**Java Q&A**

**A class** is a collection of objects. Classes don’t consume any space in the memory**.**

**An object** is areal world entity which have properties and functionalities. Object is also called an instance of class. Objects take some space in memory.

**Public Modifiers:** means that class, variable or method is accessible throughout from within or outside the class, within or outside the package, etc. It provides highest level of accessibility.

**Private Modifiers:** means that class, variable or method is not accessible from within or outside the class, within or outside the package, etc. Private field or method can't be inherited to sub class. This provides lowest level of accessibility**.**

**Protected Modifiers:** means that class, variable or method is accessible from classes in the same package, sub-classes in the same package, subclasses in other packages but not accessible from classes in other packages.

**Inheritance is** the procedure in which one class inherits the attributes and methods of another class. In other words It is a mechanism of acquiring properties or behaviors of existing class to a new class.

**Method overloading:** When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments multiple methods of same names performs different tasks within the same class.

**Method Overriding:** on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. methods having same name which can have different functionalities. That base function is said to be overridden.

**Constructor** is a special type of member function which is used to initialize an object. It is similar as functions but it's name should be same as its class name and must have no explicit return type.

**Encapsulation** is the process of binding data and methods in a single unit. In encapsulation, data(variables) are declared as private, and methods are declared as public.

**Abstraction:** Allows hiding unnecessary data from the user. This reduces program complexity efforts. it displays only the necessary information to the user and hides all the internal background details.

Ex: -phone call we don’t know the internal processing

We can achieve data abstraction by using

1. Abstract class 2. Interface

**What is an abstract class?**

Abstract class is that class which contains abstract method. Abstract methods are those methods which have only declaration not the implementation. An abstract class is declared with abstract keyword. An abstract class can also contain non-abstract methods.

**Inheritance:**

Inheritance is the procedure in which one class inherits the attributes and methods of another class. In other words It is a mechanism of acquiring properties or behaviors of existing class to a new class

**Polymorphism in Java** is a concept by which we can perform a single action in different ways.

**String**: Use when you don't need to modify the string or when you only need to create new strings (e.g., constants or strings passed between methods).

**StringBuffer**: Use when you need a mutable string in a **multi-threaded** environment and require thread safety.

**StringBuilder**: Use when you need a mutable string in a **single-threaded** environment and need better performance for frequent string modifications.

**Collection** is a root interface in the Java Collections Framework that defines common behaviors for collections like add(), remove(), contains(), etc.

**Collections** is a utility class that provides **static methods** to manipulate and work with collections (e.g., sorting, reversing, etc.).

**ArrayList** is ideal for use cases where you frequently access elements by index and where insertions/removals are mainly at the end of the list.

**LinkedList** is ideal for use cases where you frequently insert or remove elements at the beginning or middle of the list and don’t need fast random access by index.

**List** and **Set** are both **collections** in Java, but they have different characteristics:

**List:**

* **Order**: A List is **ordered**, meaning elements are stored in the order they are inserted.
* **Duplicates**: A List allows **duplicate elements**. The same element can appear multiple times.
* **Indexing**: Elements in a List can be accessed using an **index** (e.g., list.get(index)).
* **Common Implementations**: ArrayList, LinkedList, Vector, etc.

**Set:**

* **Order**: A Set does **not maintain order** (except for LinkedHashSet and TreeSet, which maintain insertion or sorted order).
* **Duplicates**: A Set **does not allow duplicates**. If you try to add the same element again, it will not be inserted.
* **Indexing**: A Set does **not** provide access by index.
* **Common Implementations**: HashSet, LinkedHashSet, TreeSet.

**Set** and **Map** are different types of collections:

**Set:**

* **Structure**: A Set is a **collection of unique elements** (no duplicates).
* **Access**: You can only store individual values (elements) in a Set.
* **No Key-Value Pairs**: A Set does not store key-value pairs.

**Map:**

* **Structure**: A Map is a **collection of key-value pairs**, where each key maps to a corresponding value.
* **Access**: You can retrieve values using keys (e.g., map.get(key)).
* **No Duplicate Keys**: A Map does not allow duplicate keys, but it can have duplicate values

\*Both **HashMap** and **HashSet** are backed by hash tables, but they serve different purposes.

**HashMap:**

* **Structure**: A HashMap is a **key-value pair** collection.
* **Access**: You use keys to retrieve values (e.g., map.get(key)).
* **Duplicates**: A HashMap does not allow **duplicate keys**, but it allows **duplicate values**.
* **Performance**: It provides **constant-time performance** for get(), put(), and remove() operations (O(1) on average).

**HashSet:**

* **Structure**: A HashSet is a **Set**, so it stores unique elements only.
* **Access**: It only stores **values**, not key-value pairs.
* **Duplicates**: A HashSet does not allow duplicate elements.
* **Performance**: Like HashMap, it provides **constant-time performance** for basic operations.

**HashMap:**

* **Thread Safety**: HashMap is **not synchronized** and is not thread-safe.
* **Null Keys/Values**: HashMap allows **one null key** and **multiple null values**.
* **Performance**: HashMap generally performs better because it is not synchronized, which makes it faster in single-threaded environments.

**Hashtable:**

* **Thread Safety**: Hashtable is **synchronized**, meaning it is thread-safe. Multiple threads can safely access and modify the Hashtable.
* **Null Keys/Values**: Hashtable does not allow **null keys** or **null values**. It throws a NullPointerException if you try to insert a null key or value.
* **Performance**: Due to synchronization, Hashtable tends to be slower than HashMap in single-threaded environments.

|  |  |  |
| --- | --- | --- |
| **Summary of Differences** |  |  |
| **Pair** | **Feature** | **List of Key Differences** |
| **List vs Set** | **Order, Duplicates** | List maintains order and allows duplicates; Set does not allow duplicates and might not maintain order. |
| **Set vs Map** | **Structure** | Set stores unique elements; Map stores key-value pairs. |
| **HashMap vs HashSet** | **Structure** | HashMap stores key-value pairs; HashSet stores unique values (elements). |
| **Set vs Map** | **Structure** | Set stores values; Map stores key-value pairs. |
| **HashMap vs Hashtable** | **Thread Safety, Null Values, Performance** | HashMap is not synchronized, allows null keys/values, faster; Hashtable is synchronized, does not allow nulls, slower. |