JAVA INTERVIEW QUESTIONS

Explain JVM, JRE, and JDK?

- JDK: Java Development Kit. It helps you write a java program and save it as source code (.java). When you compile the code, it is turned into a set of Byte Code and stored in a ".class; file.
- JRE: Java Runtime Environment: It has the minimum requirements to execute a Java application. It contains JVM and some libraries.
- JVM: Java Virtual Machine: It runs byte code (.class). JVM is included in both JDK and JRE. Whenever you run a Java program, JVM executes the program line by line so that is why it is called interpreter.

Difference between PATH and CLASSPATH

- PATH: It is used to define where the executables are; for example, .exe files, java.exe, javac.exe,etc.
- CLASS PATH: It is used to specify location of Java .class files.

Difference between Java 7 and Java 8

- Java 8 New Features:
    - Lambda Expressions
    - Date and Time API
    - Pipeline and Streams

How many types of memories located in JVM

- Heap: Objects
- Stack: Variables

How many types of Garbage collector

- Garbage collector automatically checks the heap memory and deletes the unused objects.
- You cannot force the garbage collector to delete the files.
- Types of Garbage Collector: Serial, Parallel, CSM (Concurrent Mark Sweep), G1 (Garbage First)

Explain *public static void main(String args[])*

Can one class call another class's main method?

- When we click run, JVM looks for the main method first.
- "String args []" is the parameter passed to the main method.
- Yes, a main method can be called in another class's main method.
- You can overload the main method but you cannot override it since overloading happens in the class and static attributes are not shared with the objects.

What are primitives and wrapper classes? Implicit and explicit casting?

- Primitive: It is simply just a container to keep/store data. Boolean, byte, char, int, float, double, long, short.
- Wrapper Class: Every primitive has a wrapper class. A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.
- Java Type Casting is classified into two types.
    - Widening Casting (Implicit) – Automatic Type Conversion
    - Narrowing Casting (Explicit) – Need Explicit Conversion

Autoboxing and Unboxing

- Autoboxing: Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.
- Unboxing: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double etc.

Difference between Static and Instance Variables/Block

- Static Variable:
    - It belongs to the class which means there is only one copy.
    - Static can be variables, method, block, inner class.
    - Static methods cannot call/refer non-static members.
- Static block will be executed first. Whenever you call the class, the static block will be called/executed before everything. It will be executed only once when the class is loaded.
- Instance Variable: It is the property of the object which means every object will have its own copy unlike static being only one class copy.
- Instance block: It will be called every time the object is created, even before the constructor.

    In my project:

    - Singleton Driver: I have one Driver declared static because I don't want anyone to make a copy of it but use the same variable.

What is a constructor in Java? Difference between constructor and method?

- Constructor:
    - It is a special method which instantiates/creates an object.
    - It needs to match with the class name.
    - There is always a constructor in a class event if you don't define one. You will have the default constructor in this case.
    - Abstract classes will not have constructor since you can't instantiate abstract classes.
    - We can have private constructor to prevent object creation.
        - We have a default and parameterized constructor.

In my project:

- Driver Class: In my project I have a Driver class where I define the constructor private since I don't want anyone to create any object out of it.

Can super() and this() keywords be in the same constructor?

- No, both of them need to be in the first line in the constructor. We need to pick one or the other.
- super() will call the super/parent class' constructor..
- super**.** will let us access the parent/super class' variables, members, methods, etc.
- this() will call a constructor from another constructor within the same class.
- this**.** will access instance variables and methods.

What are the access modifiers?

- Public: accessible from everywhere
- Protected: accessible within the same package or sub classes in different packages. It is a package private as long as it is not inherited.
- No modifier/default: accessible only within the same package. It's also called package private.
- Private: accessible only within the class.

In my project:

- I use private access modifiers to achieve encapsulation to limit the outside access. I make the constructor private to avoid class instantiation as in the Driver class. All fields in POJO classes are private.

Pass-by-Value vs Pass-by-Reference

- Java is a "pass-by-value" language. This means that a copy of the variable is made and the method receives that copy. Assignments made in the method do not affect the caller.

```
public static void main(String [] args) {
   int num =4;
  newNumber(num);
   System.out.println(num); // 4              }
static void newNumber(int num) {
   Num =7;                    }
```

Final vs finally vs finalize()

- Final: it is a keyword and used to apply restrictions on class, method, and variable.
  - Final Class = CAN't be inherited
  - Final Method = CAN't be overridden
  - Final Variable = CAN't be changed
- Finally: it is used for exception handling with try and catch block.
  - It is executed every time even if an exception happens or not.
  - Most of the time this block is used to close the resources like database connection, I/O resources, etc.
  - In my project, we never used it but used try and catch block only.

- Finalize(): this is a method coming from Object class inherited to every class.
    - Used by the garbage collector to perform some final operations or clean up operations on an object before it is removed from the memory.

Where did you use static in your framework?

- I create **utilities classes** in my project for better code reusability and reduce redundant codes. In the utilities classes, I use **static methods** so that I can reach/call by the class name and use anywhere in my project.
    - In my project:
        - public static WebDriver getDriver (String browser) method
        - public static void closeDriver () method
        - Browser Utils: waitPlease(); verifyIsDisplayed() are static
        - ConfigurationReader: public static getProperty(String key)
- I also have a DatabaseUtil **class** where I create **static methods** to connect and run queries and get data. Check this in the project.

Difference between equals method and "==" operator in Java?

- Double equals "==" checks if two reference variables are pointing to the same object. It is basically reference and address comparison.
- ".equals" checks if the content is the same. Even if two references are pointing to two different objects in the memory, .equals method will look if they have the same content. It is a content comparison.

What is String Pool in Java?

- String pool in Java is a special place to store all Strings in the **Java Heap Memory**. It is used to save memory. When we create a new String with double quote "", Java looks into the String Pool first to check if that Sting (text) already exists or not. If it does, Java creates another reference for the same existing String in the pool but if it does not exist, it creates a new String. This helps to save memory.

