**All below 3 are ‘has-A’ while inheritance is ‘is-A’ relationship**

1. Association: how they know each other, one to one, one to many..lifecycle maybe connected/affected.
2. Aggregation: weakest- lifecycle of container class and dependent may differ, can exist independent. Eg: Dept & employee
3. Composition: strongest eg: University & Dept , autowiring or using another object as property is example of composition.

**How to increase JVM memory**

**:** java -Xms256m -Xmx2048m

**Object Class:**

* hashcode()
* equals()
* .toString()- can override to give , how to print Java object

**HashCode & Equals:**

**The .*equals()* Contract**

**Java SE defines the contract that our implementation of the *equals()* method must fulfill**. In short, most criteria follow common sense but we can define the formal rules that the *equals()* method must follow. It must be:

* *reflexive*: an object must equal itself
* *symmetric*: *x.equals(y)* must return the same result as *y.equals(x)*
* *transitive*: if *x.equals(y)* and *y.equals(z),* then also *x.equals(z)*
* *consistent*: the value of .*equals()* should change only if a property that is contained in .*equals()* changes (no randomness allowed)

**Java SE also defines a contract for the .*hashCode()* method**. A thorough look at this contract reveals how closely related .*hashCode()* and .*equals()* are.

All three criteria in the .*hashCode()* contract mention the .*equals()* method in some way**:**

* *internal consistency*: the value of *hashCode()* may only change if a property that is in *equals()* changes
* *equals consistency*: objects that are equal to each other must return the same hashCode
* *collisions*: unequal objects may have the same hashCode

**Generally, we want to override either both *.equals() and .hashCode()* or neither of them.**

**JVM, JRE and JDK:**

* **JDK** It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development
* **JRE** consists of the *Java Virtual Machine (JVM), core classes*, and *supporting files*.
* **JVM** is responsible for executing the java program line by line, hence it is also known as an [**i*nterpreter***](https://www.geeksforgeeks.org/compiler-vs-interpreter-2/)

**OOPs :**

1. Pillars of OOPs
   * [Abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2)
   * [Encapsulation](https://www.geeksforgeeks.org/encapsulation-in-java)
   * [Inheritance](https://www.geeksforgeeks.org/inheritance-in-java)
   * [Polymorphism](https://www.geeksforgeeks.org/polymorphism-in-java)
     + Compile-time polymorphism -overloading
     + Runtime polymorphism -overriding

**SOLID Principles:** refer Baeldung

**Heap vs Stack:**

* Stack is smaller temporary memory, **thread safe**(accessed by one thread at a time), limited to execution context of method.
* Memory allocation and de-allocation are faster
* error Java. lang.[StackOverFlowError](https://www.geeksforgeeks.org/stackoverflowerror-in-java-with-examples/)
* Heap is permanent memory, relevant to application context
* [java. lang.OutOfMemoryError](https://www.geeksforgeeks.org/understanding-outofmemoryerror-exception-java/)

**Class:**

* Inner Class
* Anonymous Inner Class

**Java 8 feature:**

* Java 8 Interface Default methods Diamond Problem

-- implementing class must reimplement method else compilation error.

* Optional Class=> to facilitate NullCheck

.of(), .ofNullable()

**String** nullName = null;

**String** name = Optional.ofNullable(nullName).orElse("john");

* Stream API
* Date/TimeAPI
* *MetaSpace* instead of *PermGen*

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

* MetaSpace grows automatically by default.
* Functional Interface and Lambda Expression.

**Shallow Copy vs Deep Copy:**

Obj1=obj2; // shallow copy

For deep Copy , we can override the clone() method (of Object Class)Cloneable interface.

**Cloneable**interface is implemented by a class to make [Object.clone()](https://www.geeksforgeeks.org/clone-method-in-java-2/) method valid thereby making field-for-field copy.

