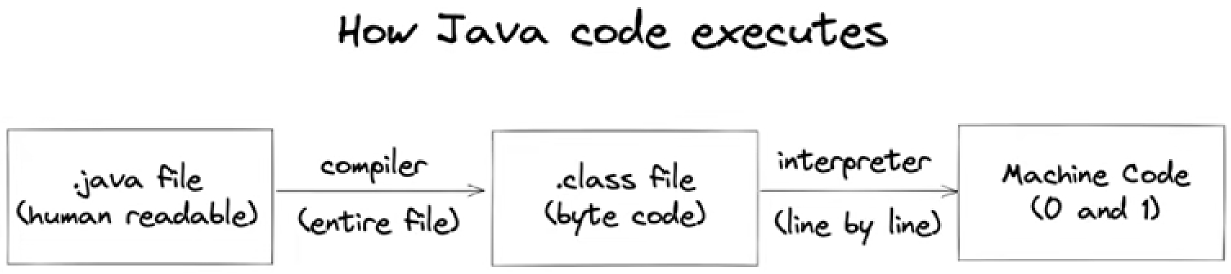
Java and DSA

I .Java architecture

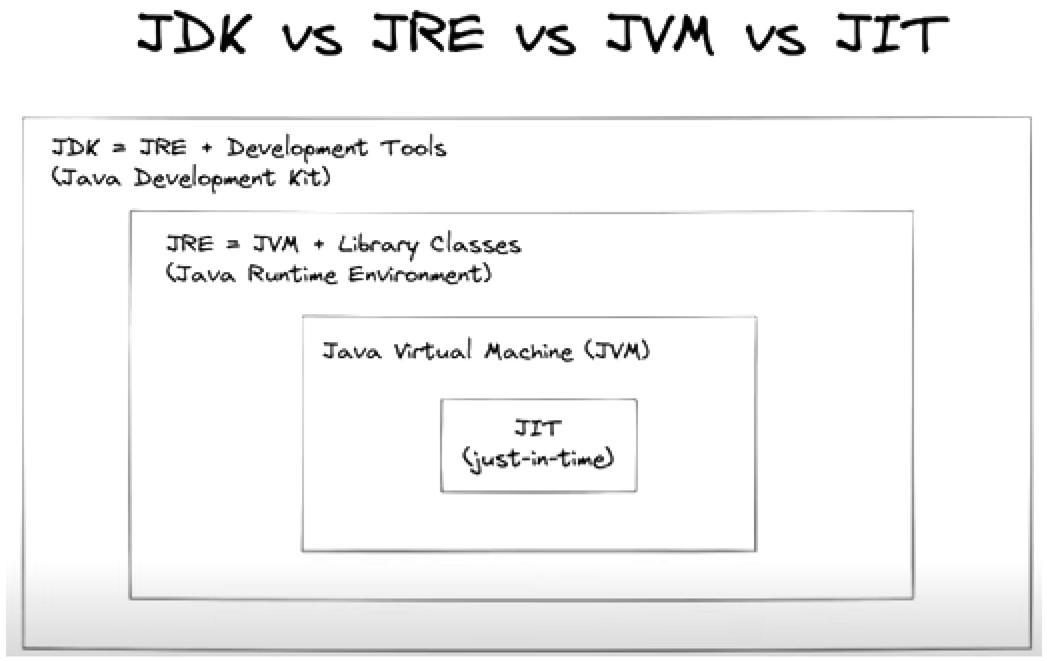
1.How Java code executes:



- 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


JDK vs JRE vs JVM vs JIT
JDK = JRE + Development Tools
(Java Development Kit)
JRE = JVM + Library Classes
(Java Runtime Environment)
Java Virtual Machine (JVM)
JIT
(just-in-time)

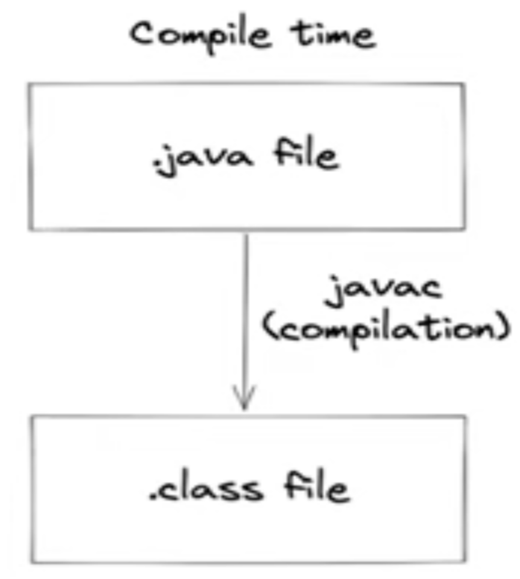
2.1 JDK

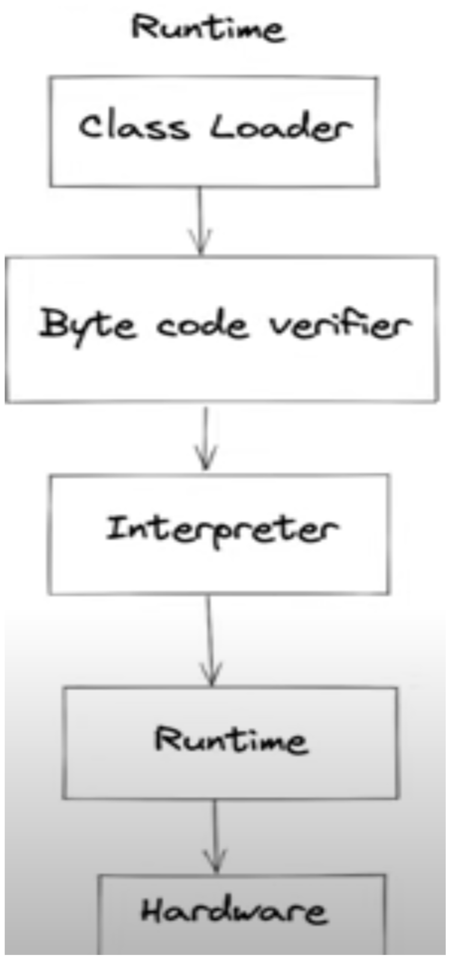
- Provides environment to develop and run the Java program
- It is a package that includes:
1. development tools to provide an environment to develop your program
2. JRE to execute your program
3. a compiler - javac
4. archiver jar
5. docs generator - javadoc
6. interpreter/loader

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

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:

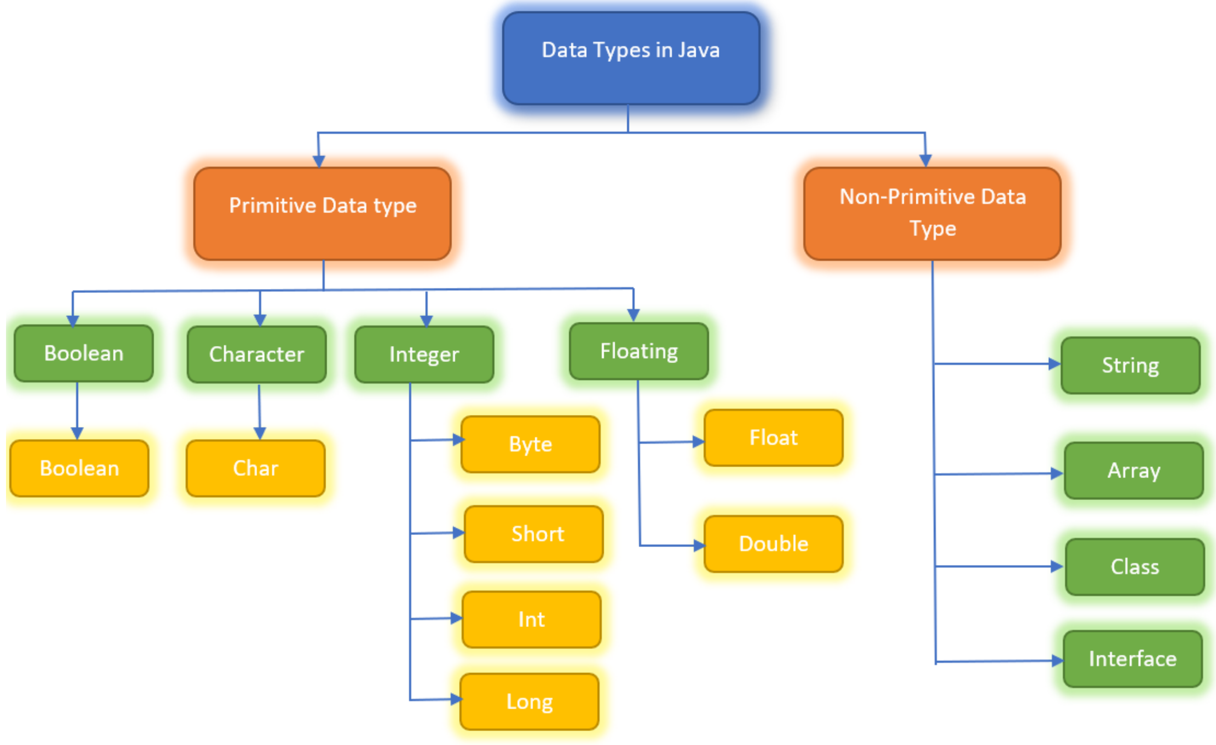
```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

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-compleme nt 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-compleme nt integer	0	16 bits	(none)	-32,768 to 32,767
int	twos-comple ment intger	0	32 bits	-2,-1,0,1,2	-2,147,483,648 to 2,147,483,647
long	twos-comple ment 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

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

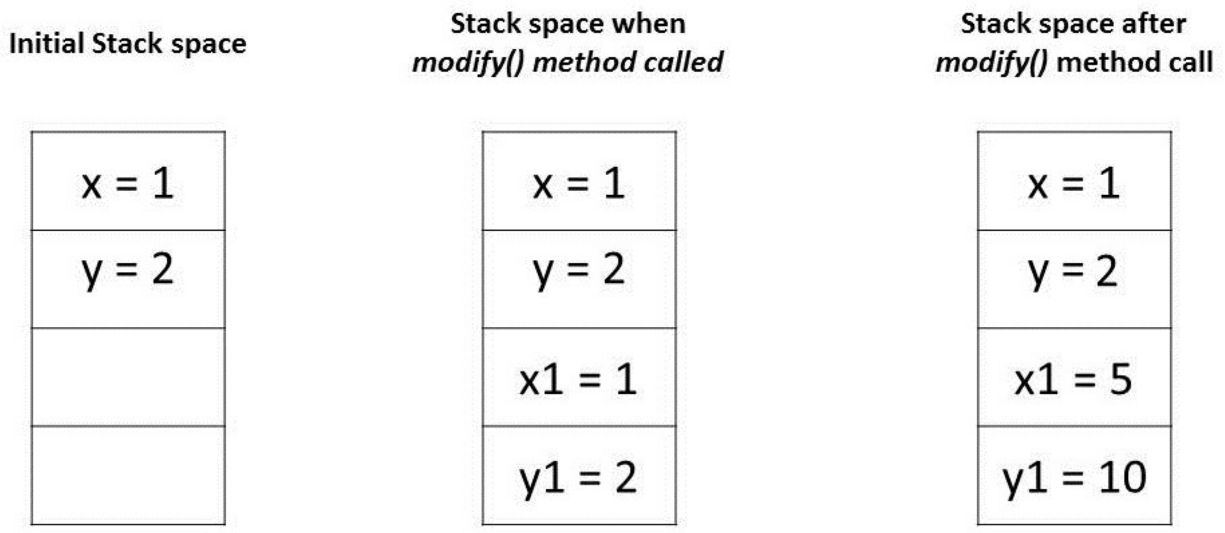
Refer: https://www.baeldung.com/java-pass-by-value-or-pass-by-reference

-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:

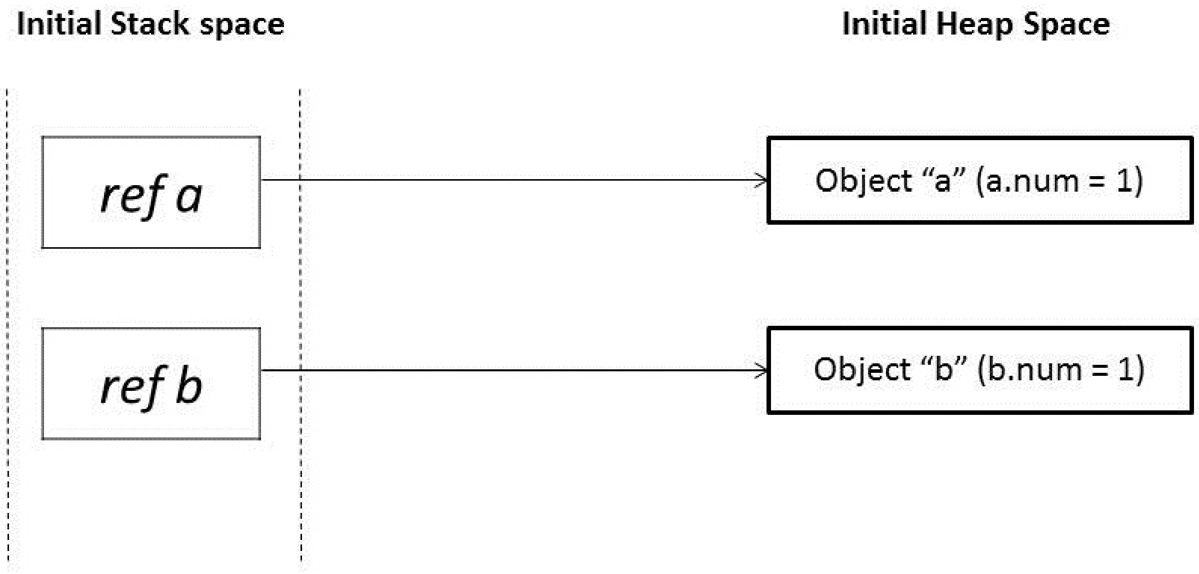
```
public class PrimitivesUnitTest {

   @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;
   }
}
```

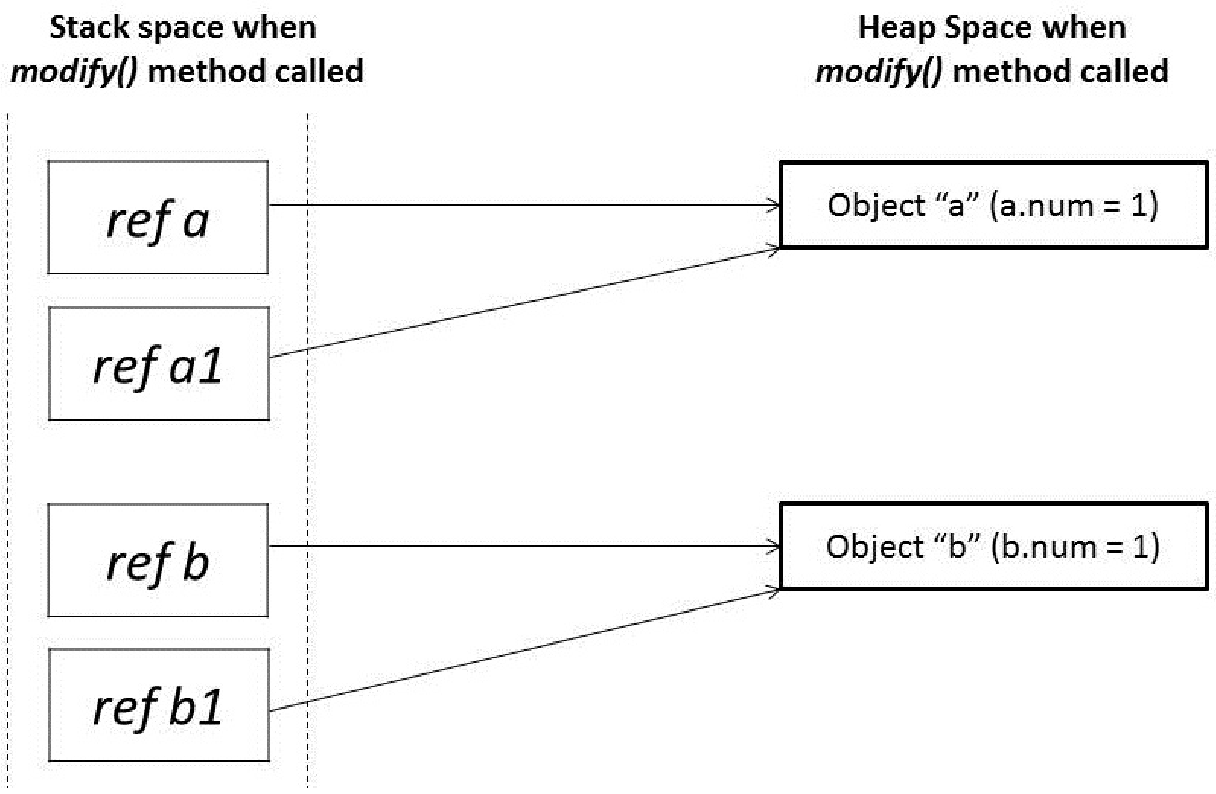

Initial Stack space
x = 1
y = 2
Stack space when modify() method called
x = 1
y = 2
x1 = 1
y1 = 2
Stack space after modify() method call
x = 1
y = 2
x1 = 5
y1 = 10

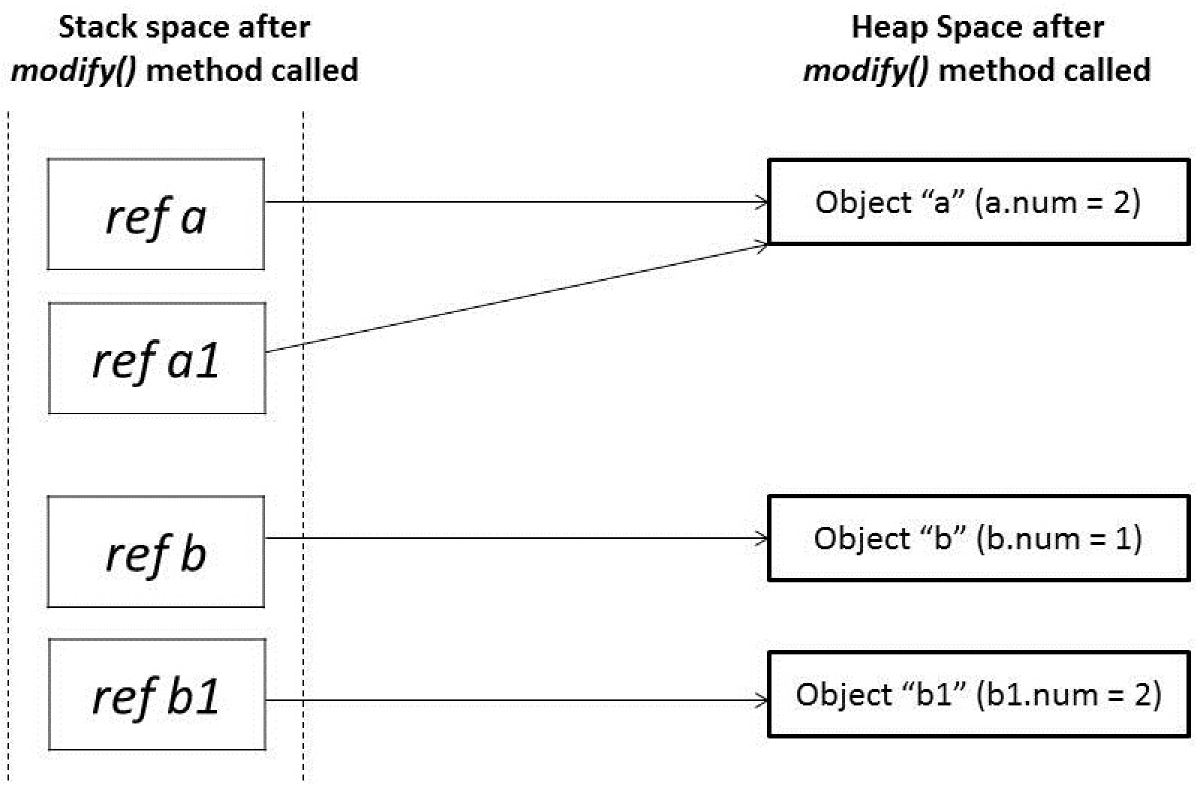
7.2 Passing Object References

-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;
    }
}
```




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


Stack space after *modify()* method called
Heap Space after *modify()* method called
ref a
ref a1
ref b
ref b1
Object “a” (a.num = 2)
Object “b” (b.num = 1)
Object “b1” (b1.num = 2)

8.Array

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

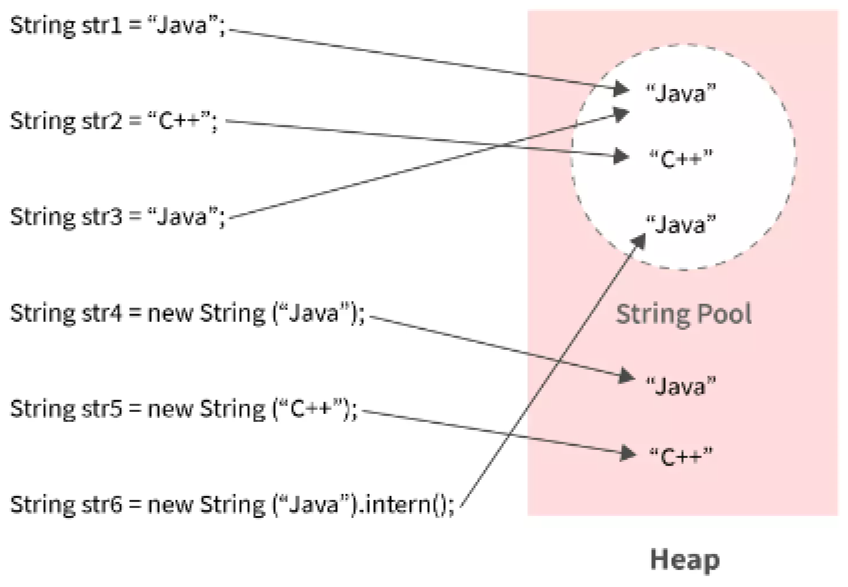
Uppercase Letters − A - Z having ASCII values from 65 - 90 where 65 and 90 are inclusive.
Lowercase Letter − a - z having ASCII values from 97 - 122 where 97 and 122 are inclusive.
Numeric values − 0 - 9 having ASCII values from 48 - 57 where 48 and 57 are inclusive.

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 str1 = "Java";
String str2 = "C++";
String str3 = "Java";
String str4 = new String ("Java");
String str5 = new String ("C++");
String str6 = new String ("Java").intern();
"Java"
"C++"
"Java"
String Pool
"Java"
"C++"
Heap

```
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

O(N^2)

String builder