How to reverse a String?

```
String name = "HadiBeniCevir";
String reversed ="";
for (int i = name.length()-1; i>=0; i--){
  reversed += name.charAt(i); }
System.out.println(reversed);
```

How to reverse a sentence word by word?

```
String sentence = "Hadi bakalim bu cumleyi cevir!";
String reversedSentence = "";
String [] sentenceArray = sentence.split(" ");
for (int i = sentenceArray.length-1; i>=0; i--) {
  reversedSentence += sentenceArray[i]+" "; }
reversedSentence = reversedSentence.trim();
System.out.println(reversedSentence);
```

String vs StringBuilder vs StringBuffer?

- Strings are **immutable** while StringBuffer and StringBuilder are mutable. So, Strings cannot be changed when you use the String class; whereas Strings can change if you use the StringBuffer and StringBuilder class.
- StringBuffer is **synchronized** or **thread-safe**, StringBuilder is **non-synchronized** or **not-thread-safe.**
- StringBuilder can be called simultaneously which makes it more efficient (faster) than StringBuffer.
- StringBuilder and StringBuffer have limited methods compared to String:
    - StringBuilder str = new StringBuilder("Hello"); o str.append("Java"); ⇒ //Hello Java
    - str.insert(1,"Java"); ⇒ //HJavaello
    - str.replace(1,3,"Java"); ⇒ //HJavalo
    - str.delete(1,3); ⇒ //Hlo
    - str.reverse(); ⇒ //olleH
- In my project, I used String only

How to check if a word is Palindrome?

- It is very easy when you reverse the word and use the ".equals()" method.
- word.equals(reversedWord) = if this returns "true" then it is a palindrome.
- You can also use the StringBuilder to reverse the word.

What is Singleton class and how can we make a class Singleton?

- Singleton is a design pattern in OOP.
- Design Pattern is a collection of best practices by the experienced OOP developers and their solutions to the general problems that they faced during the development.
- How to make a class Singleton:
    - Create static instance
    - Make constructor private
    - Create a public static getInstance() method, which will return an instance.

```
public class Singleton{
        private static WebDriver driver = new ChromeDriver(); // private static instance
        private Singleton(){        } // private constructor
        public static WebDriver getDriver(){ // public static getInstance() method
          return driver;   }}
```

- The purpose is to make sure that there is only one instance of the class and provide a global/public access point to it.

  In my project: I have a Driver class in our utilities package which is created in Singleton design pattern.

How can you prevent instantiation of a class?

- You can create the class with a **private constructor** or you can make the class **abstract.**
- The best way is with the private constructor since you can create a subclass of an abstract and instantiate it.

What are the OOP Concepts in Java?

- There are four components: Abstraction, polymorphism, encapsulation, and Inheritance.
- **Encapsulation:**
    - It is hiding data by making the variables **private** and giving them **public** access with **getters and setters**.
    - In my project: I created multiple POJO/BEAN classes(Json to java (Gson, Jakson)) in order to manage test data and actual data. EX: I take JSON from API response and convert to object of my POJO class all variables are private with getters and setter. Also in order to store credentials or sensitive data in my framework I have used encapsulation, configuration reader also known as property file or excel sheet to hide data from the outside world to get access. I use Apache POI in the data store in Excel in order to extract/read and modify data..
- **Inheritance:**
    - It is used to create a relationship between parent and child classes.
    - In my project, we have an abstract BasePage class. It is used as the parent/super class for all of the page classes since it's extended by all page classes. For example, it initializes PageFactory and contains common global web elements/methods. I have waitUntilLoaderScreenDisappear(), navigateToModule(String tab, String module), etc.
- **Abstraction:**
    - It is about hiding the implementation but showing the functionality only.
    - Abstraction helps you to focus on what the object does instead of how it does it.
    - Two ways to achieve abstraction in Java: Abstract classes (partial abstraction) and interface (full/pure abstraction).
    - In my project, BasePage class is abstract since it shouldn't be instantiated and we don't have a related page in our application.
- **Polymorphism:**
    - It is performing one single action/method in different ways or forms.
    - In my project, WebDriver driver = new ChromeDriver(). I also use method overloading and method overriding which are considered as polymorphism. List<String> = new ArrayList();

Abstract Class and/vs Interface

- Both help us to achieve abstraction in Java and both cannot be instantiated.
- Abstract Class is a class and can be partially abstract since it can include concrete methods.
- Interface is pure abstraction since there is no concrete method in it and it is just like a blueprint for the class implementing it. Interface can have abstract methods, default methods, static methods, and public final static variables.
- We can **extend** only one Abstract class but when we want to use an interface, we may **implement** as many interfaces as we want.

Difference between overloading and overriding?

- **Overriding:**
  - Happens in the child class. It is called dynamic binding.
  - Same method name and same parameters (same method signature).
  - Must have the same or sub return type.
  - We cannot override static methods, we can only hide them.
  - Access modifiers can only be the same or more visible.
  - Private methods cannot be overridden since they are not inherited.
  - Final methods cannot be overridden since they are final, cannot be changed.
  - In my project: I override .toString(), .equals(), and .hashCode() methods.
- **Overloading:**
  - Happens in the class nesting the method or in the inherited methods. It is called static binding.
  - Same method name but different parameters.
  - Return type can be different.
  - We can overload static methods.
  - Access modifiers can be different.
  - Final methods can be overloaded in the same class.
  - In my project:
    In the BrowserUtils, I have methods like:

```
public static WebElement waitForVisibility(WebElement element, int timeToWaitInSec) {
WebDriverWait wait = new WebDriverWait(Driver.get(), timeToWaitInSec);
 return wait.until(ExpectedConditions.visibilityOf(element));        }

public static WebElement waitForVisibility(By locator, int timeout) {
WebDriverWait wait = new WebDriverWait(Driver.get(), timeToWaitInSec);
 return wait.until(ExpectedConditions.visibilityOf(element));        }
```

What are Data Structures and Why do we need them?

