**OOPS Concepts**

**Class:**

* In java, Class is a blueprint or template that defines the properties (fields) and behaviours (methods) common to a group of objects.
* It serves as a model for creating objects with specific attributes and functionalities.
* Class is a blueprint for creating objects in Java.

**Object:**

* An object is an instance of a class that represents a real-world entity. It encapsulates data (fields) and behaviour (methods) defined in its class, allowing manipulation and interaction within a program.
* objects are instances of classes.

**Encapsulation:**

* Encapsulation is the concept of bundling data (fields) and methods (behaviours) within a class, where data is accessed and modified only through methods, ensuring data integrity and providing control over access levels (public, private, protected).
* Encapsulation is the process of bundling data and methods within a class to restrict access and ensure data integrity.

**This keyword:**

* **This** keyword refers to the current object in a method or constructor.

**Constructors:**

* A constructor in Java is a special method that is used to initialize objects.
* The constructor is called when an object of a class is created.
* It can be used to set initial values for object attributes

**Access Modifiers:**

* Controls the access level for classes, attributes, methods and constructors.
* Public – Accessed by other classes. (Class)
* Default – If we don’t provide anything, default is used. It provides access to class in the same package. (Class).

**For Attributes, methods and constructors**

* Public - The code is accessible for all classes
* Private- The code is only accessible within the declared class
* default - The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter
* protected-The code is accessible in the same package and subclasses

**Non-Access Modifiers:**

**Classes**

* Final - The class cannot be inherited by other classes
* Abstract - The class cannot be used to create objects (To access an abstract class, it must be inherited from another class.

**For attributes and methods**

* Final- Attributes and methods cannot be overridden/modified
* Static- Attributes and methods belongs to the class, rather than an object
* Abstract- Can only be used in an abstract class and can only be used on methods. The method does not have a body.
* Transient- Attributes and methods are skipped when serializing the object containing them
* Synchronized- Methods can only be accessed by one thread at a time
* Volatile- The value of an attribute is not cached thread-locally, and is always read from the "main memory"

**Inheritance:**

* In Java, it is possible to inherit attributes and methods from one class to another.
* It is useful for code reusability: reuse attributes and methods of an existing class when you create a new class.
* The done using **extends** keyword.
* Superclass – parent class from which we can inherit
* Subclass/child class - the class that inherits from another class.

**Polymorphism:**

* Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.
* With method overloading, multiple methods can have the same name with different parameters.
* The **print ()** method is also an example of polymorphism. It is used to print values of different types like char, int, string, etc.

**Method Overriding:**

* If the same method is present in both the superclass and the subclass. Then, the method in the subclass overrides the same method in the superclass. This is called method overriding

**Method Overloading:**

* we can create methods with the same name if they differ in parameters.

**Java Super:**

* The super keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods).

**Data Abstraction:**

* Data abstraction is the process of hiding certain details and showing only essential information to the user.
* Abstraction can be achieved with either abstract classes or interfaces.

**Abstract Method:**

* A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods.

**THREADS**

**Synchronization:**

* Synchronization in Java is the capability to control the access of multiple threads to any shared resource.
* Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Thread.currentThread().getName() – To get thread information

**Synchronized Method:**

* Locks the instance (object) on which the method is called, allowing only one thread to execute the synchronized method on that instance at a time

**Synchronized Block:**

* Allows for more granular synchronization by locking a specific block of code within a method, typically using synchronized(this) or a shared object to control access to critical sections.

**Static Synchronized:**

* Synchronizes a static method or block at the class level, preventing concurrent execution by multiple threads across all instances of the class by locking the class's monitor.

**wait ():**

* Causes the current thread to wait until another thread calls notify () or notifyAll() on the same object.

**notify ():**

* Wakes up one waiting thread that is waiting on the same object, allowing it to continue execution.

**notifyAll():**

* Wakes up all waiting threads that are waiting on the same object, allowing them to compete for the object's lock and resume execution.

**FILES**

* **createNewFile()** – To create new file
* **Scanner** – To read the file contents
* **getName()** – To get name of the file
* **Write()** – To write inti the file
* **FileWriter** – used along with write to write into the file.
* **Delete()** – to delete the file.

**BINARY FILES**

Binary format refers to representing data using the binary number system, which uses only two digits: 0 and 1. In computer systems, data is typically stored, processed, and transmitted in binary format because it directly corresponds to the on-off states of electronic switches in digital circuits, making it efficient for electronic devices to work with.

* **FileOutputStream:** Used for writing bytes to a file. It's part of Java's IO package (java.io).
* **FileInputStream:** Used for reading bytes from a file. Also part of Java's IO package (java.io).
* **ObjectOutputStream:** Used for writing serialized Java objects to an output stream (e.g., a file). It's used with FileOutputStream to save objects to a file.
* **ObjectInputStream:** Used for reading serialized Java objects from an input stream (e.g., a file). It's used with FileInputStream to read objects from a file.
* **FileOutputStream** writes bytes to a file.
* **FileInputStream** reads bytes from a file.
* **ObjectOutputStream** writes serialized Java objects to a stream. It "serializes" the objects, which means it converts them into a format that can be saved to a file.
* **ObjectInputStream** reads serialized Java objects from a stream

**Text File vs Binary File**

* A text file is a type of file that contains human-readable characters, such as letters, numbers, and symbols.
* Text files store data using character encoding, such as ASCII or Unicode, where each character corresponds to a specific binary value.
* binary file is a type of file that contains data in binary format, represented as sequences of 0s and 1s.
* Binary files store data in its raw binary form, without encoding characters directly. They are used for storing non-textual data, such as images, audio, video, or serialized Java objects.

**serialization and De-serialization**:

* Serialization in Java is the process of converting an object into a byte stream for storing in file or transport over network. while deserialization is the reverse process of reconstructing an object from a byte stream.

**transient**

* The transient keyword in Java is used to indicate that a variable should not be serialized when an object is serialized.

**OBJECT CLASS**

The object class is the parent class of all classes in Java by default. It has 11 methods

**Clone()** –

* To create exact copy of an object. When we create object with new it will take lot of processing time. Using clone, it will be faster.
* It should implement Cloneable interface and throw CloneNotSupportedException.

**toString()**

* Converts object to string.

**hashCode()**

* For every object, JVM generates a unique number which is a hashcode. It returns distinct integers for distinct objects.

