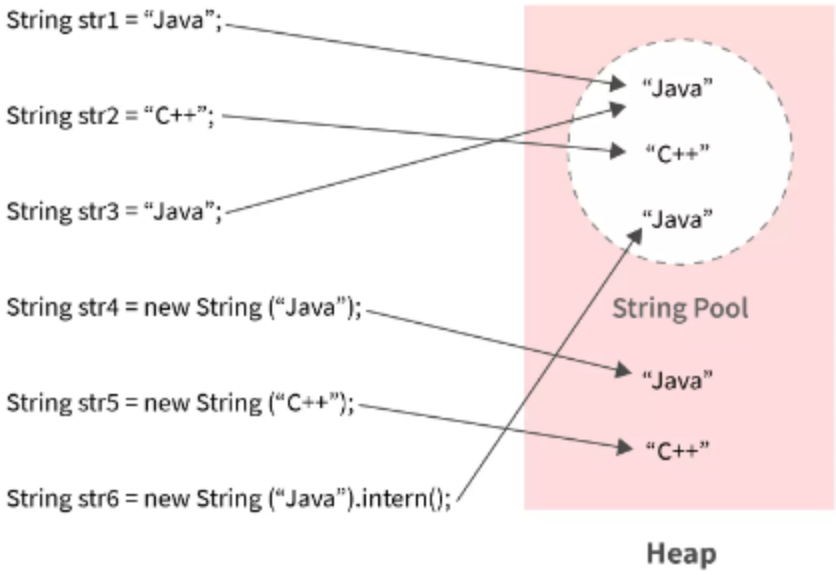
Uppercase Letters − A - Z having ASCII values from 65 - 90 where 65 and 90 are inclusive.
Lowercase Letter − a - z having ASCII values from 97 - 122 where 97 and 122 are inclusive.
Numeric values − 0 - 9 having ASCII values from 48 - 57 where 48 and 57 are inclusive.

String pool

- **String pool** is nothing but a storage area in Java heap where string literals store.
- It is also known as **String Intern Pool** or **String Constant Pool**. It is just like object allocation.
- By default, it is empty and privately maintained by the **Java String** class.
- Whenever we create a string the string object occupies some space in the heap memory. Creating a number of strings may increase the cost and memory too which may reduce the performance also.
- The JVM performs some steps during the initialization of string literals that increase the performance and decrease the memory load.
- To decrease the number of String objects created in the JVM the String class keeps a pool of strings.
- When we create a string literal, the JVM first checks that literal in the String pool.
- If the literal is already present in the pool, it returns a reference to the pooled instance.
- If the literal is not present in the pool, a new String object takes place in the String pool.
- String pool is an implementation of the String Interring Concept.

String intern

- The **String.intern()** method puts the string in the String pool or refers to another String object from the string pool having the same value.
- It returns a string from the pool if the string pool already contains a string equal to the String object.
- It determines the string by using the **String.equals(Object)** method.
- If the string is not already existing, the String object is added to the pool, and a reference to this String object is returned.