- Data structures are a way of organizing data for efficient manipulation: insertion, searching, reading, and deletion of data.
- When there is more than one variable, it is data structure: Array, ArrayList, Maps, Set, HashMaps, LinkedLists, etc.
- Array: It is the simplest data structure. It is not resizable so it is static.
- In my project, I always use Java data structures for reading data and storing data from our application, database, or API.

What is Array? - Data Structure

- An array is a **container** object that holds a fixed number of **values of a single type**. The length of an array is established when the array is created. After creation, its l**ength is fixed**.
- Each item in an array is called an element, and each element is accessed by its **numerical index.** Numbering begins with **0**. The 9th element, for example, would therefore be accessed at index 8.
- **Advantage of Java Array:**
  - Code Optimization: It makes the code optimized, we can retrieve or sort the data easily.

- Random access: We can get any data located at any index position.

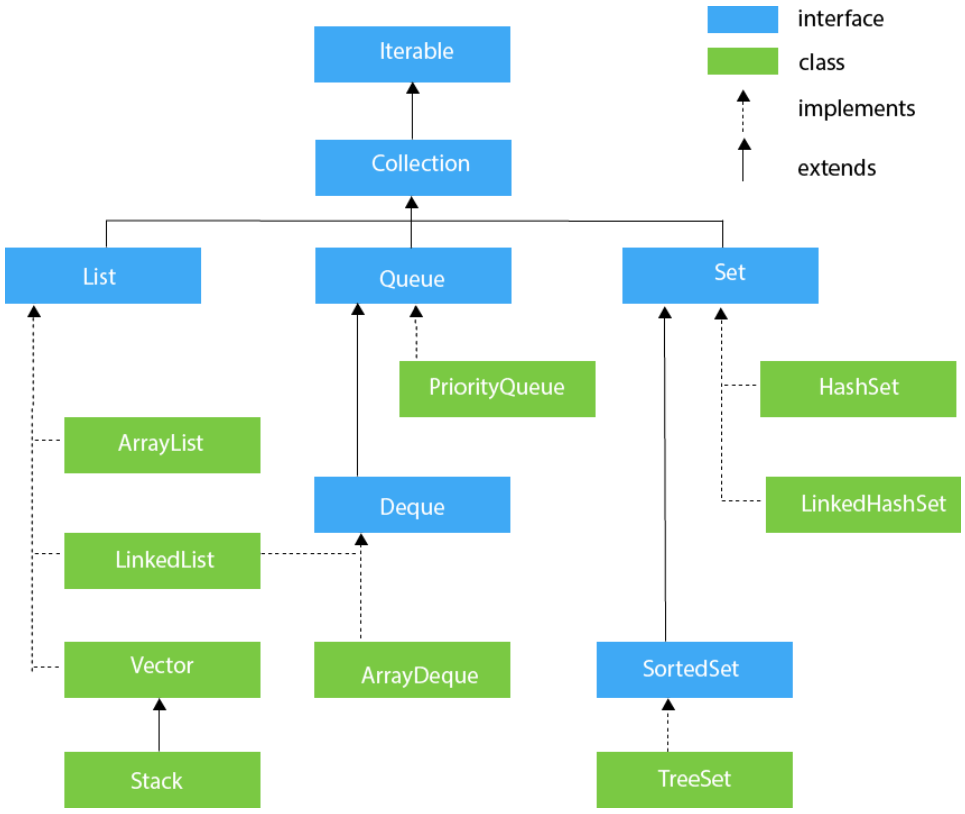
- **Disadvantage of Java Array:**
  - Size Limit: It is not dynamic so it is not resizable.

Array vs ArrayList - Data Structure

- **Array:**
  - Fixed size, cannot be resized.
  - Can store primitive types and objects.
  - Can be multidimensional.
  - It is faster but in my project it does not matter that much.
  - We find the length of an Array with ***.length*** attribute.
- **ArrayList:**
  - Dynamic and can be resized.
  - Can store only wrappers and objects,
  - Cannot be multidimensional.
  - It is slower compared to Arrays.
  - We find the length of an ArrayList with ***.size()*** method.

Java Collection Framework - Data Structure

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit. The Collection interface (java.util.Collection) and Map interface (java.util.Map) are the two main "root" interfaces of Java collection classes.



java.util.Collection: This includes interfaces like List, Set, and Queue

1. LIST - Interface: *(list of things) - A List is an ordered Collection (sometimes called a sequence). Lists may contain **duplicate elements**. Elements can be inserted or accessed by their position in the list, using a **zero-based index**. They are **dynamic sized - resizable.***

- ArrayList - class:
  - It is a class implements List interface and it is based on an **Array data structure**. It is widely used because of the functionality and flexibility it offers.
  - It is **resizable** unlike the Arrays. ArrayList can dynamically grow and shrink after addition and removal of elements
  - It has a fast **iteration** and is ordered by **index.** Preferably to use when searching for an element/object.
  - It is **not sorted** but can be sorted using ***Collections.sort()*** method.
- LinkedList - class:
  - LinkedList is a l**inear data structure**.
  - LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using pointers. Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList.
  - It has **slower iteration** but **faster addition** and **deletion**. Preferably to use when adding and deleting an object/element.
- Vector - class:
  - Vector implements List Interface.
  - Like ArrayList it also maintains **insertion order** but it is rarely used in a non-thread environment as it is **synchronized (thread-safe)** and due to which it gives poor performance in searching, adding, deleting and updating of its elements.
  - Vector has a child class: Stack
    - Stack: synchronized (thread-safe)
    - pop(): LIFO ==> Last in First Out - removes the last object from the stack

2. SET - Interface: *(Set of **unique** things)* A Set is a Collection that **cannot** contain **duplicate elements**. Classes implementing Set are HashSet, TreeSet, and LinkedHashSet.