**finalize()**

* This method is called just before an object is garbage collected. It is called the Garbage Collector on an object when the garbage collector determines that there are no more references to the object.

getClass()

The getClass() method in Java returns the runtime class of an object, which is an instance of the Class class representing the object's type."

1. **equals(Object obj):** Compares two objects for equality.
2. **hashCode():** Returns a hash code value for the object.
3. **toString():** Returns a string representation of the object.
4. **getClass():** Returns the runtime class of the object.
5. **notify():** Wakes up a single thread waiting on this object's monitor.
6. **notifyAll():** Wakes up all threads waiting on this object's monitor.
7. **wait():** Causes the current thread to wait until another thread notifies the calling object.
8. **wait(long timeout):** Causes the current thread to wait until another thread notifies the calling object or a specified amount of time elapses.
9. **wait(long timeout, int nanos):** Causes the current thread to wait until another thread notifies the calling object or a specified amount of time elapses, with nanosecond precision.
10. **finalize():** Called by the garbage collector before the object is reclaimed to perform clean-up.
11. **clone():** Creates and returns a copy of the object

**Reflection API:**

Reflection is an API that is used to examine or modify the behaviour of methods, classes, and interfaces at runtime.

**String vs String Buffer**

**1.Mutability:**

**String**:

* Immutable: Once created, String objects cannot be changed. Any modification creates a new String object.

**StringBuffer:**

* Mutable: StringBuffer objects can be changed after creation. Modifications are made directly to the existing object.

Example:

**2. Thread Safety:**

**String**:

* Thread-safe: String objects are inherently thread-safe due to their immutability. They can be shared across multiple threads without synchronization.

**StringBuffer:**

* Thread-safe: Methods in StringBuffer are synchronized, making it safe to use in multi-threaded environments. Only one thread can execute a method at a time on a particular instance.

**3. Performance:**

**String:**

* Performance Issues with Modifications: Modifying a String creates new objects, leading to potential performance overhead if many modifications are performed.

**StringBuffer:**

* Better Performance for Modifications: StringBuffer is more efficient for extensive string manipulations as it modifies the existing object without creating new ones.

**4. Usage Scenarios:**

**String**:

* Constant Strings: Use String for fixed sequences of characters that do not change.

**StringBuffer**

* Strings with Frequent Modifications: Use StringBuffer for strings that undergo frequent changes, especially in a multi-threaded environment.

Example:

**5. Memory Usage:**

**String:**

* Higher Memory Usage for Modifications: Since each modification creates a new String object, memory usage can be higher and may lead to more frequent garbage collection.

**StringBuffer:**

* Lower Memory Usage for Modifications: Modifications to a StringBuffer do not create new objects, resulting in lower memory usage and less strain on the garbage collector.

**StringBuffer vs StringBuilder**

**1. Mutability:**

**StringBuffer**:

* Mutable: StringBuffer objects can be changed after creation. Modifications are made directly to the existing object.

**StringBuilder**:

* Mutable: Similar to StringBuffer, StringBuilder objects can also be changed after creation.

2**. Thread Safety:**

**StringBuffer**:

* Thread-safe: Methods in StringBuffer are synchronized, which means they are safe to use in multi-threaded environments. Only one thread can execute a method at a time on a particular instance.

**StringBuilder**:

* Not Thread-safe: Methods in StringBuilder are not synchronized, meaning they are not safe for use in multi-threaded environments without additional synchronization.

3. **Performance**:

**StringBuffer**:

* Slower Performance: Due to synchronization, StringBuffer has an overhead, making it slightly slower than StringBuilder in single-threaded environments.

**StringBuilder**:

* Faster Performance: Without synchronization overhead, StringBuilder is faster and preferred in single-threaded environments.

**4. Usage Scenarios:**

**StringBuffer**:

* Multi-threaded Environment: Use StringBuffer when working with strings in a multi-threaded environment to ensure thread safety.

**StringBuilder**:

* Single-threaded Environment: Use StringBuilder when working with strings in a single-threaded environment for better performance.

**5. Method Synchronization:**

**StringBuffer**:

* Synchronized Methods: Each method in StringBuffer is synchronized, making it thread-safe but with additional overhead.

**StringBuilder**:

* Non-Synchronized Methods: Methods in StringBuilder are not synchronized, which avoids the synchronization overhead.

**StringBuffer and StringBuilder important methods**

1. append(String s): Appends the specified string to this string buffer.
2. insert(int offset, String s): Inserts the specified string at the specified position in this string buffer.
3. replace(int startIndex, int endIndex, String str): Replaces the characters in a substring of this string buffer with characters in the specified string.
4. delete(int startIndex, int endIndex): Deletes the characters in a substring of this string buffer.
5. reverse(): Reverses the sequence of characters in this string buffer.
6. capacity(): Returns the current capacity of the string buffer.
7. ensureCapacity(int minimumCapacity): Ensures that the capacity is at least equal to the specified minimum capacity.
8. charAt(int index): Returns the character at the specified position in this string buffer.
9. length(): Returns the length (number of characters) of this string buffer.
10. substring(int beginIndex): Returns a new string that is a substring of this string buffer, starting from the specified begin index.
11. substring(int beginIndex, int endIndex): Returns a new string that is a substring of this string buffer, starting from the specified begin index and extending to the character at index end index - 1.

**COLLECTIONA FRAMEWORK**

**Iterator:**

* The Iterator interface in Java is a part of the java.util package
* Provides a way to traverse through a collection of objects one by one.
* It is used to retrieve elements from a collection, such as a List or Set, without exposing the underlying structure of the collection
* Iterator interface provides the facility of iterating the elements in a forward direction only.

**Methods:**

**hasNext()**

* Returns true if there are more elements in the collection to iterate over.
* Returns false if there are no more elements.

**next()**

* Returns the next element in the collection.
* Throws a **NoSuchElementException** if there are no more elements.

**remove()**

* Removes the last element returned by the iterator from the underlying collection.
* Can only be called once per call to next().
* Throws an **IllegalStateException** if next() has not been called before remove(), or if remove() has already been called after the last next().

