# JAVA

1. JDK comes with Compiler and JRE. JRE comes with JVM and a set of APIs. JVM has JIT.
2. A code is compiled to byte code, which is interpreted by the JVM having JIT (Just in Time) Compiler. Creating Platform Independency.
3. Objects have variables and functions.
4. Encapsulation is protecting the variables and functionalities of an object from other objects.
5. Inheritance is about extending previous classes. Reusability and IS-A relationship.
6. Abstraction is about hiding the unnecessary details of an object.

Static Context vs Non-Static Context

1. Static Context is class level context. All objects share them, we don’t need objects to access them. It is accessed using class name.
2. Non-Static Context is object level context and accessed using object.
3. Static Block is printed before main. In case of many Static Blocks, they are accessed in order.
4. Static variables are stored in the method area of the JVM memory.
5. Non static variables are stored in the heap memory.
6. In Non-Static Context, main method then non-static block, then constructor, then method calls.
7. Static block is executed only once. Even when multiple times class is called.
8. Static variables are initialized at their time of declaration.
9. Java cant do pointer arithmetic like in C provides security and simplicity. It works on references, not direct manipulations.

This keyword:

1. this points to the current object of the class.
2. It can be used only in non-static context, not in main and static context.

Wrapper Class:

1. Wrapper classes convert primitive to objects, this is boxing or autoboxing.
2. Object to primitive conversion is unboxing.
3. These classes are needed, because Collections class uses objects, Primitives can’t be stored in it.
4. Int a=10; Integer x=Integer.valueOf(a); or Integer x=new Integer(a); int z=x.intValue();
5. Int z=5; String s=Integer.toString(z); int y=Integer.parseInt(s);

Only operator that is overloaded in Java is + operator. Adds integers, concatenates strings. If added integer and a string, they are concatenated.

Binary Operations:

1. &,|,^ gives true if both are different.
2. 4&5 equivalent to 100&101=100=4.
3. 4|5 equivalent to 100|101=101=5.
4. 4^5 equivalent to 100^101=001=1.
5. ~ bitwise unary complement which can be applied to only integer values.
6. ! negates the value.
7. && and || are short circuit operators. If one false && returns false, not checking any other. If one true || returns true, without checking any other. The performance improves. Works only for Boolean. Not on Integers.

Labeled Block: l1{if(true){break l1;}}

Switch: for cases, the value should be constant. It should not be variable. final int x=10; Once defined as final an expression can be written in the cases.

For loop: for(sopln(“Hlo”);;) works fine. There could be any statement inside it and it will work well.

Access Specifiers:

1. Public
2. Private. Accessible only inside class.
3. Protected. Accessible by inherited classes too. They are accessed by inheriting the class, and then accessing it from the subclass.
4. <Package>. Accessible inside the same package. This is default access specifier if not mentioned explicitly.
5. Access modifiers can’t be used on blocks.

Package implicitly available is java.lang package.

Static imports allow us to use methods from packages without always mentioning the class name. import static java.lang.System.\*; out.println() works after this.

If two classes with same name, when calling one of them, need to be qualified fully with the whole package name.

Inheritance:

1. Every class extends the java.lang.Object by default. hashCode() and getClass() are some functions defined there.
2. If a class extends a parent class. Creating an object of the subclass we can access the function of the parent class.
3. Super keyword is like a pointer to the child’s superclass. It calls the constructor of the superclass. It is a reference to the object of the superclass. Super can’t be printed.
4. Not returns object when this is printed. this calls the constructor of that class. So, this is the object of the class. this can be printed.
5. *This* and super can be used together for constructor chaining. They need to be written at the first line of the constructor.
6. When inheriting a class which has a parametrized constructor, the parent class constructor needs to be called explicitly with the arguments from the subclass using super().
7. Same field name in child class and parent class, the child class field is used. This is called shadowing.

Abstraction:

1. Is the process of hiding the internal details of the class so that it is easy to implement the class, because it allows to focus on what the object does rather than how it does.
2. Abstract classes are partially abstract. Some methods may be implemented.
3. Interfaces are concretely abstract.
4. abstract keyword is used with the class or methods.
5. Abstract classes can have only static main, because no instances can be created of it. Interfaces too can have it.
6. Class extending the abstract class needs to override and implement all the abstract methods. It can’t be described as final.
7. They have contract like nature, because they define what a class should do.
8. All methods are public and abstract by default in an interface.
9. Interfaces can’t have constructors, thus we can’t instantiate it, while abstract classes can have constructors it cant be instantiated. But the constructor provided initializes all the fields.
10. Only public methods allowed in an interface. But, after java 8, default and static methods were allowed, allowing concrete implementations. After java 9, private methods were also allowed.
11. Default method in interface allows concrete implementations, and makes it unnecessary to override it.
12. If a class implements two interfaces with same default methods, there is conflict and thus the method must be overridden. Otherwise to access a specific interface, use super. InterfaceA.super.method();
13. Abstract classes can implement interfaces. Allows part of the implementation to be centralized, not all methods need to be implemented, while forcing subclasses to define specific behavior.
14. Marker Interfaces are inbuilt interfaces that don’t have any method, once a class extends them, the class gets some features. Eg: Serializable, Cloneable, RandomAccess.
15. Variables defined inside the interfaces are implicitly constants, with public static final. Thus, initialized when declared.
16. Implementing two interfaces. Both have same method name in it, if function signature same for both methods, no problem. If both methods have different return types, but same method names, then compiler throws an error.
17. If in a class, there is abstract method, then the class needs to be defined as abstract.
18. Multiple Inheritance allowed in interfaces, one class inheritance allowed in abstract class.
19. An interface extends another interface.

Final Keyword:

1. A class marked as final can’t be extended or inherited.
2. If a variable is marked as final, it can’t be changed. It must be initialized either at the first or inside the constructor.
3. If a method is marked as final, it can’t be overridden in a subclass.
4. Final and Abstract can’t be used together because, final prevents extending and abstract asks to be extended.
5. Static final variables need to be initialized at the time of declaration, or inside a static block. Can’t be done in a constructor.

Polymorphism

1. If an object can act differently while talking to different objects, then it is called polymorphism. Poly means many and morphism means shape here behavior.
2. Compile Time or Static Binding. Using Method Overloading. At compile time, the compiler selects and links the method to its method call.
3. Run Time or Dynamic Binding. Using Method Overriding. When multiple classes inherit a class and override a method from the base class.
4. Upcasting. A child object is converted to parent class. This is implicit. Only members of the parent class can be accessed through the object created.
5. Downcasting. A parent object is converted to child class. Done explicitly. May throw exceptions, ClassCastExceptions. Parent p=new Parent; Child c=(Child)p; will throw this exception.
6. Parent p=new Child() parent reference to a child object. This is upcasting. Converting back to child reference is downcasting. Child c=(Child)p; is correct.
7. If exact function not found, the compiler auto promotes the variable to bigger. Eg: show(float) is defined and show(2) is called it will work, but if show(char) is defined, then it will not work, and compiler will throw an error.
8. Suppose show(string) and show(Object) is created. String will be binded, even though string is subclass of object class. If show(string) is removed, object version will be binded. Java prefers specificity.
9. Variables are linked in compile time. Thus Parent c1=new Child(); c1.s will print parent. But, if c1.show() is called it will print child, as it will be run time linked.
10. Variables are accessed by the reference type not the object type. Thus Parent c=new Child(). C.var will access parent’s variable.
11. Main methods can be overloaded, but the one with String[] args is run. main(10) will run the main method with int args.
12. Main methods can be overridden. If subclass main method is not declared, the parents class main method is run when the subclass is run.

Encapsulation

1. The practice of binding data(fields) and code(methods) together within a class, and restricting access to it from outside. Through encapsulation data is bound tightly to the logic that manipulates it, using private fields, access is provided by public methods.
2. Security, we need getter and setter methods, to access and change the variables. Easy maintainability, as any change in the underlying code need not reflect in the calling method if function prototype need not change.

Exception Handling:

1. Compile Time Error: if we don’t follow rules of the programming language.
2. Run Time Error: causes abnormal termination of a program. So, resources are not closed properly, and unfriendly message is shown to the user, who can’t make sense out of it. Exception in java is a class.
3. If an exception occurs, JVM will create an object of that and throw the exception. The exception propagates the call stack, and when it reaches the main method, even then if it is not caught, jvm outputs an error report.
4. Logical Error: what we want the code to do it does not.
5. Arithmetic exception(division by zero), Number Format Exception(trying to convert from string to a number but the string can’t be converted, 123abc cant be converted to integer), ArrayIndexOutOfBound Exception(accessing an index bigger than the array length), NullPointer Exception(when we try to access an object’s methods without creating its reference).
6. The exception and error class in java falls under Throwable class(under java.lang package). Error class can’t be handled by our program. Some errors like NoSuchMethodError, OutOfMemoryError, NoClassDefinitionFound, StackOverflowError, ThreadDeath(thread is forcibly terminated).
7. All classes that directly inherit Exception class are called checked exceptions. These exceptions are checked at compile time. They are out of programs control, like database access. They need to be handled using throws or try catch block which is ensured by the compiler. Eg: IOException, SQLException. The program will give error if compiled.
8. All classes under Runtime Exception are called unchecked exceptions. These exceptions are checked at runtime. These exceptions highlight programming errors, logical bugs or improper use of APIs. No error when compiled, error under certain conditions.
9. Checked exceptions directly inherit Exception class, and are thrown at compile time. Unchecked exceptions inherit Runtime exception. Meaning they throw exception in runtime, and in certain cases.
10. Handling Exceptions: try, catch, throw, finally, throws.
11. try catch block handles the exception where it occurs. A try block can be followed by multiple catch blocks. finally followed by try-catch always executes, connections, inputStreams, outputStreams are closed here.
12. Some methods of exception class:
    1. toString()
    2. getMessage()
    3. printStackTrace()
13. throws delegates the exception handling of a method to the method which is calling it. If no method handles it, finally the JVM default exception handler handles it.
14. Try catch or throws only handles the implicit exceptions thrown by the JVM.
15. Try can be used without a catch block, then it must be followed by finally block.
16. Throw we can throw exceptions explicitly. Super(msg) shows that msg with the exception.
17. ClassCastException. We can check if object instanceof String then convert object to string. Otherwise, this exception is raised.
18. ExceptionInInitializerError occurs when an exception occurs during the static initialization phase, that is code inside static block gives exception or a static variable is incorrectly initialized.
19. IllegalArgumentException.

Scanner class handles IOException internally abstracting out the details from user, BufferedReader allows direct control over the input streams, therefore, we must handle the IOException manually.

Assertions:

1. Used for testing. assert expression. Expression returns a Boolean value. If true then assertion passes. Otherwise throws AssertionError under java.lang.Error class.
2. It is disabled by default. To enable it, java -ea MyClass, java -esa MyClass(system level assertions).
3. assert (expression):”Returns if expression returns false”.
4. Used with Junit, Spring. No real usage in coding as for now in Java.

Multithreading:

1. By extending Threads class or implementing Runnable Interface.
2. We can do more work by not wasting Processor time.
3. Creating a thread and thread. start(). Starts it. run() method is overridden which defines the task to be done by the thread. It is executed concurrently with the main function which is a different thread.
4. Sleep() method puts the current thread being executed into sleep. It is used when connecting to server where the code needs to wait for the resources to be available.
5. Join() method ensures the thread being called dies before caller thread continues executing. We want to sum uptil n, and display in main method. But, threads execute concurrently. So, main thread must wait for sum to complete it. Join() method will complete the sum thread and then displayed.
6. System.currentTimeMillis() counts time from January 1, 1970.
7. Thread currentThread=Thread.currentThread(); currentThread.getName(); currentThread.setName(“Example”);
8. Thread.setPriority() sets priority of execution for thread. The default priority is 5. Maximum priority is 10, minimum priority is 1. No guarantee though that JVM will follow this.
9. If implementing Runnable interface, Thread t=new Thread(instance of the runnable implementation/instance of custom exception class), a thread needs to be created on the object of that class, then t.start().
10. Runnable is preferred because implementing it we can also extend other classes.
11. Yield() temporarily pauses thread execution to allow other threads of the same priority to execute. The thread is moved from running state to ready state. Scheduler decides when it resumes back or which thread is started after it is paused.
12. Interrupt() interrupts the thread, when thread is in sleep/wait mode or blocking state interrupting it returns InterruptedException. When in running state, the interrupted flag of the thread changes to true. The flag can be reset using Thread.interrupted(), isInterrupted() checks the flag.
13. Synchronized methods.
    1. Adding this keyword in method initialization ensures no two threads can simultaneously access that and any other synchronized methods in that object. This prevents data corruption when multiple threads try to modify the object.
    2. When a thread accesses it, it acquires a lock on the object for non static methods whereas it acquires class level lock for static synchronized methods, which is released only after complete execution of the method.
    3. Synchronized block allows code above to be executed, and other threads wait when a lock is acquired. This improves code performance. Done by synchronized(this){}, synchronized(x){} where x is the object reference no thread will be able to execute the code inside synchronized block of that object, synchronized(ClassName.class) to acquire a class level lock, no other thread can execute static synchronized block of that class. This block is defined inside the method code.
14. Interthread Communication, multiple threads working together communicate to each other using wait(), notify(), notifyAll(). Once wait() is executed on a thread, the thread might not move to the wait state at that moment and take some time. But notify() when executed, the thread in wait state moves to the Runnable State and then to the Running State depending on the scheduler.
15. Thread Groups, can be used to collectively handle all the threads under one umbrella. Every thread group directly or indirectly comes under System. System has thread groups named main, finalizer, signalDispatcher etc. When we create threads, it comes under the main thread.
    1. ThreadGroup parent=new ThreadGroup(“Parent”); or new ThreadGroup(parent, “Child”);
    2. getName() returns name of the thread group.
    3. getParent() returns the parent of the thread.
    4. Threads .getPriority() returns the priority of the thread group.
    5. setMaxPriority() sets the maximum priority of the thread group.
    6. .activeCount() gives the number of active threads in a thread group.
    7. .list() gives description of the active threads.
16. Daemon Threads run in the background and assist any other active thread. Eg: Garbage Collector, which is run by the JVM will clean the memory that objects are not using.
    1. A thread can be set as daemon by invoking setDaemon on it. Any thread must be set as daemon before it is running, otherwise IllegalThreadStateException is thrown. Thus, main thread can’t be made daemon as when program starts it starts running.
    2. The isDaemon checks if a thread is daemon.
    3. A daemon thread’s child is also daemon.
    4. The main thread does not wait for the daemon thread to finish.

Garbage Collector

1. The threads have a stack where the local variables are stored.
2. The objects are stored in a heap space where it is allocated memory.
3. The GC removes the objects that are unreachable.
4. System.gc() we can request to run the gc, not necessary that gc will run.
5. Finalize() method is run just before the object is garbage collected.

Inner classes

1. Static inner classes with static methods can be accessed Outer.Inner.Method().
2. Static inner classes with non static methods can be accessed Outer.Inner inner=new Outer.Inner(); inner.method();
3. Static class name{} is the inner class.
4. Non Static inner class. Outer outer=new Outer(); Outer.Inner inner=outer.new Inner();
5. To access variable with same name as declared inside inner class. Outer.this.var used.
6. Local Inner Classes are the ones created inside a method. They can not have access specifiers.
7. Anonymous Runnable Class. Thread t=new Thread(new Runnable(){public void run(){ } }); t.start();

String

1. String s=””; s=new String(“”/char c[]/byte b[]);
2. String is a subclass of Object.
3. It is immutable. Java stores strings in a heap memory called String Pool. Thus, two different string but with same values point to the same memory location. Also, this ensures thread safety. As, if a thread manipulates the value, then a new string is created and other thread reading the string is safe.
4. == compares the memory location.
5. .equals() compares the contents of the string.
6. New String(“”) will create a new string even if that content is present in the string pool.
7. equals() is overridden in string to compare the value, otherwise it compares the object memory location in object class.
8. str.split(“”) returns an array of string.
9. str.replace(“a”,”b”). str a is replaced with b character.
10. StringBuffer initial capacity is 16. sb.capacity(); sb.insert(); sb.delete(start,end);
11. StringBuffer is synchronized whereas StringBuilder is not.

I/O Streams:

1. ByteStreams, CharacterStreams, BufferedStreams, ObjectStreams. All falls under java.io.
2. BufferedStreams are wrappers over the other streams to read more data.
3. FileInputStream f=new FileInputStream(new File(“location”)); while((i=f.read())!=-1) sysout((char)i);
4. FileInputStream and File throws FileNotFoundException which falls under IOException.
5. FileOutputStream fos=new FileOutputStream(“”); Uses fos.write(data).
6. InputStream and OutputStream best for binary data.
7. Readers and Writers for text data. Handles encoding automatically making it suitable for character data.
8. StringTokenizer splits a string, but does not returns an array.
   1. Instead it works on the original string and uses pointers to keep track of the position of tokens.
   2. It is lightweight and straight forward.
   3. Has methods like nextToken() and hasMoreTokens().
   4. It is sequential in nature and all the tokens can’t be accessed at once.
   5. Using BufferedReader we can read lines at a time, otherwise we read one byte at a time.
9. Try with resource block, try(file readers are declared and initialized here){}. JVM automatically closes the streams and we don’t have to do so in finally.
10. Serialization:
    1. Writing objects to a stream is called Serialization, fileStream or outputStream where multiple computers are exchanging information.
    2. Class should implement Serializable. It is a marker interface and has no methods to override.
    3. Sensitive information should be marked as transient int ssn, then it won’t be shared.
    4. ObjectOutputStream is for serialization and ObjectInputStream is for deserializaion.
11. Scanner.hasNext() functions accept input as long as it is of the given type. hasNextInt will continue accepting int values and if any other type found it will return Boolean false.

Collections framework

1. Collections interface is implemented by three interfaces namely:
   1. List. Allows duplicate objects.
      1. ArrayList
      2. LinkedList
      3. Vector is synchronized and thus not used in multithreading.
   2. Set. Does not allow duplicate objects.
      1. HashSet stores the elements randomly does not maintain order.
         1. LinkedHashSet stores elements in the order of arrival.
      2. SortedSet
         1. NavigableSet
            1. TreeSet stores elements in ascending or descending order.
   3. Queue. FIFO.
      1. PriorityQueue
      2. LinkedList implements both Queue and List.
2. Map is not a subclass of collection interface.
   1. HashMap
      1. LinkedHashMap
   2. SortedMap
      1. TreeMap
   3. HashTable. Similar to HashMap but methods are synchronized.
3. Using generics, we can restrict the type of data that goes into the Collection. Like List<Integer>l=new ArrayList< >(); List l=new ArrayList(); is also correct but all types of datatypes can be inserted into it. Since List is the interface implemented by the ArrayList class we create object of that type.
4. ArrayList
   1. It stores the objects in an array. Providing great access time. But is slow when multiple insertions are made causing the list to reshuffle.
   2. Default size is 10. If size exceeds new arrayList is created with 1.5 times size of arrayList and the elements of the old arrayList is copied.
   3. Objects are stored in it. Primitive types if stored it is autoboxed to the object.
   4. List.add(element) and List.add(index,element).
   5. List.set(index,element) replaces the element.
   6. List.addAll(index,Collection) adds a collection to a list.
   7. List.contains(element) returns Boolean.
   8. List.get(index) returns the element at that index.
   9. List.remove(index) removes element at index from list.
5. LinkedList
   1. Faster Insetion and deletions as the nodes of the linkedlist are stored randomly in the heap memory so no need to reshuffle like arraylist.
   2. Slow access because of random storage, arraylist has fast access time.
   3. More memory because a node has prev,val and next.
6. Random
   1. Random ob=new Random();
   2. Ob.nextInt() will return random integer.
   3. Ob.nextInt(value) will return random integer with value as the maximum bound.
7. Iterator
   1. Iterator<Integer> itr=list.iterator();
   2. Itr.hasNext() inside of while
   3. Itr.remove() removes the element.
   4. Itr.next() returns the element.
8. TreeSet
   1. Orders the elements inside it.
9. ListIterator<> l=list.listIterator();
   1. While(l.hasPrevious()) print(l.previous())
   2. While(l.hasNext()) print(l.next())
   3. It can only work on List collections.
   4. L.add()
   5. L.set()
   6. L.nextIndex(), L.previousIndex() for index retrieval.
   7. While Iterator works on all collections it can only traverse forward, and only supports remove().
10. Comparable
    1. TreeSet or TreeMap decides which element comes before which using this interface.
    2. Our class should implement the Comparable class and override the compareTo()
    3. O1.compareTo(o2).
       1. +ve value if o1 has to come after o2.
       2. -ve value if o1 has to come before o2.
       3. 0 if o1 and o2 are same.
    4. Comes under java.lang package.
    5. This defines the natural order of sorting.
    6. Very tight bound and the ordering can’t be changed as the sorting mechanism is defined inside the class.
11. Comparator
    1. Loosely bound, and the comparison logic can be dynamically changed.
    2. StringBuffer needs to implement compare(ob1,ob2) the logic is same as comparable. Because stringBuffer compareTo has not been defined.
    3. The Comparator needs to be passed like new TreeSet<>(Comparator);
    4. It needs to be implemented in a different class.
    5. Sorting can be done as needed.
12. Maps.

Lambda:

1. Lambda is an anonymous function that does not have any name, return type.
2. Introduces Functional Programming.
3. Lambdas can be passed as parameters to other methods.
4. An interface with only one abstract method is functionalInterface.
5. @FunctionalInterface is an annotation that can be added to interfaces after Java8.
6. Interface obj=()->sopln(); obj.method();
7. Runnable Interface example in lambda.
   1. Runnable r=()->{ statements that were written inside void run() are written here};
   2. To start this, need to create a thread. Thread t=new Thread(r); t.start();
   3. Thread t=new Thread(()->{…}); t.start();
8. Comparator<Integer> comp=(i1,i2)->i1.compareTo(i2);

Diamond Problem where two classes have same methods, and now when one class extends them, it does not know which method of which class it should call. So the class extending both must provide its own implementation.

## Streams

1. List l2=l1.stream().filter(i->i%2==0).collect(Collectors.toList());
2. Streams is a powerful way to process and manipulate data collections, such as arrays or list, using a functional programming style.
3. l.stream().map(s->s.toUpperCase()).collect(Collectors.toList());
4. Collect(Collectors.toList()/.toMap()/.toSet()) is used to collect the elements from the stream.
5. Stream.sorted(a comparator can also be passed here) sorts the stream.
6. .forEach(a lambda exp) will apply this expression to each element in the stream. It is generally used with side effects. It does not generate a new list of the modified elements.
7. System.out::println prints all the values of a list.

Optional. To overcome nullpointerexception related issues.

1. Optional<String> emp=Optional.empty(); is a method that returns an empty Optional instance. It represents null value but is more type-safe than using null. It prevents NullPointerException.
2. .orElse(“”); prints it when the optional is empty.
3. .isPresent(); Boolean value if data is present.
4. .of(“”) is used to insert data to it.
5. .get(). If empty throws exception. NoSuchElementException. Otherwise returns the elements of Optional.