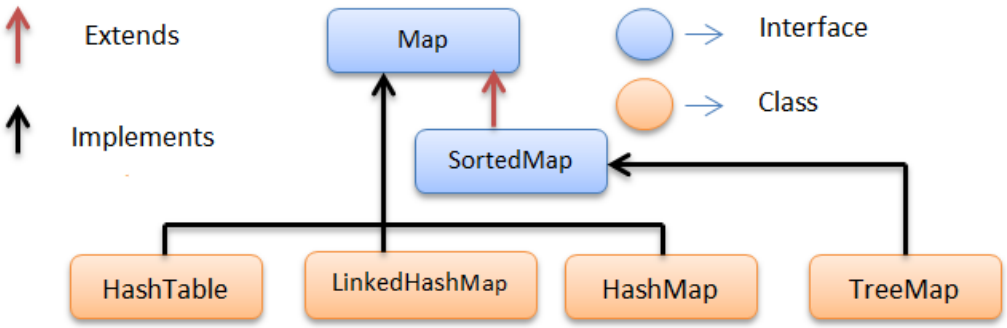
- HashSet - class:
  - It doesn't maintain any order, the elements would be returned in any **random order**.
  - It doesn't allow duplicates. If you try to add a duplicate element in HashSet, the old value would be overwritten. **(no duplicate value)**
  - It allows null values however if you insert more than one null, it would still return only one null value.
- LinkedHashSet - class:
  - It maintains the **insertion order**. Elements get sorted in the same sequence in which they have been added to the Set.
  - It is basically an ordered version of HashSet so you can use it when you care about the insertion order.
- TreeSet - class: implementing SortedSet Interface

- TreeSet is similar to HashSet except that it sorts the elements in the **ascending order** while HashSet doesn't maintain any order.
- It is substantially **slower** than HashSet.

3. QUEUE - Interface:

A Queue is designed in such a way so that the elements added to it are placed **at the end of Queue and removed from the beginning of Queue.** The concept here is similar to the queue we see in our daily life, for example, when a new iPhone launches we stand in a queue outside the apple store, whoever is added to the queue has to stand at the end of it and persons are served on the basis of **FIFO (First In First Out),** The one who gets the iPhone is removed from the beginning of the queue. The Queue interface in Java collections has two implementations: LinkedList and PriorityQueue, these two classes implement Queue interface.

Java Map Interface - Data Structures



MAP Interface: A Map is an object that maps **keys** to **values**. A map cannot contain duplicate keys. Both key and value do not support primitives. There are three main implementations of Map interfaces: **HashMap**, **TreeMap**, and **LinkedHashMap**.

- HashMap: - class
  - It is **NOT an ordered collection** which means it does not return the keys and values in the same order in which they have been inserted into the HashMap.
  - It **does NOT sort** the stored keys and Values.
  - Accepts only **one null key** but any number of null values.
- LinkedHashMap: - class
  - maintains the **insertion order** as it is doubly linked. It has a **faster iteration**.
  - put() & remove() are faster than hashmap // modifying the data.
  - get() is slower than hashmap // retrieving the data.
- HashTable: - class
  - It is **synchronized**. It ensures that no more than one thread can access the Hashtable at a given moment of time.
  - It doesn't allow null keys and null values. **NO nulls** at all.
- TreeMap: - class implements SortedMap Interface
  - It is sorted in the **ascending order of its keys**.
- **MAP Methods:**
  - **.put(key, value):** inserts key and value objects to the map.

- **.get(key):** retrieves the value of the given key.
- **.remove(key):** removes the given key object and its value.
- **.size():** returns the size of the map.
- **.containsKey(key):** verifies if the given Key exists in the map and returns boolean.
- **.containsValue(value):** verifies if the given value exists in the map and returns boolean.
- **.keySet():** returns all the keys as Set Interface.
- **.clear():** removes everything in the map, size will become 0 after the execution.

Difference between Set, List and Map in Java?

- **Set**, **List** and **Map** are 3 important interfaces of Java collection framework.
  - **List** provides **ordered** and **indexed** collection which may contain **duplication**.
  - **Set** provides an unordered collection of **unique** objects. There is **NO duplication**. Both List and Set implement the **collection** interface.
  - **Map** provides a data structure based on **Key - Value**. Key is always **unique**, Value can contain **duplicate values**.

TreeSet vs TreeMap

- TreeSet: can contain only unique values - is sorted in ascending order
- TreeMap: can contain only unique keys. - keys are sorted in ascending order

Difference between Hashtable and HashMap in Java?

- Hashtable is synchronized, whereas HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
- Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
- For example; one of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.

If synchronization is not an issue for me, I prefer using HashMap. If it becomes an issue, then I prefer Collections.synchronizedMap() or ConcurrentHashMap.

- Both Hashtable and HashMap implement Map interface and both are Key and Value.
- HashMap is not thread-safe while Hashtable is a thread-safe collection.
- Second important difference is performance since HashMap is not synchronized so HashMap performs better than Hashtable.

Iterator and Iterable

- Iterable(I): extended by root interface collection we can apply Iterator(I) to the classes or interfaces that are sub type of Iterable
- Iterator(I):
    - Iterator is used for iterating (**looping**) various collection classes such as HashMap, ArrayList, LinkedList etc.
    - Allows us to get access to each objects of the collection type
    - Allows us to remove objects from a collection type. **The ONLY legit way to remove data elements** from a collection-type.
    - **.iterator() method:** iterates the collection, and returns Iterator.
    - **.hasNext() method:** checks if there are enough elements that can be iterated and returns boolean expression.
    - **.next() method:** if hasNext() is true, it retrieves the current iteration from the collection type
    - **.remove() method:** removes current element of the iteration from the collection type

What is the Iterator and difference between for-each loop?