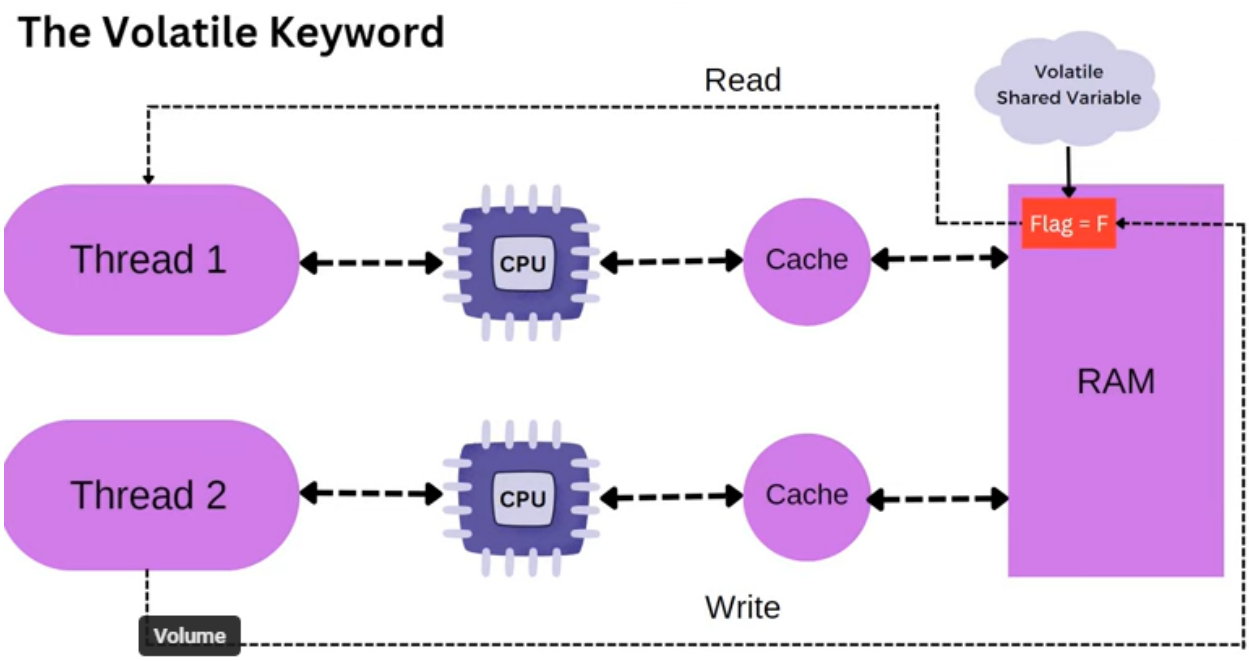

```
String a = "Hello"; //a(reference) will be created in stack memory and "Hello"
(object) is created in String pool(a place within heap memory)

String b = "Hello"; //b(reference) will be created in stack memory and it will
point to same "Hello" (object) in String pool

System.out.println(a==b); //true because both a and b are pointing to same object
in string pool
*****************************************

String c = new String("World"); //c(reference) will be in stack memory and "Hello"
(object) is created outside String pool,in heap memory

String d = new String("World"); //d(reference) will be in stack memory and "Hello"
(object) is created outside String pool,in heap memory

System.out.println(c == d); //false because both c and d are pointing to different
object in heap

System.out.println(c.equals(d));//true because both c and d has same value
*************************************************

String e = new String("Hello").intern();//e(reference) will be created in stack
memory and it will point to same "Hello" (object) in String pool

System.out.println(e==a);//true
```

String concatenation

Space Complexity: O(N^2)

String builder v/s Stringbuffer

- String is immutable whereas StringBuffer and StringBuilder are mutable classes.
- StringBuffer is thread-safe and synchronized whereas StringBuilder is not.
- That's why StringBuilder is faster than StringBuffer
- StringBuilder and buffer does not override Object's equals method



Few IMP reference:

- https://medium.com/gitconnected/dont-just-leetcode-follow-the-coding-patterns-instead-4beb6a197fdb

IV Math for DSA

1.Bitwise operator

AND Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

NOT Truth Tabl

A	B
0	1
1	0

AND:

- Anything you and with 1, result will be a same

OR:

- Anything you or with 1, result will be true
- Any number xored with 1, result is opposite of that.

XOR:

- Any number xored with 0, result is the same as that.
- Any number xored with the same, result 0.

Base Conversion:

1.Base 10 to any base: divide
2.Any base to base 10: multiply and add the power of base with digit

Left shift:

- a<<1 = 2a (double the number)
- a<<b = 2^b * a

Right shift:

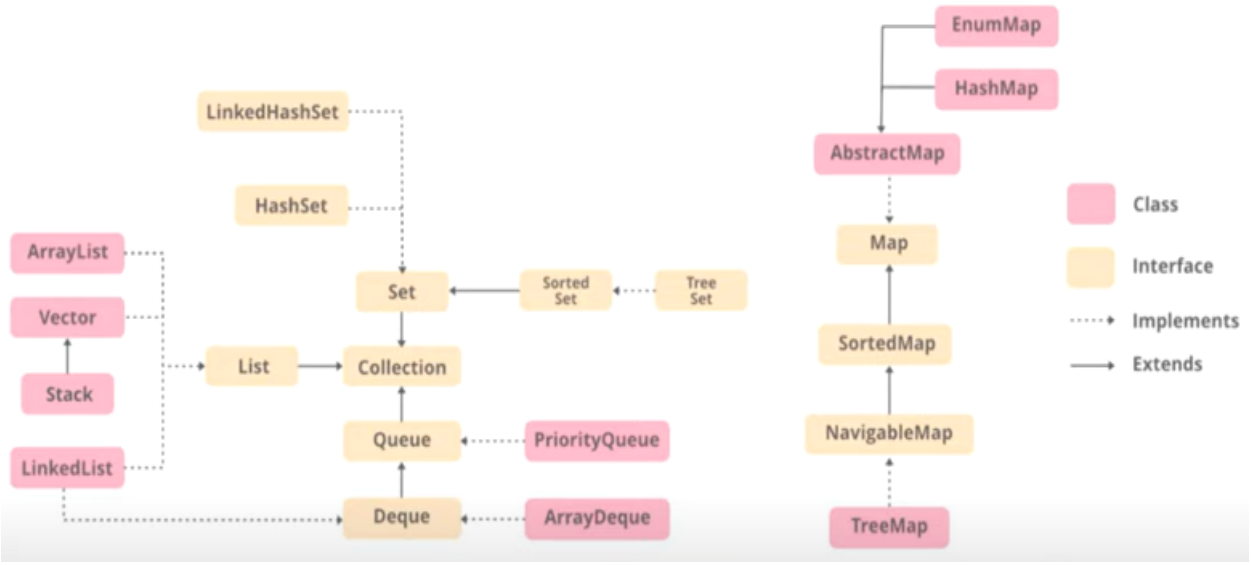
- a>>b= a/2^b
- In Java, the >>> operator is known as the "unsigned right shift" operator. It's a bitwise operator that shifts the bits of a binary number to the right by a specified number of positions, filling the leftmost positions with zeros. **Unlike the >> operator, which**

performs a signed right shift (fills the leftmost positions with the sign bit), the >>> operator always fills with zeros.

- **Ex:**
  - **1101 >>1 = 1110**
  - **1101>>>1 = 0110**

Important tip:

- To get complement of number:
  - Subtract the number from 1
- To get last bit
  - And a number with 1
- To check 2 number are same
  - Xor it
- To get nth bit
  - Number & (1<<(n-1))
  - Or (Number>>n)&1
- To get a range of number:
  - $-\ 2^{n-1}\ to\ (\ 2^{n-1}\ -\ 1\ )$
  - N-1 in power because the last 1 bit (MSB)is used to indicate whether a number is positive or negative.
    - 0: +ve
    - 1: -ve
  - 1 is deducted from +ve end because of 0
- To get the number of digit:
  - $(int)(log(num)/log(base))\ +\ 1$
- Pascal triangle
  - Sum of any row (2^n):
    - 1<<(n-1)
- To check whether a number is power of 2
  - Number & (Number-1) = 0 then it is a power of 2 else not
  - Number is a power of 2 when its binary representation has only one 1.
- To get MSB:
  - n&-n