**Iterable Interface:**

* The Iterable interface is the root interface for all the collection classes.
* The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
* Present in Java.lang package

**Collection Interface:**

* The Collection interface is the interface which is implemented by all the classes in the collection framework.
* It declares the methods that every collection will have.
* Basic methods, add, addAll, clear

**List Interface:**

* list type data structure in which we can store the ordered collection of objects. It can have duplicate values.\
* List interface has ArrayList, LinkedList, Vector and stack ( Subclass of vector)

**Queue Interface:**

* Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed.
* There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

**Set Interface:**

* Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items.
* We can store at most one null value in Set.
* Set is implemented by HashSet, LinkedHashSet, and TreeSet.
* HashSet – stored in hash table by suing hashing technique, Unique vales.
* LinkedHashSet – Uses Linked implemntion and Hashsset, Stores unique values.
* Treeset: Stores unique values, searching is fast, stored in increasing order.

**SortedSet Interface:**

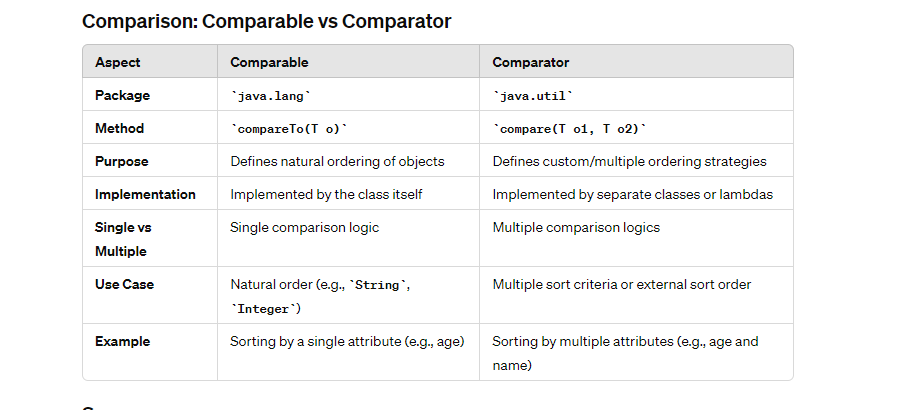
* SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order.

**Comparator interface:** (Multiple Criteria and custom ordering)

* The Comparator interface in Java is part of the java.util package and is used to define a custom order for objects. It provides a way to compare two objects or more to determine their ordering with respect to each other.

**Comparable interface:** (Single Attribute Ordering and Natural ordering)

* The Comparable interface is part of the java.lang package. It imposes a natural ordering on the objects of each class that implements it. This ordering is typically based on a single attribute of the objects.



**Java Map Interface**

* A map contains values based on key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.
* A Map is useful if you have to search, update or delete elements on the basis of a key.
* A Map doesn't allow duplicate keys

**HashMap**

* HashMap is the implementation of Map, but it doesn't maintain any order.

**LinkedHashMap**

* LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.

**TreeMap**

* TreeMap is the implementation of Map and SortedMap. It maintains ascending order

**Enumeration:**

* The Enumeration interface in Java is part of the java.util package and is used to iterate over a collection of objects, such as vectors and hashtables.
* Unlike Iterator, remove () is not possible. Read only.

Vector:

add vs addElement

Some are same, which add elements to the vector. Add method is method from list interface and addElement is method specific to vector class.

The size is added by one, when we add addelement.

Add is Boolean, addelement is void return type

Capacity in vector

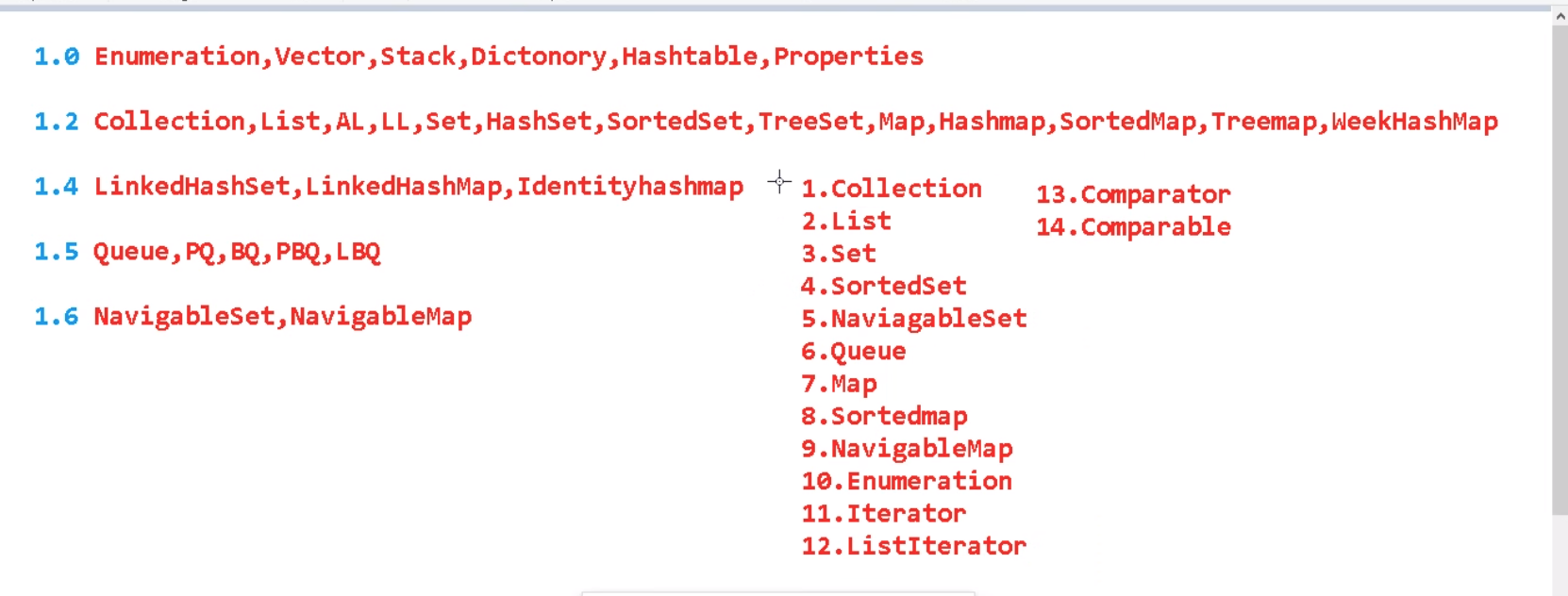
Thestart adding elements to the Vector.

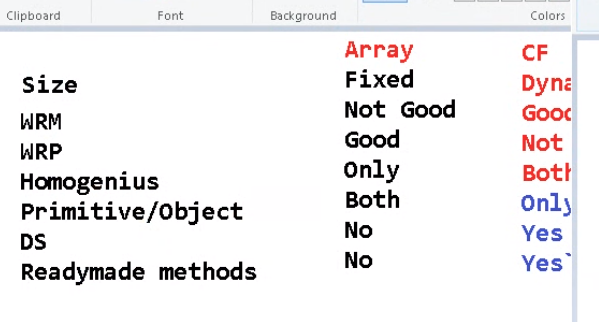
After adding the 4th element, the capacity is full.

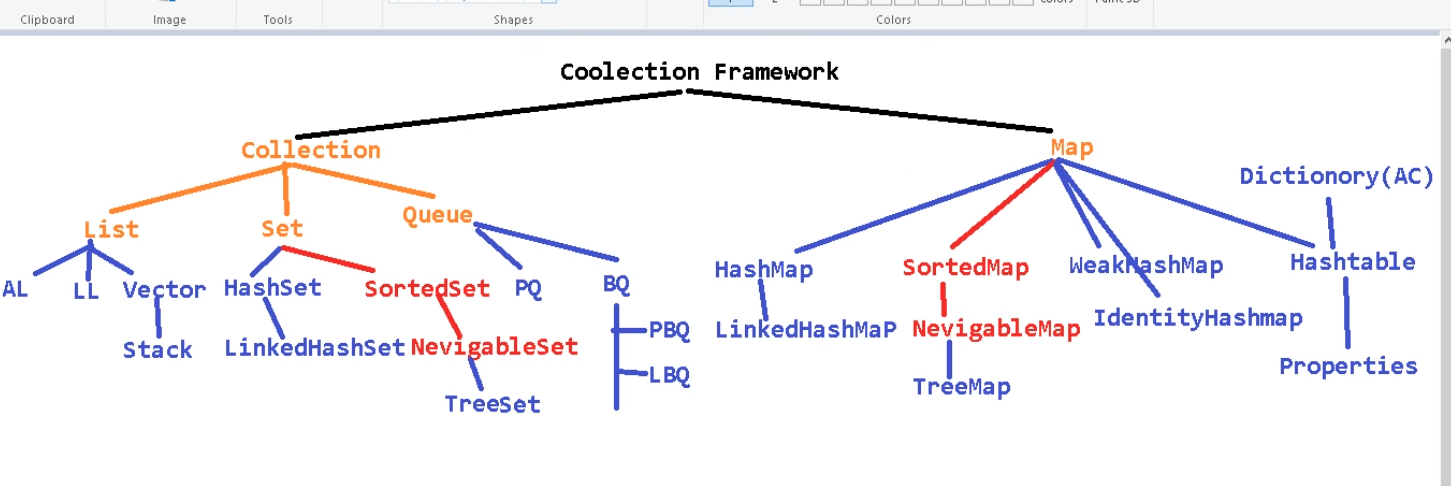
When you add the 5th element, the Vector exceeds its current capacity and thus, it doubles its capacity from 4 to 8.

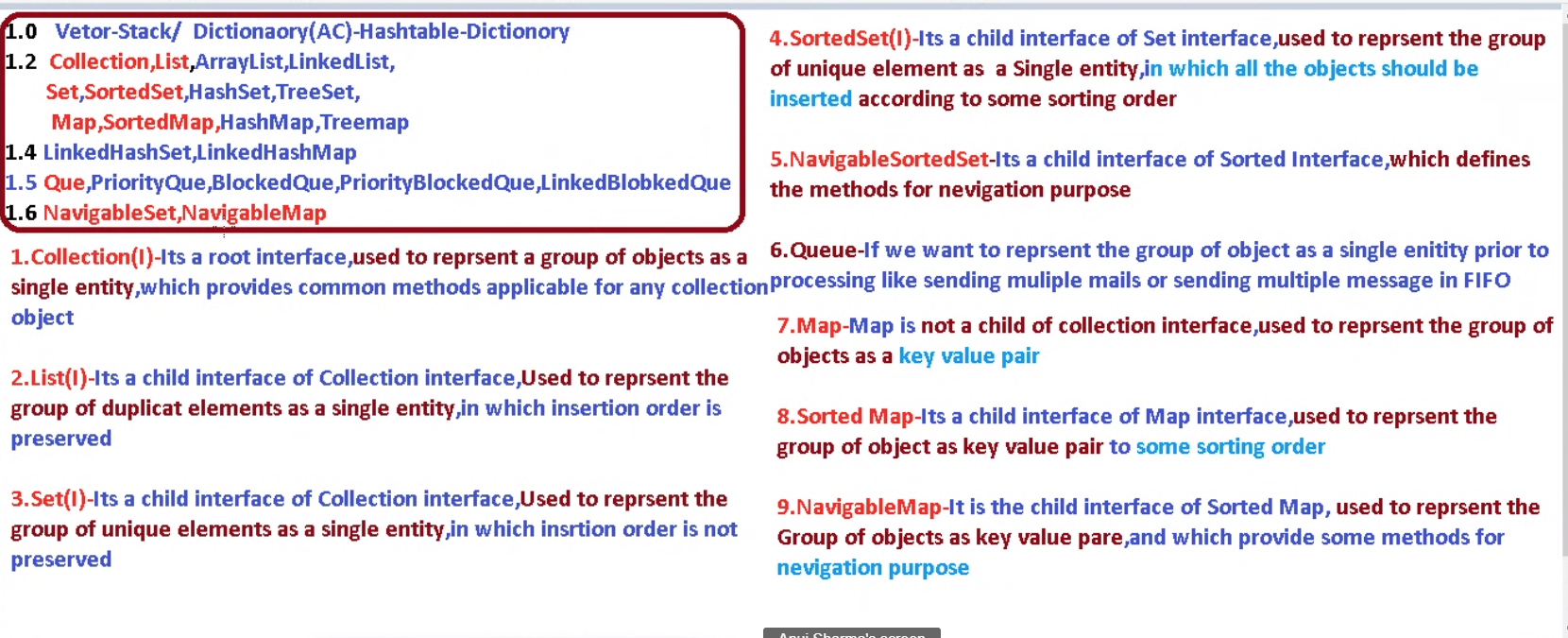
Size: The number of elements currently in the Vector.

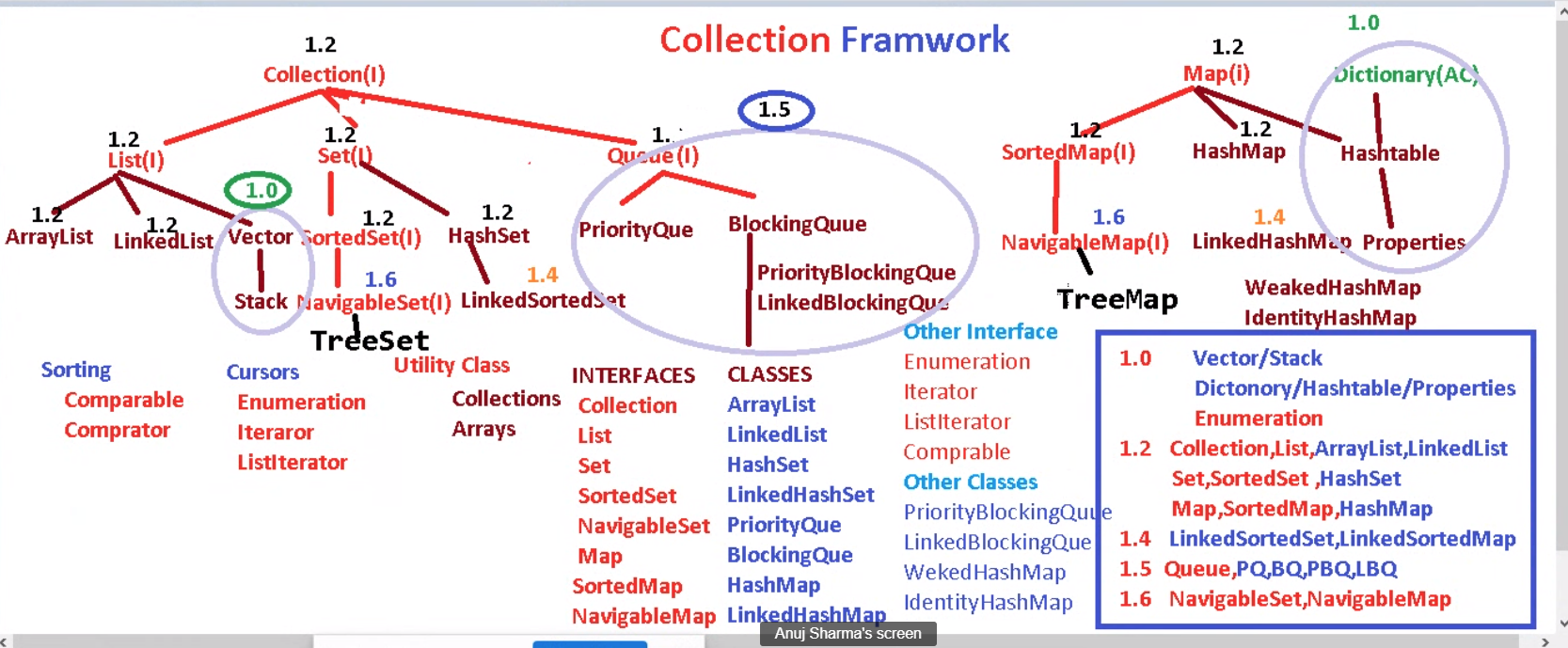
Capacity: The total number of elements the Vector can hold













**Exception Handling**

**Checked Exceptions**

* **IOException:** Occurs when an I/O operation fails or is interrupted.
* **SQLException:** Indicates a database access error.
* **ClassNotFoundException:** Thrown when an application tries to load a class through its name but no definition for the class with the specified name could be found.
* **FileNotFoundException:** Raised when a file with the specified pathname does not exist.
* **InterruptedException:** Occurs when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted.
* **NoSuchMethodException:** Indicates a particular method cannot be found.
* **InvocationTargetException:** Thrown by an invoked method or constructor when it throws an exception.

**Unchecked Exceptions**

* **NullPointerException:** Occurs when trying to access an object through a null reference.
* **ArrayIndexOutOfBoundsException:** Thrown when an array is accessed with an illegal index.
* **ArithmeticException:** Raised when an exceptional arithmetic condition occurs, such as division by zero.
* **ClassCastException:** Occurs when trying to cast an object to a subclass of which it is not an instance.
* **IllegalArgumentException:** Thrown to indicate that a method has been passed an illegal or inappropriate argument.
* **NumberFormatException:** Occurs when an attempt is made to convert a string to a numeric type but the string does not have an appropriate format.
* **IllegalStateException:** Indicates a method has been invoked at an illegal or inappropriate time.

**Differences of all child classes in collection framework**

**HashMap vs. Vector**

* **Underlying Data Structure:** HashMap uses a hash table; Vector uses a dynamic array.
* **Order Maintained:** HashMap does not maintain order; Vector maintains insertion order.
* **Synchronized**: HashMap is not synchronized; Vector is synchronized and thus thread-safe.
* **Use Case:** HashMap is used for fast key-value pair lookups. Vector is used when you need a thread-safe list that allows random access.

**HashMap vs. LinkedHashMap**

* **Underlying Data Structure:** Both use hash tables, but LinkedHashMap also uses a linked list to maintain insertion order.
* **Order Maintained:** HashMap does not maintain order; LinkedHashMap maintains insertion order.
* **Performance:** Both have similar average time complexity for get/put operations, but LinkedHashMap has a slight overhead due to maintaining the linked list.
* **Use Case:** HashMap is used when order does not matter. LinkedHashMap is used when you need a key-value pair with fast lookup and predictable iteration order.

**Vector vs. LinkedHashMap**

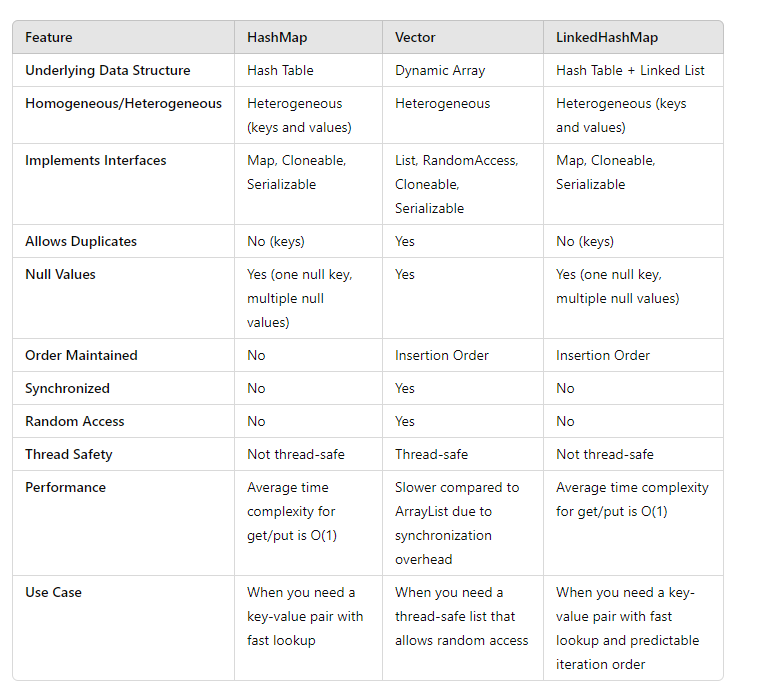
* **Underlying Data Structure:** Vector uses a dynamic array; LinkedHashMap uses a hash table with a linked list.
* **Data Type:** Both can hold heterogeneous data.
* **Order Maintained:** Vector maintains insertion order; LinkedHashMap maintains insertion order.
* **Synchronized:** Vector is synchronized (thread-safe); LinkedHashMap is not synchronized (not thread-safe).
* **Use Case:** Vector is used for thread-safe lists that allow random access. LinkedHashMap is used for maintaining a key-value pair with predictable iteration order and fast lookup.

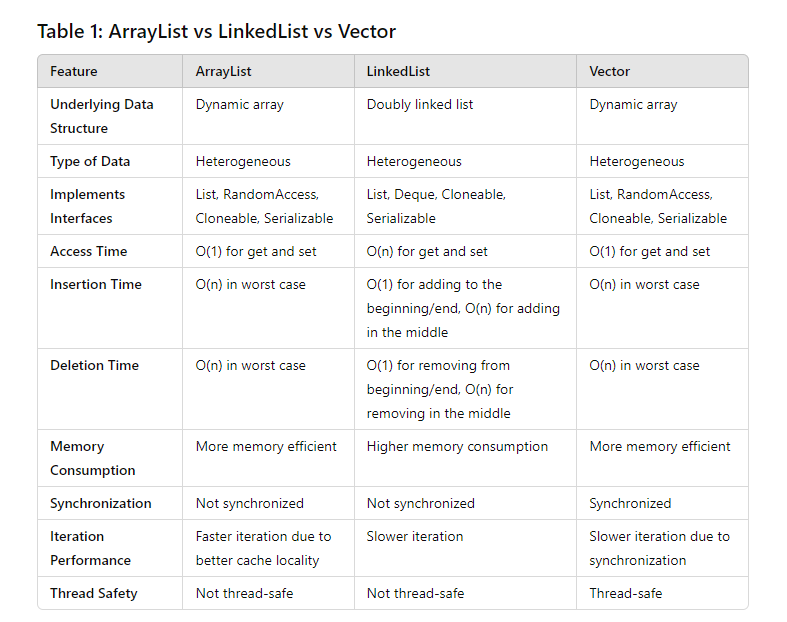
**Summary**

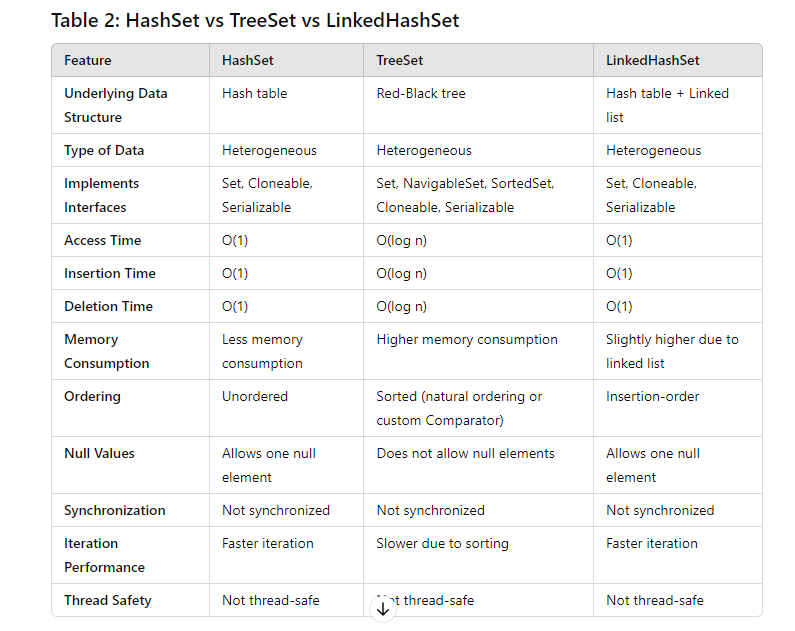
**HashMap**: Ideal for fast key-value lookups where order does not matter, not synchronized.

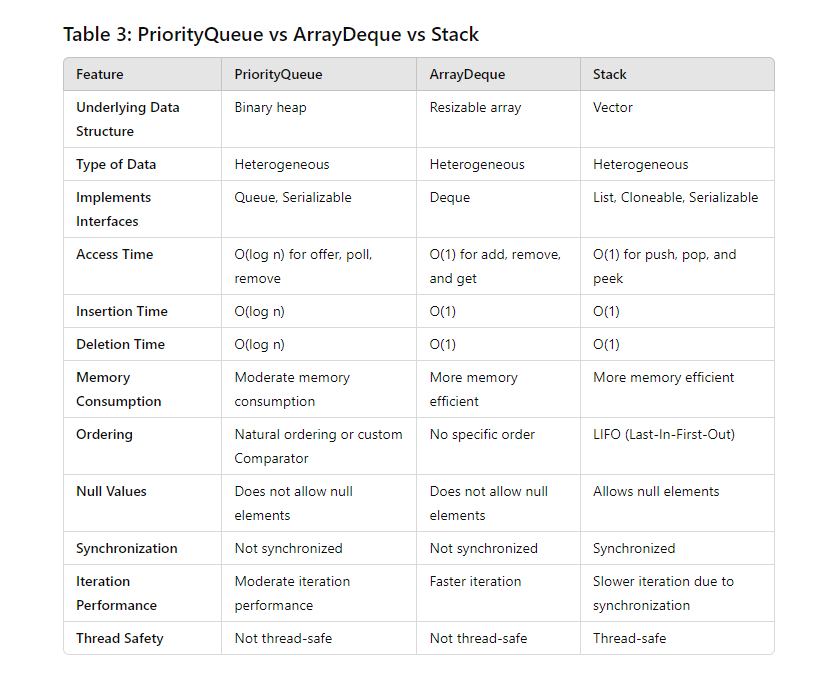
**Vector**: Ideal for thread-safe lists that require random access and maintain insertion order, synchronized.

**LinkedHashMap**: Ideal for fast key-value lookups where order matters, maintains insertion order, not synchronized.









* **PriorityQueue** - Homogeneous- Elements must be comparable to maintain the heap property.
* **TreeSet**- Homogeneous -Elements must be comparable to maintain order in the Red-Black tree.
* **TreeMap**- Homogeneous (for keys) - Keys must be comparable to maintain order in the Red-Black tree.
* **ArrayDeque**- Homogenous

**Iterator:**

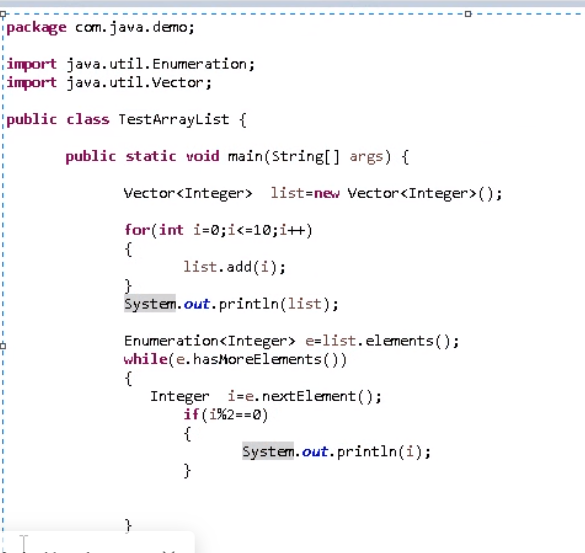
* Supports all collections.
* One-way traversal.
* Can remove elements during iteration.
* Introduced in Java 1.2.

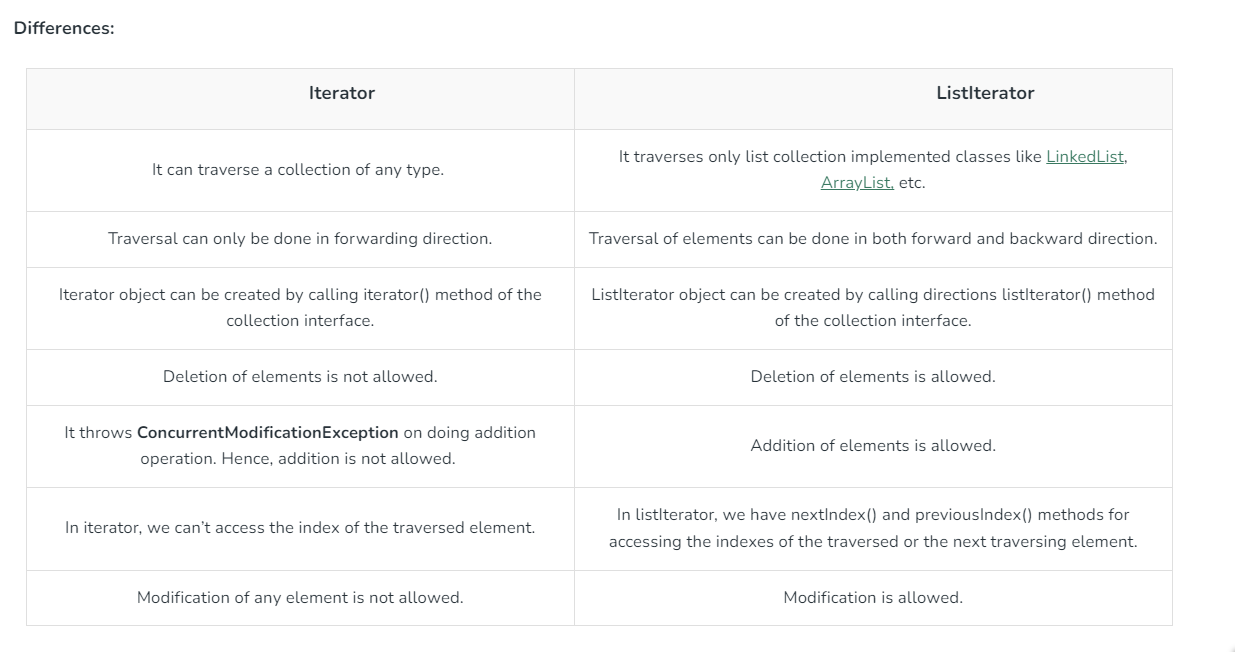
**ListIterator:**

* Extends from iterator
* Supports only lists.
* Two-way traversal.
* Can modify elements and add new elements during iteration.
* Provides methods to get indices of elements.
* Introduced in Java 1.2.

**Enumeration:**

* Supports legacy collections like Vector and Hashtable.
* One-way traversal.
* Cannot remove elements.
* Introduced in Java 1.0.
* Less functional than Iterator.





Collections Class

Sort, binary search, shuffle, reverse, unmodifiableList

The Arrays class provides utility methods for array manipulation, such as sorting and searching:

Sort, binary search, array to (list List<String> list = Arrays.asList("A", "B", "C")

JAVA 8

Java 8 released on 2014 by oracle

Features are:

1. Lambda expressions,
2. Method references,
3. Functional interfaces,
4. Stream API,
5. Default methods,
6. Base64 Encode Decode,
7. Static methods in interface,
8. Optional class,
9. Collectors class,
10. ForEach() method,
11. Nashorn JavaScript Engine,
12. Parallel Array Sorting,
13. Type and Repating Annotations,
14. IO Enhancements,
15. Concurrency Enhancements,
16. JDBC Enhancements

These features will improve **code conciseness, readability, and performance, facilitating modern programming practices**.

**Lambda Expressions**

It is very useful in collection library in which it helps to iterate, filter and extract data. Will provide implementation of an interface which has functional interface.

Lambda expression provides implementation of **functional interface**. An interface which has only one abstract method is called functional interface. Java provides an anotation **@FunctionalInterface**, which is used to declare an interface as functional interface.

Need of Lamdha expressions:

* One of the key benefits of lambda expressions in Java 8 is that they allow you to implement functional interfaces (interfaces with a single abstract method) **more concisely without the need to create a separate class.**
* In traditional Java before lambda expressions, you would need to create a class that **implements the interface, instantiate it**, and **then call its methods**. With lambda expressions, this process is simplified significantly.