- Iterator works with ArrayList, not with Array. It helps us to iterate through the elements.
- With an iterator you can make changes (remove item) to the list while iterating.
- With for-each loops, we can only iterate through lists but cannot make any changes to our lists.

Remove all data elements that are equal to 1

```
List<Integer> list2 = new ArrayList<>();
list2.addAll(Arrays.asList(1,1,1,2,2,2,3,3,4,5,6,7,8,1,1,1));
Iterator<Integer> it =  list2.iterator();
        while (it.hasNext()){
          int num = it.next();
          if(num==1){
            it.remove();  }
        }          System.out.println(list2);
```

How to remove names which contain a specific letter “e” from a collection type?

```
String[] namesArray = {"Ali", "Veli", "Semavi", "Sami", "Hayati", "Memati", "Canan"};

Set<String> namesSet = new LinkedHashSet<>();

namesSet.addAll(Arrays.asList(namesArray));

Iterator<String> namesSetIterator = namesSet.iterator(); // returns an Array of “namesSet”
        while (namesSetIterator.hasNext()){
          String str = namesSetIterator.next();
                  if(str.toLowerCase().contains("e")){
                    namesSetIterator.remove();  }
        }          System.out.println(namesSet);
```

Write a program which returns even numbers only

```
List<Integer> numbers = new ArrayList<Integer>(Arrays.asList(1,2,5,343,12,6,8,9,34,53));
Iterator<Integer> numIt = numbers.iterator();
        while (numIt.hasNext()){
          int num = numIt.next();
          if(num%2==0){
            System.out.println(num);        }           }
```

How to Iterate thru Maps?

```
Map<String,String> contacts = new HashMap<String,String>();
        contacts.put("Ali", "Durhan");
        contacts.put("Veli", "Karaca");
        contacts.put("Semih", "Sayginer");
        contacts.put("Halit", "Karabiyik");
        contacts.put("Sami", "Yavuz");
```

ForEach:

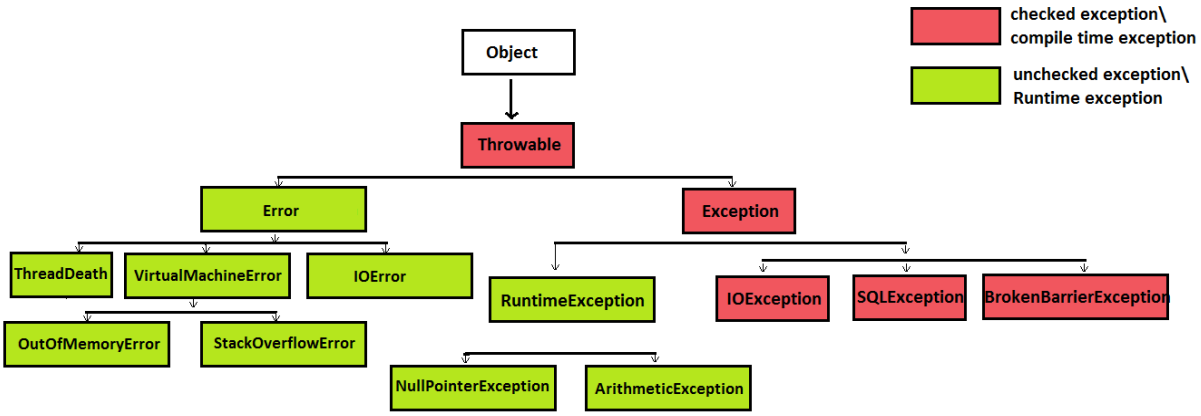
```
for (String each:contacts.values()) {
System.out.println(each);
        }
```

Iterator:

```
Iterator<Map.Entry<String, String>> iterator = contacts.entrySet().iterator();
while (iterator.hasNext()){
  Map.Entry<String, String> entry = iterator.next();
  System.out.println(entry.getKey() + " = " + entry.getValue());        }
```

Exceptions in Java

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.



Difference between error and exception

- **Errors** indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.
- **Exceptions** are events that occur in the code. A programmer can handle such conditions and take necessary corrective actions.

Types of exceptions

- **Checked exceptions:**
  - All exceptions other than Runtime Exceptions are known as Checked exceptions as the **compiler checks them during compilation** to see whether the programmer has handled them or not.
  - If these exceptions are not handled/declared in the program, you will get **compilation error.** For example, SQLException, IOException, ClassNotFoundException etc.
- **Unchecked Exceptions:**
  - **Runtime Exceptions** are also known as Unchecked Exceptions. These exceptions are **not checked at compile-time** so the compiler does not check whether the programmer has handled them or not.
  - It's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Exception Handling

- Try & Catch Block: to handle the exception

```
try {
  System.out.println(new int[] {1,2,3}[100]); // out of bound exception occurs here
} catch (Exception e){
  System.out.println("Out of Bound Exception caught here");
} finally {
  System.out.println("Finally will be executed here regardless of any exception occurs or not"); }
```

- Throws keyword: to pass it to the caller's responsibility to handle it. "**throws"** keyword indicates that this method throws an exception.

```
public static void method1 () throws Exception{
  Thread.sleep(2000);   }
```

Difference between throw and throws in Java

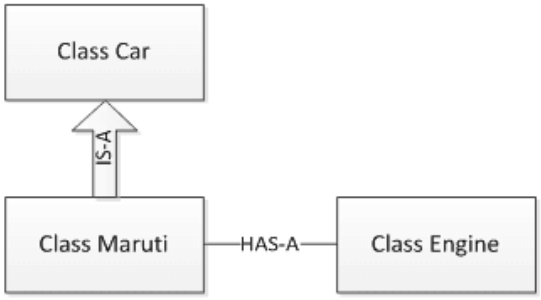
- **throw** keyword is used to throw an exception **explicitly**
- **throws** keyword is used to **declare** an exception which means it works similar to the try--catch block
- throw new RuntimeException ("Invalid browser name!") - throws ArithmeticException; **We throw RuntimeException in the Driver class.** You need to pass the driver's name (chrome, firefox,etc.) if the text does not meet the available browser we want to throw an exception.

```
public void deposit(double amount) throws RemoteException {
  // Method implementation
  throw new RemoteException();          }
```

What's the difference between IS-A and HAS-A relationships?

- **IS-A Relationship:**

  In object-oriented programming, the concept of IS-A is totally based on **Inheritance**, which can be of two types: Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is unidirectional. For example, House is a Building. But Building is not a House.



  Wherever you see an "**extends**" keyword or "**implements**" keyword in a class declaration, then this class is said to have IS-A relationship. It is a parent-child relationship.

- **HAS-A Relationship:**

  Composition(HAS-A) simply means **the use of instance variables** that are references to other objects. For example, Honda Civic has an Engine, or a House has a Bathroom.

What Is a Static Keyword?

- The static keyword can be used with class level variables to make it global i.e all the objects will share the same variable.
- The static keyword can be used with methods also. A static method can access only static variables of class and invoke only static methods of the class.

What Is System.gc()?

- A request to JVM to run Garbage collector to free up memory but it is not a command so it may not be executed.
- It is up to the garbage collector to honor this request.

What is thread safe or synchronized?

- The Java synchronized keyword is an essential tool in concurrent (simultaneous) programming in Java.
- Its overall purpose is to only allow **one thread at a time** into a particular section of code thus allowing us to **protect**, for example, variables or data from being corrupted by simultaneous modifications from different threads.
- JAVA keyword synchronized is used to create synchronized code and internally it uses locks on Object or Class to make sure only one thread is executing the synchronized code.
- JVM guarantees that synchronized code will be executed by only one thread at a time.
- In my project, I created the "getDriver()" method in Driver class as thread safe because I don't want the Driver to be called simultaneously.

JAVA CODING PRACTICE

Write a program that prints out the numbers from 1 to 30 but for numbers which are a multiple of 3, print "FIN" instead of the number and for numbers which are a multiple of 5, print "RA" instead of the number. For numbers which are a multiple of both 3 and 5,print "FINRA" instead of the number.

```
for(int i=1; i <= 30; i++) {

  String print = "";

  if(i % 3 == 0) print = "FIN";

  if(i % 5 == 0) print += "RA";

  if(print.isEmpty()){
    System.out.println(i);
  } else {
    System.out.println(print);
  }
}
```

Write a program that will count how many times “java” is found in any given String:

Approach One:

```
String givenStr = "Kac defa java geceiyok ki java sayisini javada say";
int counter = 0;
for (int i = 0; i<givenStr.length()-3; i++){
  if(givenStr.substring(i, i+4).equalsIgnoreCase("java"))
    counter++; }
System.out.println(counter);
```

Approach Two:

```
String str = "burada java yazdik ki Java kacta gesiyor bul ve java sayisini yazdir";
int javaCounter=0;
str = str.toLowerCase(); //to be make the str case insensitive

while(str.contains("java")){
  javaCounter++;
  str = str.replaceFirst("java","");
  //this will replace java with an empty string and return the remaining str.
}  System.out.println(javaCounter);
```

Given any String determine if it is Palindrome. Print “Palindrome” if it is and “Not Palindrome” if it is not:

Approach One:

```
String text = "amalama";
boolean check = true;

for (int i=0; i<text.length()/2; i++){
  if(text.charAt(i) != text.charAt(text.length()-1-i)){
```

```
        check=false;
      break; }
    }
    System.out.println(check ? "Palindrome" : "Not Palindrome");
```

Approach Two:

```
    String str1 = "deneme";
    String str2  ="";

    for (int i= str1.length()-1; i>=0;i--)
      str2 += str1.charAt(i);

    if (str1.equalsIgnoreCase(str2))
      System.out.println(str1 + " is a palindrome" + " Here is the reversed = " + str2);
    else
      System.out.println(str1 + " is NOT a palindrome!"+ " Here is the reversed = " + str2);
```

Approach Three:

```
    String str1 = "deneme";
    String str2 = new StringBuffer(str1).reverse().toString();

    System.out.println(str2);
```

Given any String print out how many times each character is found in the String

```
    String str0 = "CyberTek";
    int letterCounter =0;
    str0 = str0.toLowerCase();
    String usedLetters="";

    for (int i=0; i<str0.length();i++){
      letterCounter=0;
      if(usedLetters.contains(str0.charAt(i)+"")) continue;
        // if we used the letter before it will skip this iterator/loop

              for (int j=0; j<str0.length();j++ ){
                if(str0.charAt(i)==str0.charAt(j)){
                  letterCounter++;
                } }
        System.out.println(str0.charAt(i)+ " = " + letterCounter);
      usedLetters +=str0.charAt(i);  // creating a new string from the letters coming out of the loop so
                  // we don't count them again and have the unique letters only
    }
```

Create a program that will take any String and print the total sum of all the numbers in the String. Note: numbers can be more than digits from 1-9 so if you have "14" next to each other it should be considered 14 and not 1 and 4 separate:

Approach One:

```
    String str = "jav45ai15sgre1at82";
    int sum =0;
    String num = "";

      for (int i = 0; i <str.length() ; i++) {

        if(str.charAt(i)>='0' && str.charAt(i)<='9'){
          num += str.charAt(i);
```

```
        if(i==str.length()-1 || !(str.charAt(i+1)>='0' && str.charAt(i+1)<='9')){
          sum += Integer.parseInt(num);
          num="";      }      }            }
    System.out.println(sum);
```

Approach Two:

```
String str = "jav45ai1000sgre1at82";
int sum = 0; // 4 + 158
String num = ""; // 158

for(int i=0; i < str.length(); i++) {

  if(Character.isDigit(str.charAt(i))) {
    num += str.charAt(i);

    if(i == str.length()-1 || !Character.isDigit(str.charAt(i+1))){
      sum += Integer.parseInt(num);
      num = "";
    } } }              System.out.println(sum);
```

Given two Strings determine if they are Anagrams -> Are built of the same characters:

Approach One:

```
String str1 = "listen";
String str2 = "silent";
for (int i = 0; i <str1.length() ; i++) {
    str2 = str2.replaceFirst("" + str1.charAt(i), "");  }
 System.out.println(str2.isEmpty() ? "Anagram" : "NOT Anagram"); }
```

Approach Two:

```
String str1 = "listen";
String str2 = "silent";

String str11 = "";
String str22 = "";

char[] ch1 = str1.toCharArray();
char[] ch2 = str2.toCharArray();
Arrays.sort(ch1);
Arrays.sort(ch2);

for (char each:ch1) {
  str11+=each;
}
for (char each:ch2) {
  str22+=each;
}

 System.out.println(str11.equalsTo(str22)? "Anagram" : "NOT Anagram");
```

Approach Three:

```
public static boolean Same(String str1, String str2) {
  str1 = new TreeSet<String>(Arrays.asList( str1.split("") ) ).toString( );
  str2 = new TreeSet<String>(Arrays.asList( str2.split("") ) ).toString( );
  return  str1.equals(str2);   }
```

Password Validation

1. Password MUST be at least 8 characters
2. Password should at least contain one uppercase letter
3. Password should at least contain one lowercase letter
4. Password should at least contain one special characters
5. Password should at least contain a digit

if all requirements above are met, the password is valid, if not all are met it is invalid

Approach One:

```
String password = "a?G6jdsaja";

boolean length = password.length() >= 8;
boolean lower = password.matches(".*[a-z].*");
boolean upper = password.matches(".*[A-Z].*");
boolean number = password.matches(".*[0-9].*");
boolean special = password.matches(".*[!@.,#$?].*");

boolean valid = length && lower && upper && number && special;

if(valid) {
  System.out.println("Password is valid");
} else {
  System.out.println("Not a valid password");
}
```

Approach Two:

```
String password = "a?Gj6dsaja";

boolean length = password.length() >= 8;
boolean lower = false;
boolean upper = false;
boolean number = false;
boolean special = false;

for(int i=0; i < password.length(); i++) {
  char c = password.charAt(i);

  if(c >= 97 &&  c <= 122) lower = true;
  if(c >= 65 && c <= 90) upper = true;
  if(c >= 48 && c <= 57) number = true;
  if((c >= 33 && c <= 47) || (c >= 58 && c <= 64)
       || (c >= 91 && c <= 96) || (c >=123 && c <=126)) special = true;}

if(!length) {
  System.out.println("Invalid length");
} else if(!lower) {
  System.out.println("Missing lowercase letter");
} else if(!upper) {
  System.out.println("Missing uppercase letter");
} else if(!number) {
  System.out.println("Missing number");
} else if(!special) {
  System.out.println("Missing special character");
} else {
```

```
    System.out.println("Valid password");
  }
```

Write a return method that can find the frequency of characters

Ex: FrequencyOfChars("AAABBCDD") ==> A3B2C1D2

Approach One:

```
String str = "AAABBKKACCDDDWWWBDD";
String expected ="";

for (int i=0; i<str.length(); i++){

  int num=0;

  for (int j = 0; j < str.length(); j++) {
    if (str.charAt(i) == str.charAt(j)){
    num++;
    } }
  expected = str.charAt(i)+""+num;
  str = str.replace(""+str.charAt(i), "");
 i=-1;
 System.out.print(expected);                    }
```

Approach Two:

Write a return method that can remove the duplicate values from String

Ex: removeDup("AAABBBCCC") ==> ABC

Approach One:

```
String str="AAABBBCCC";
String result ="";

for (int i = 0; i <str.length() ; i++) {

  for (int j = 0; j <str.length() ; j++) {
    if(str.charAt(i) == str.charAt(j)){
      result += str.charAt(i);
      str = str.replaceAll(""+str.charAt(i),"");
    } }
    System.out.println(result);      }
```

Approach Two:

```
String str="AAABBBCCC";
String result = "";

for (int i = 0; i < str.length(); i++)
  if (!result.contains("" + str.charAt(i)))
    result += "" + str.charAt(i);

System.out.println(result);
```

Write a return method that can find the unique characters from the String

Ex: unique("AAABBBCCCDEF") ==> "DEF";

Approach One:

```
String str="AAABBBCCCDEF";
String result = "";

for (int i = 0; i <str.length(); i++) {
    if(str.contains(""+str.charAt(i))){
      str=str.replaceAll(""+str.charAt(i),"");
    } }
  System.out.println(str);
```

Approach Two:

```
String str="AAABBBCCCDEF";
String result ="";

for(String each : str.split(""))
  result +=(  (Collections.frequency(Arrays.asList(str.split("")), each)) ==1 ) ? each : "";
System.out.println(result);
```

Sort Letters and Numbers from alphanumeric String

```
public static void SortLettersAndNumbersFromString(String str) {

 String str2 = "";

 for(int i = 0; i < str.length(); i++) {
   str2 += ""+str.charAt(i);
   if(Character.isAlphabetic(str.charAt(i)) && i < str.length()-1 ) {
     if(Character.isDigit(str.charAt(i+1)) ) {
       str2 +=",";       }       }
   if(Character.isDigit(str.charAt(i)) && i < str.length()-1) {
     if(Character.isAlphabetic(str.charAt(i+1))) {
       str2 +=",";       }       }  }
 String[] arr = str2.split(",");
 str ="";
 for(String each: arr) {
   char[] chars=each.toCharArray();
   Arrays.sort(chars);
   for(char eachChar: chars){
     str +=""+eachChar;     }  }
 System.out.println(str);     }
```

Write a method that can find the maximum number from an int Array

Approach One:

```
int [] nums = {1,4,7,9,3,56,3};
int max=0;

for (int i = 0; i <nums.length ; i++) {
  if (nums[i]>max){
    max=nums[i];
  } }
  System.out.println(max); }
```

Approach Two:

```
public static int maxValue( int[]  n ) {

  int max = Integer.MIN_VALUE;
  for(int each: n)
    if(each > max)
      max = each;
  return max;
}
```

Approach Three:

```
int [] arr = {9,4,23,6,78,4};
Arrays.sort(arr);
int max = arr[arr.length-1];

System.out.println(max);
```

Write a method that can find the minimum number from an int Array

Approach One:

```
int [] arr = {9,4,23,6,78,4};
int min = arr[0];

for (int i=0; i<arr.length; i++){
  if (min > arr[i]){
     min=arr[i];
}          }
System.out.println(min);
```

Approach Two:

```
int [] arr = {9,4,23,6,78,4};
Arrays.sort(arr);
int min = arr[0];

System.out.println(min);
```

Write a return method that can sort an int array in Ascending order without using the sort method of the Arrays class:

```
int [] arr = {9,4,23,6,78,4};

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

     int temp=0;
     if (arr[i]<arr[j]){          // if I change '<' to '>' it will sort from bigger to smaller
       temp=arr[i];
       arr[i]=arr[j];
       arr[j]=temp;                }        }         }
System.out.println(Arrays.toString(arr));
```

Create a method that will accept a number and print all the prime numbers from 2 to that given number:

```
public static void printPrimeNums(int num) {

  for (int i = 2; i <=num ; i++) {
    if (isPrime(i))
      System.out.println(i);
  } }

public static boolean isPrime(int num){
  for (int i =2; i<num; i++){
    if (num%i==0){
      return false;
    }
  }
    return true; }
```

Create a method that will accept a number and check if the number is an Armstrong number. If the number is an Armstrong number return true otherwise return false.

Approach One:

```
public static boolean isArmstrongNum(int num) {

  String number = String.valueOf(num);
  int multiplier=number.length();
  int sum=0;

  for (int i = 0; i < multiplier; i++) {
    int digit = Integer.parseInt(""+number.charAt(i));
    sum+=Math.pow(digit,multiplier);
  }

  return sum==num;
}
```

Approach Two:

```
public static boolean isArmstrongNum(int num) {
  String numo = String.valueOf(num);
  int length =numo.length();
  int number = num;
  int temp=0;
  int sum=0;

  while (number!=0){
    temp=number%10;
    sum+=Math.pow(temp,length);
    number/=10;
  }
  return sum==num;
}
```

Approach Three:

```
int num= 153;
int sum=0;
```

```
String numStr=String.valueOf(num);
int power=numStr.length();
System.out.println(numStr);

int[] numArr= new int [power];
for (int i = 0; i < numStr.length(); i++) {
  numArr[i]=Integer.parseInt(""+numStr.charAt(i));
}

for (int each:  numArr) {
  sum+= Math.pow(each,power);
}
System.out.println("num = " + num);
System.out.println("sum = " + sum);
```

Create a method that will accept a number (long) and determine if the number is palindrome or not.

Approach One:

```
public static boolean isDigitPalindrome1(long number){

  long temp = number;
  long reverse=0;
  int numOfLoops = String.valueOf(number).length();
  String revStr="";

  for (int i = 0; i <numOfLoops ; i++) {
    revStr+=temp%10;
    temp/=10;
  }
  reverse=Integer.parseInt(revStr);
  return reverse==number;
}
```

Approach Two:

```
public static boolean isDigitPalindrome2(long number){

  String num = String.valueOf(number);
  String reverse = String.valueOf(new StringBuilder(num).reverse());
  long reverseLong = Long.parseLong(reverse);

  return reverseLong==number;
}
```

Approach Three:

```
public static boolean isDigitPalindrome3(long number){

  long temp=number;
  long reverseLong =0;
  long lastDigit;

  while (temp!=0){
    lastDigit = temp%10;
    reverseLong=reverseLong*10+lastDigit;
    temp/=10;
```

```
    }
    return reverseLong==number;
}
```

Swap two variable' values without using a third variable

```
int a =10; int b =20;

a = a+b; // a = 30
b = a-b; // b = 10
a = a-b; // a = 20

System.out.println(a + " --- " + b);
```

Write a program that can print the numbers between 1 ~ 100 that can be divisible by 3, 5, and 15.

- ➔ if the number can be divisible by 3, 5 and 15, then it should only be displayed in **DivisibelBy15** section
- ➔ if the number can be divisible by 3 but cannot be divisible by 15, then it should only be displayed in **DivisibelBy3** section
- ➔ if the number can be divisible by 5 but cannot be divisible by 15, then it should only be displayed in **DivisibelBy5** section

ex:
OutPut:

Divisible By 15 15 30 45 60 75 90
Divisible By 5 5 10 20 25 35 40 50 55 65 70 80 85 95 100
Divisible By 3 3 6 9 12 18 21 24 27 33 36 39 42 48 51 54 57 63 66 69 72 78 81 84 87 93 96 99

```
String divisibleBy15 ="";
String divisibleBy5 ="";
String divisibleBy3="";
int[] arr = new int[100];
for(int i=0; i < arr.length; i++)
  arr[i] = i+1;

for(int each: arr) {
  if(each %15==0 && each %3==0)
    divisibleBy15+= each+" ";
  if(each %5==0 && each % 15!=0)
    divisibleBy5 += each+" ";
  if(each%3==0 && each %15!=0)
    divisibleBy3 += each+" ";

}

System.out.println("Divisible By 15: "+divisibleBy15);
System.out.println("Divisible By 5: "+divisibleBy5);
System.out.println("Divisible By 3: "+divisibleBy3);
```