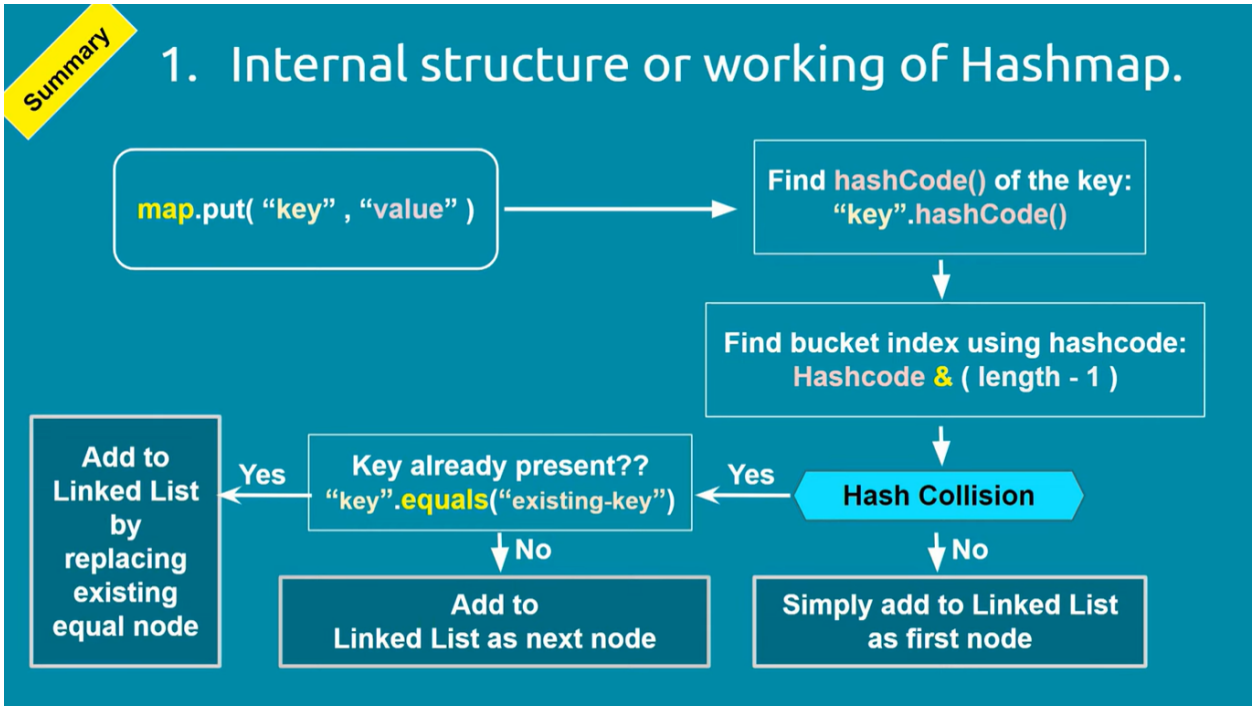
V Collection FrameWork

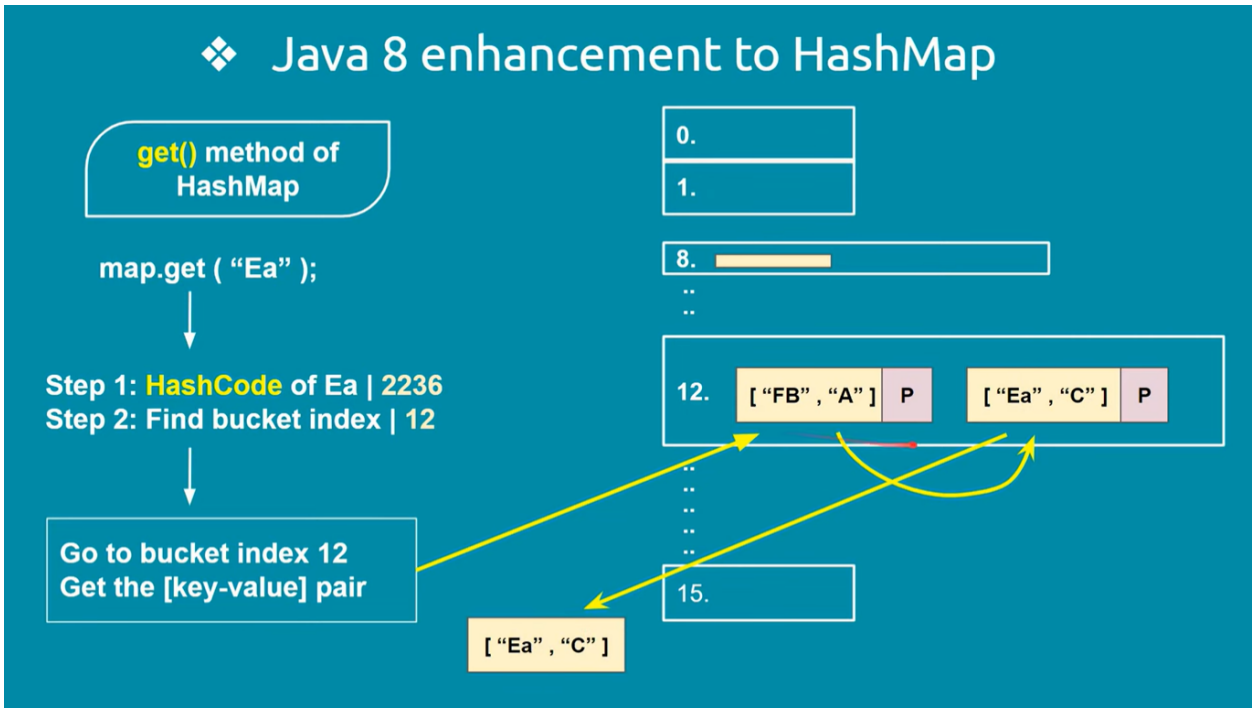
Collection framework:

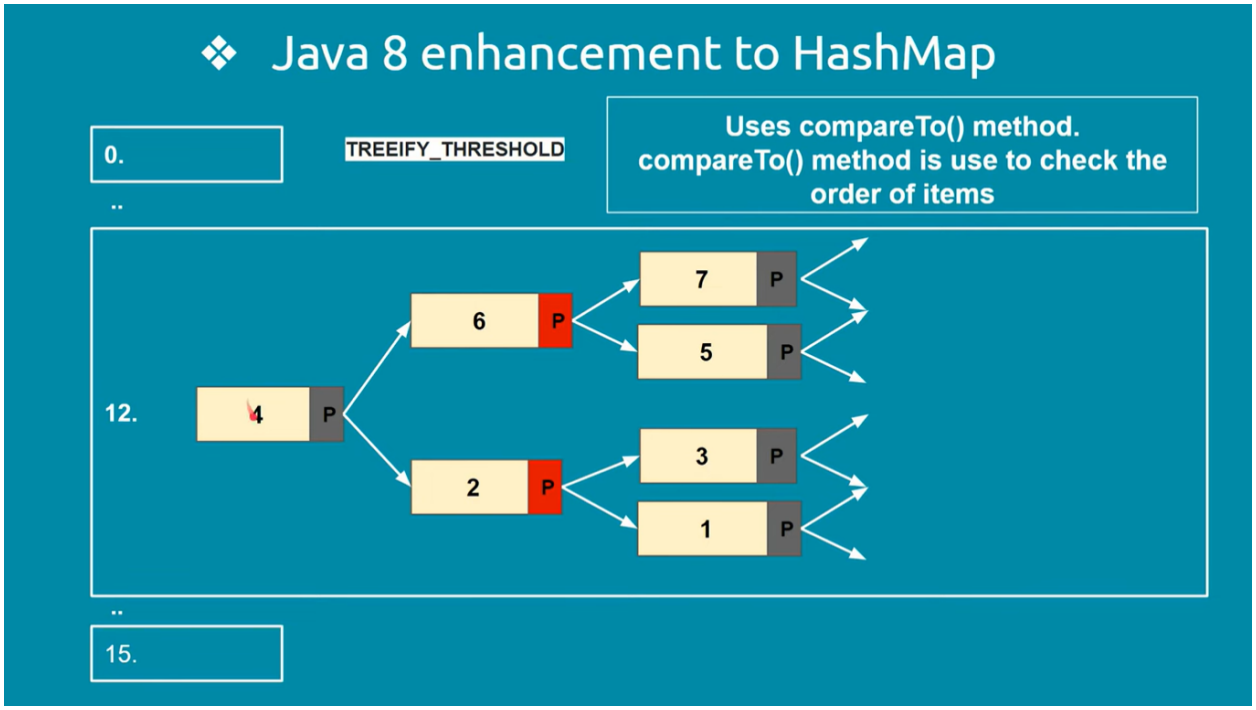
Array v/s collections

Collection v/s collections
List v/s set v/s queue v/s map

Hash map:


Summary
1. Internal structure or working of Hashmap.
map.put(“key” , “value”)
Find hashCode() of the key:
“key”.hashCode()
Find bucket index using hashcode:
Hashcode & (length - 1)
Hash Collision
Yes
Key already present??
“key”.equals(“existing-key”)
Yes
Add to
Linked List
by
replacing
existing
equal node
No
Add to
Linked List as next node
No
Simply add to Linked List
as first node


❖ Java 8 enhancement to HashMap
get() method of
HashMap
map.get (“Ea”);
Step 1: HashCode of Ea | 2236
Step 2: Find bucket index | 12
Go to bucket index 12
Get the [key-value] pair
0.
1.
8.
12.
[“FB” , “A”]
P
[“Ea” , “C”]
P
15.
[“Ea” , “C”]



Tree

Each node in the tree has a value and zero or more child nodes, which are typically represented as branches. The topmost node in the tree is called the root, and nodes with no children are called leaves.