**By default, all the methods in interface are abstract. In function interface we can have only one abstract method.**

**Lambda with Runnable Interface (Threads)**

When you use a lambda expression to implement the Runnable interface, you do not need to explicitly use the @Override annotation or the run method signature. The lambda expression implicitly provides the implementation for the single abstract method in the Runnable interface.

**Lamdha with Comparater**

Before Java 8, you would typically use an anonymous inner class to implement the Comparator interface. We have @overide the compare method and implements the comparator interface. With lambda we can easily use the compare abstract method.

**Lambda with Predicate:**

A Predicate is a functional interface that represents a boolean-valued function of **one argument**.

**Predicate** and **BiPredicate** are functional interfaces provided in the java.util.function package in Java. They are used to represent boolean-valued functions of one or two arguments, respectively.

BiPredicate<T, U>: A functional interface that takes **two arguments** and returns a boolean. Useful for conditions or comparisons involving two objects.

**Lambda with Consumer and BiConsumer**

A Consumer is a functional interface that represents an operation that accepts a **single input** argument and returns no result. Its abstract method is void accept (T t).

A BiConsumer is a functional interface that represents an operation that accepts **two input** arguments and returns no result. Its abstract method is void accept (T t, U u).

**Lambda with Supplier<T>:**

Definition: Represents a supplier of results. It is used to generate or supply values without taking any input.

Abstract Method: T get().

Use Case: Commonly used for lazy generation of values, such as random numbers, timestamps, or unique identifiers.

**Lambda with Function and BiFunction:**

A Function is a functional interface that represents a function that takes one argument and returns a result. Its abstract method is R apply (T t).

Example: Converting Strings to Uppercase

**Function<T, R>:**

Definition: Represents a function that accepts **one argument** and produces a result.

Abstract Method: R apply(T t).

Use Case: Commonly used for transforming data, such as converting types, processing strings, or performing calculations.

**BiFunction<T, U, R>:**

Definition: Represents a function that accepts **two arguments** and produces a result.

Abstract Method: R apply (T t, U u).

Use Case: Used for operations that require two inputs, such as combining values, performing binary operations, or concatenating strings.

Array List

Linked List

Hash Set

Linked Hash Set

Tree set

Priority Queue

Blocking Queue

Hash Map

Linked Hash Map

Tree Map

Vector

Stack

Priority Blocking queue

Linked Blocking queue

WeekHash Map

Identity Hash Map

Hash Table

Dictionary

Properties

Collections

Arrays

20 classes, 14 interfaces



Vector program   
package com.atp.demo;

import java.util.Vector;

//ds -growable array

//old/legecy

//thread safe

//peformence wise not good

//default capesity 10

//afer a limit double the capesity

//duplicate allowed

//hetrogenius

//null is allwoed

//ccX3/4+1 (AL)

//ccx2(v)

public class Test {

public static void main(String[] args) {

ArrayL v=new Vector();

System.out.println(v.capacity());//10

for(int i=1;i<=10;i++)

{

v.addElement(i);

}

System.out.println(v.capacity());//10

v.add("kabuter");

System.out.println(v.capacity()); //20

}

}

**Singleton Class in Java**

A singleton class is a class that allows only one instance of itself to be created and provides a global point of access to that instance. This pattern is useful when exactly one object is needed to coordinate actions across the system, such as a shared resource or a configuration manager.

Here's an example of how to create a singleton class in Java:

java

Copy code

public class Singleton {

// Private constructor to prevent instantiation from outside

private Singleton() {}

// Static variable to hold the single instance of the class

private static Singleton instance = null;

// Static method to get the single instance of the class

public static Singleton getInstance() {

// Lazy initialization: Create the instance only if it's null

if (instance == null) {

instance = new Singleton();

}

return instance;

}

// Example method of the singleton class

public void showMessage() {

System.out.println("Hello, I am a singleton class!");

}

}

public class Main {

public static void main(String[] args) {

// Get the singleton instance

Singleton singleton = Singleton.getInstance();

// Call methods on the singleton instance

singleton.showMessage();

}

}

In this example:

The Singleton class has a private constructor to prevent instantiation from outside the class.

It has a private static variable instance to hold the single instance of the class.

The getInstance() method provides a global point of access to the single instance. It uses lazy initialization to create the instance only if it's null.

The Main class demonstrates how to use the singleton class by getting the instance and calling its methods.

**Singleton classes are commonly used in scenarios such as:**

* Database connection management
* Logging frameworks
* Caching mechanisms
* Configuration settings management

However, it's important to note that singleton classes can introduce global state, which can make the code harder to test and maintain. Care should be taken when using the singleton pattern, and alternatives such as dependency injection or factory patterns should be considered when appropriate. Additionally, in multithreaded environments, thread safety needs to be ensured in the singleton implementation to prevent race conditions and ensure correct behavior.

**JAVA 8 Features**

**DEFAULT METHODS IN JAVA 8**

Default methods are methods in an interface that have a default implementation. They are declared with the default keyword.

default String name();

Before Java 8 if we want to create an interface with methods, its implemented classes should also override all the methods. It will be problem because if we want to add any method to that interface we should disturb all the codes to implement that method. But by using default method we can ignore that.

The default method is non-abstract method.

* default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

**Static Method in Java 8**

Java 8 introduced the ability to define static methods in interfaces, which allows related utility methods to be grouped within the interface they are associated with.

Yes, before Java 8, static methods were always defined in classes, not in interfaces. To call a static method, the method itself had to be defined as static in a class. Static methods belong to the class rather than instances of the class, and they can be called without creating an instance of the class.

Yes, starting from **Java 8,** you can provide a **method body for static methods in an interface**. This allows you to define utility or helper methods directly within the interface, making the interface more self-contained and cohesive.

We can directly call these static methods from main method without creating objects. (Similar to Static keyword).

**Important Points:**

* Interfaces can have default methods with implementation in Java 8 on later.
* Interfaces can have static methods as well, similar to static methods in classes.
* Default methods were introduced to provide backward compatibility for old interfaces so that they can have new methods without affecting existing code.

**Method References in Java 8**

The use of an instance method reference of an arbitrary object of a particular type. This type of method reference is useful when you want to call an instance method on any instance of a particular type, rather than a specific instance.

Constructor references are used to instantiate objects by referring to a constructor.