**Marker Interface:**  interface having no fields or methods eg: Cloneable and Serializable.

**Access Modifiers:**

* Public: everywhere
* Private: only within class
* Protected: within pkg, outside pkg only in subclasses.
* Default: only within pkg.

**Abstract Class vs. Interface:** extends/implements

* A class can inherit one class, can implement multiple interfaces i.e. interface helps to achieve **multiple inheritance**.

Interface Rules:

* [constants variables](https://www.baeldung.com/java-final)
* [abstract methods](https://www.baeldung.com/java-abstract-class)
* [static methods](https://www.baeldung.com/java-static-default-methods) and default- can be given implementation after java 8
* we can’t instantiate interfaces directly
* an interface can be empty, with no methods or variables in it
* we can’t use the *final* word in the interface definition, as it will result in a compiler error
* all interface declarations should have the *public* or default access modifier; the *abstract* modifier will be added automatically by the compiler
* an interface method can’t be *protected* or *final*
* up until Java 9, interface methods could not be *private*; however, **Java 9** introduced the possibility to define [**private methods in interfaces**](https://www.baeldung.com/java-interface-private-methods)
* interface variables are *public*, *static*, and *final* by definition; we’re not allowed to change their visibility

**Multithreading:**

* Extends Thread Class, override Run method Or Implement Runnable Interface
* **Volatile Keyword**- improves **Visibilty**- change made by one thread should be immediately visible to other , though it doesn’t ensure **atomicity**
* **synchronised** keyword-

-Instance methods, Static methods & code blocks

- Ensures both visibility and atomicity- hard lock.

* Thread Pool: Fixed and cached

1. Fixed : fixed number of threads created irrespective of usage, It  will keep all the threads running until they are explicitly terminated
2. Cached: Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

Threads that have not been used for sixty seconds are terminated and removed from the cache

**Exception Handling:**

* Try, catch, finally
* Multiple catch(), Catch can have multiple parameters.
* Throw() from try {}
* Throws with method declaration- overriding method also need to declare
* Checked(FileNotFound, IOException, SQLException) , Unchecked (NullPointer, ArithmeticException) Exception : checked must be declared as throws or should be handled in try-catch

**Serialisation:**

* Object->stream-> use case to transfer object over the network.
* **Serial Version UID** : The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization
  + --user may give or can be generated default also --- UID Mismatch in serialization and deserialization -> **InvalidClassException**
* **Transient** Keyword: When applied to a variable, it instructs the Java Virtual Machine (JVM) to exclude that variable from the serialization process.
* **Static Variables:** These variables are not serialized, So during deserialization static variable value will loaded from the class.(Current value will be loaded.)

**String :**

* String is a literal as well as Class
* String in Java that is specified as immutable, as **the content shared storage in a single pool to minimize creating a copy of the same** value.
* **String Constant pool**: dedicated space in heap for String Constants.

-- String s1=”ABC”;

-- String s2=”ABC”; ->>> s1 & s2 point to same literal in SCP, no new space allocated.

* == Compares values, .equals() compares references

Eg: String s3=new String (“ABC”);

String s4= new String (“ABC”);

s1==s2 ->> false;

s3==s4->> false

s3.equals(s4)->> true

s3.equals(s2)->> true

* String str=”ABC”;

Sop(str.concat(“DEF”)); -> ABC

Str=str.concat(“DEF”)); -> sop(str) -> ABCDEF

**StringBuilder and StringBuffer:** provide mutability of string with methods like append()

* StringBuilder is not threadsafe, no-synchronisation and hence faster

**Static Keyword:**

* **declare a field static, exactly a single copy of that field is created and shared among all instances of that class.**
* **static variables are stored in the heap memory.**
* **access static fields without object initialization.**
* *static* methods in Java are resolved at compile time. Since method overriding is part of Runtime Polymorphism,***static* methods can’t be overridden.**
* **Abstract methods can’t be static.**
* *static* methods**can’t use *this* or *super* keywords.**
* instance methods can directly access both instance methods and instance variables
* instance methods can also access *static* variables and *static* methods directly
* *static* methods can access all *static* variables and other *static* methods
* ***static* methods can’t access instance variables and instance methods directly.** They need some object reference to do so.

**Final Keyword:**

* Can be class, variable or method
* Immutability
* Final class can’t be inherited, method cant overridden, variables cant be reassigned.

**API Stream:** refer Geeks

* Java 8 feature – to handle & process collections
* Method chaining, clean code
* .map(), .filter(), .sorted()
* .collect(), .forEach(), .reduce()

**java.lang.Comparable -** .compareTo() method

* use case:

List<Emp> list =new ArrayList<>();

*Collections.sort(list);* -> error ->logic needs to be given for sorting -> by Id, name , DoB-> how??

* Emp class needs to implement Comparable and need to override compareTo() -> will give one logic at a time;

**java.util.Comparator Interface:** .compare() method

* For multiple logics of sorting use comaparator.
* Collections.sort(list, IdComaparator());
* *Collections.sort(list, NameComaparator());*

**Java Generics:**

**Functional Interface-**having only one abstract method eg: Comparator Interface, Runnable Interface

**Lambda function:**

**Garbage Collection:** Automatic in JAVA, explicit System.gc().

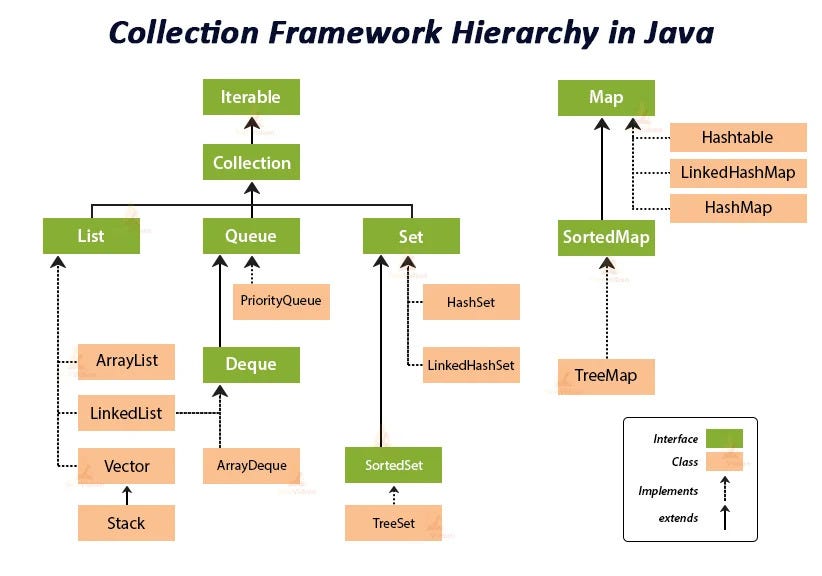
1. **Serial GC :** as the name suggests is serial i.e. to say it uses only 1 thread to get the STW(Stop The World) work done. It cannot take advantage of multiprocessor hardware.
2. **Parallel:**
3. **GarbageFirst (G1):** G1 GC is the default algorithm used in Java9 and beyond. G1 focuses equally on throughput and latency.

The G1 GC traces the old generational memory and finds out all the live/active objects currently being used by application (It uses table and different Data Structures to get this information ). Now since G1 GC knows the live objects currently active, so the scope of memory in old generation to be checked for garbage collection reduces by a lot as the space which has active objects can’t be cleared or garbage collected.

*The regions that contain the most garbage are collected first. Young memory is checked first, and then old memory excluding the active/live object memory is checked. Hence the name: garbage-first collection.*

**Collections:**

* Collections Hierarchy



Synchronized collections ensure thread safety by locking the entire collection for each operation, while concurrent collections achieve thread safety by allowing multiple threads to access different parts of the collection concurrently and hence giving better performance.

* **List:** resizable array
* **Set** : It is an **unordered** **collection** of objects in which **duplicate values cannot be stored**. It is an interface that implements the mathematical **set**. LinkedHashset is ordered.
* **Queue:** It is an **ordered** list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the **FIFO**

--implemented by LinkedList Class, PriorityQueue Class and ArrayDequeue Class

* **Map:** (doesn’t implement Collection interface)
* Hashmap, hashset, treemap
* **ArrayList vs Linkedlist**

AL stores one type of data, Storing and accessing the data is -> better in AL, Implements List Interface

LL stores any type of data, Manipulation/Deletion of data -> LL as AL needs bit shifting, Implememts List and Dequeue Interface

**Implementation of HashMap:**

* Uses HashCode()-> gives hashcode
* Hashcode helps to find index in bucket : index = hashCode(key) & (n-1)
* Hash Collision: If same hashcode, keys are checked with .equals()
  + keys same replace the old key value == > so **no duplicate keys ( but can have null key).**
  + if keys not same, element is added at the tail of index with colliding hashcode
* Collisions occur when multiple keys hash to the same index. To manage collisions, each bucket in the array is actually a linked list. New key-value pairs are added to the list at the corresponding index. Retrieval involves traversing the list to find the correct key.
* In situations where a bucket’s linked list becomes too long, Java 8 replaces it with a balanced tree (TreeNode)

**Implementation of HashSet:**

It internally creates a [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) object with element you have specified as it’s key and constant Object called “PRESENT” as it’s value.

**Implementation of Treemap:**

**ReSizing of HashMap:**

Every time an insertion crosses the load factor of 0.75 the capacity is doubled for a constant time performance of get() { retrieval } and put() { insertion } operations.

Initial Size :16

**ReSizing of ArrayList:**

Initial Size :10

Every time an insertion crosses the load factor of 0.75 the capacity is doubled.

Elements are copied to a new array of double size.

**Miscellaneous:**

1. **Pair**
2. **Record :** simple “immutable & final java data carrier” class with final methods
3. **Var Keyword**
4. **Sealed Classes:** a sealed class is a technique that limits the number of classes that can inherit the given class.

**Initialisation:**

**List<Integer> list = new ArrayList<>( Arrays.asList(1, 2, 3));**

**int[] arr = { 1, 2, 3, 4, 5 };**

* Arrays.asList()
* Arrays.sorted()
* Map Traverse:

   HashMap<String, Integer> hm

            = **new** HashMap<String, Integer>();

        // Adding mappings to above HashMap

        // using put() method

        hm.put("GeeksforGeeks", 54);

        hm.put("A computer portal", 80);

**for** (Map.Entry<String,Integer> mapElement : hm.entrySet())

* Set Traverse:

   HashSet<String> h = **new** HashSet<String>();

        // Adding elements into HashSet using add()

        h.add("Geeks");

        h.add("for");

        h.add("Geeks");

        // Iterating over hash set items

**for** (String i : h)

            System.out.println(i);

**PLSQL:**

* **Nth highest:**

SELECT salary

FROM employee

ORDER BY salary desc limit n-1,1

Limit offset (rows to ignore), rows to print

Eg: 2nd highest – ignore first row the highest, and print one row.

* **Today’ Date:**

SELECT GETDATE();

* **Joins:**

Inner Join= Common

Left/Right= Common +left/right

Cross = total-> common+left+right

* **Oreder by multiple column:**

SELECT \* FROM Customers ORDER BY Country, CustomerName;

This means that it orders by Country, but if some rows have the same Country, it orders them by CustomerName

* **DDL & DML:**

**DDL-** CREATE, ALTER, DROP and TRUNCATE.

**DML-** insert, update, delete.

Project Learning:

ProAdmin-3rd party app MSI, Silent Installation for AMI automation- w/o automation, you cant attach Ec2 and no scaling –only standalone Ec2 .

Performance testing Compatibility- S3 Storage gateway slower than Fsx file share path.