Basic Tree Terminology:

1. Node: Each element in the tree is a node. Nodes are the building blocks of a tree and contain data.
2. Root: The topmost node in the tree is the root. It is the starting point for traversing the tree.
3. Parent: A node in the tree that has one or more child nodes.
4. Child: Nodes that have a common parent are considered children of that parent.
5. Leaf: Nodes that have no children are called leaves. They are the endpoints of the tree.
6. Subtree: A tree formed by a node and its descendants.
7. Depth: The level of a node in the tree. The root is at depth 0, and each level below increases the depth.(**Height of root- height of node)**
8. Height: The length of the longest path **from a node to a leaf. The height of the tree is the height of the root.**

Types of Tree:

Binary Tree:

- A tree in which each node has **at most two children**, referred to as the left child and the right child.
- It is a basic form of a tree and is used in various applications.

Binary Search Tree (BST):

- **A binary tree with the property that the value of each node is greater than or equal to the values in its left subtree and less than or equal to the values in its right subtree.**
- Enables efficient searching, insertion, and deletion operations.

AVL Tree:

- A self-balancing binary search tree.
- It maintains a balance factor for each node, ensuring that the height difference between the left and right subtrees is limited to a small constant.

Red-Black Tree:

- Another self-balancing binary search tree.
- Nodes are colored either red or black, and there are rules to ensure the tree remains balanced during insertions and deletions.

B-Tree:

- A self-balancing tree structure that maintains sorted data and allows searches, insertions, and deletions in logarithmic time.
- Widely used in databases and file systems.

Trie (Prefix Tree):

- A tree-like structure that is used for storing a dynamic set of strings.
- Particularly efficient for searches involving strings and is used in autocomplete systems.

Segment Tree:

- A tree data structure used for storing information about intervals or segments.
- Commonly used in applications involving range queries, such as finding the sum or minimum value in a given range.

Quadtree:

- A tree data structure in which each internal node has exactly four children.
- Often used in computer graphics and spatial indexing for organizing spatial data.

Octree:

- A three-dimensional extension of a quadtree, where each internal node has exactly eight children.
- Used in 3D computer graphics and spatial indexing.

Heap:

- A specialized tree-based data structure that satisfies the heap property.
- Commonly used to implement priority queues.

Expression Tree:

- A binary tree representation used to represent expressions.
- Each leaf represents an operand, and each internal node represents an operator.

Binary trees based on their structural properties:

1. Full Binary Tree:

- Every node in the tree has either 0 or 2 children.
- No node has only one child.
- Also known as a proper or plane binary tree.

2. Complete Binary Tree:

- All levels of the tree are completely filled, except possibly for the last level, which is filled from left to right.
- A complete binary tree is efficient for storage in an array.

3. Perfect Binary Tree:

- A binary tree in which all the internal nodes have exactly two children, and all leaf nodes are at the same level.
- The number of nodes in a perfect binary tree is $2^{(h+1)} - 1$ where h is the height.

4. Balanced Binary Tree:

- A binary tree is balanced if the height of the left and right subtrees of any node differ by at most one.
- Balancing ensures that the tree remains relatively flat, leading to more efficient operations.

5. Degenerate (or pathological) Tree:

- A binary tree where each parent node has only one associated child node.
- Essentially, it is a linked list in tree form.

6. Skewed Binary Tree:

- A special case of a degenerate tree where the tree is either entirely left-skewed or right-skewed.
- The height of a skewed tree is
- $\text{\textdiamond}-1$
- $n-1$, where
- \textdiamond
- n is the number of nodes.

7. Threaded Binary Tree:

- A binary tree in which some of the null pointers are used to store information about the next (in-order) or previous (reverse in-order) node in the sequence.

Tree Traversal:

Traversal is the process of visiting all the nodes in the tree and performing an operation at each node.

Depth-first search

Depth-first search is a type of traversal that goes **deep as much as possible in every child before exploring the next sibling.**

The three common types of tree traversal are:

Inorder Traversal: Traverse the left subtree, visit the **root**, traverse the right subtree.

Preorder Traversal: Visit the **root**, traverse the left subtree, traverse the right subtree.

Postorder Traversal: Traverse the left subtree, traverse the right subtree, visit the **root**.

BSF

AVL Tree:

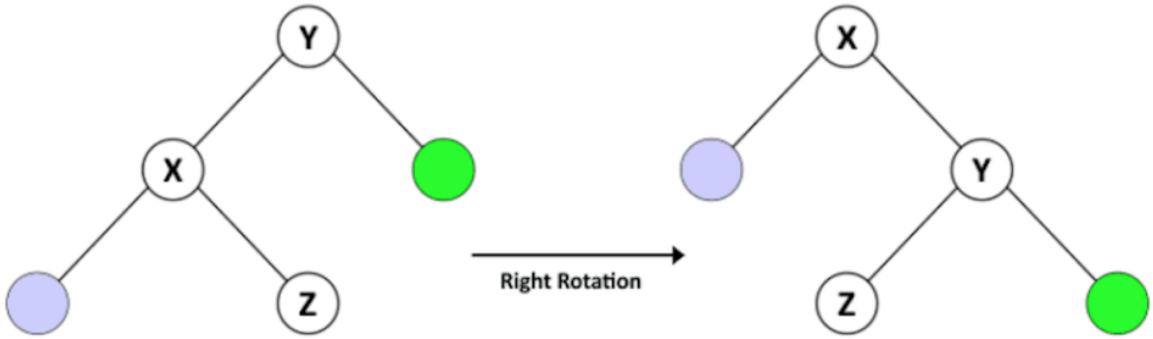
- The AVL Tree, named after its inventors Adelson-Velsky and Landis**, is a self-balancing binary search tree (BST).**
- **A self-balancing tree is a binary search tree that balances the height after insertion and deletion according to some balancing rules.**
- The worst-case time complexity of a BST is a function of the height of the tree. Specifically, the longest path from the root of the tree to a node. For a BST with N nodes, let's say that every node has only zero or one child. Therefore its height equals N, and the search time in the worst case is O(N). So our main goal in a BST is to keep the maximum height close to log(N).
- **The balance factor of node N is height(right(N)) – height(left(N)). In an AVL Tree, the balance factor of a node could be only one of 1, 0, or -1 values.**

How to Balance an AVL

Checks the balance factor of its nodes after the insertion or deletion of a node. If the balance factor of a node is **greater than one or less than -1,** the tree rebalances itself.
There are two operations to rebalance a tree:

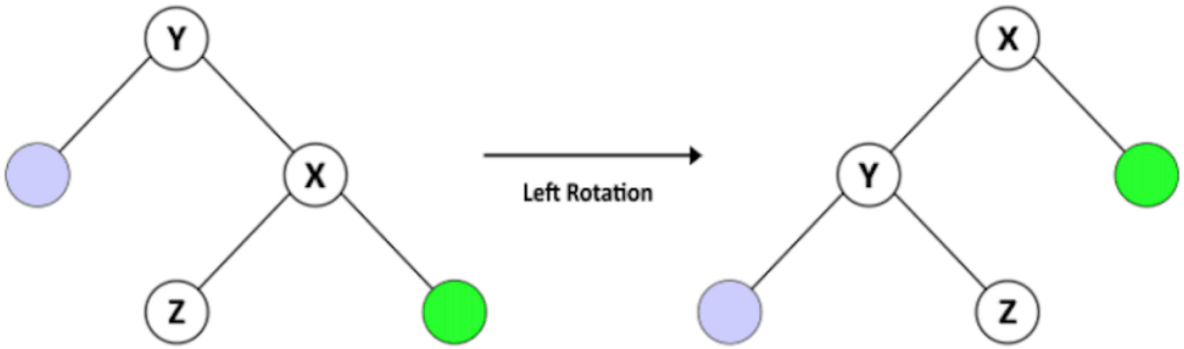
- right rotation and
- left rotation.

Right rotation



- Given the characteristics of a BST, we know that X < Z < Y.
- After a right rotation of Y, we have a tree called T2 with X as the root and Y as the right child of X and Z as the left child of Y. T2 is still a BST because it keeps the order X < Z < Y.

Left Rotation



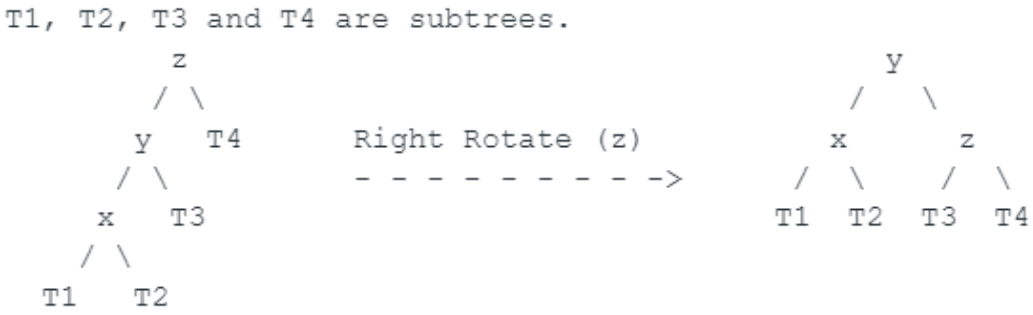
- Given this, we know that Y < Z < X.
- After a left rotation of Y, we have a tree called T2 with X as the root and Y as the left child of X and Z as the right child of Y. T2 is still a BST because it keeps the order Y < Z < X.

AVL Steps to follow for insertion:

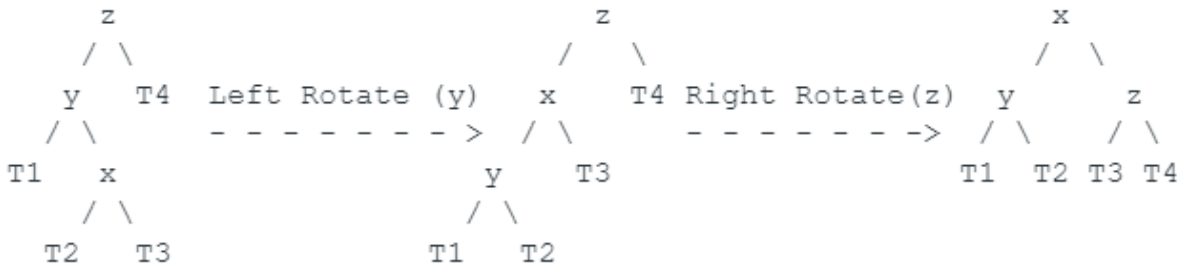
- Let the newly inserted node be w
- Perform standard BST insert for w.
- **Starting from w, travel up and find the first unbalanced node.**
- Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that need to be handled as x, y and z can be arranged in 4 ways.

Following are the possible 4 arrangements:
y is the left child of z and x is the left child of y (Left Left Case)
y is the left child of z and x is the right child of y (Left Right Case)
y is the right child of z and x is the right child of y (Right Right Case)
y is the right child of z and x is the left child of y (Right Left Case)

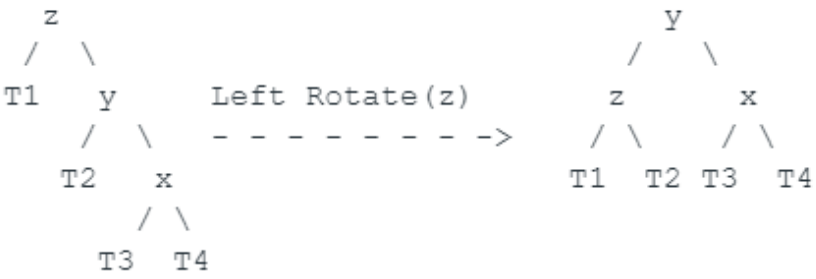
1. Left Left Case



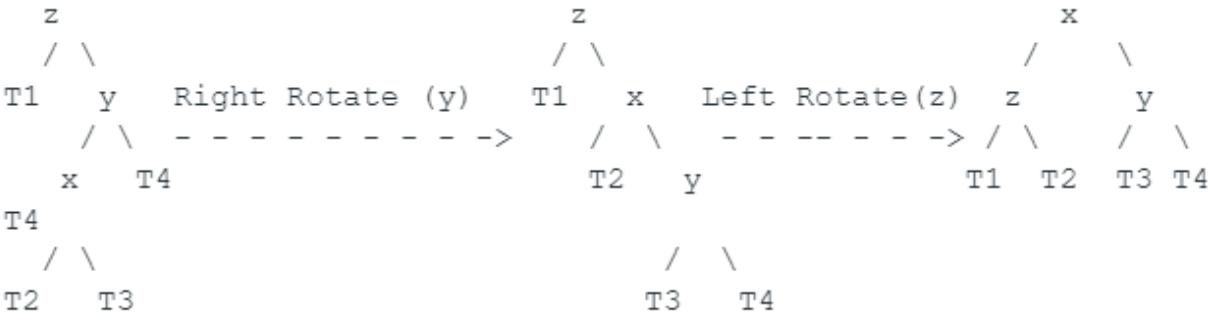
2. Left Right Case



3. Right Right Case



4. Right Left Case



VI Design Patterns

https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples