**JAVA ENUM:**

An enum stands for enumeration is a special data type that enables for variable to be a set of predefined constants.

The enum data type is used when we know all possible values at compile time, such as days of the week, colours, deck of cards, etc.

**Program**

package practice;

import java.util.Scanner;

public class JavaEnum{

enum Meals{

SMALL(1,1,1,3),

MEDIUM(2,1,1,4),

LARGE(2,2,1,5);

private int burgers;

private int fries;

private int drink;

private int price;

Meals(int burgers, int fries, int drink, int price) {

this.burgers = burgers;

this.fries = fries;

this.drink = drink;

this.price = price;

}

public int[] getDetails() {

int[] arr = {burgers,fries,drink};

return arr;

}

public int getPrice() {

return price;

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Welcome to our Coffee Shop!");

System.out.println("Available sizes:");

// Display all available coffee sizes

for (Meals size : Meals.values()) {

System.out.printf("%s (burgers-%d,fries-%d,drink-%d) - $%d%n",

size.name(),

size.getDetails()[0],

size.getDetails()[1],

size.getDetails()[2],

size.getPrice());

}

System.out.print("Enter your choice (SMALL/MEDIUM/LARGE): ");

String userChoice = scanner.nextLine().toUpperCase();

try {

Meals selectedSize = Meals.valueOf(userChoice);

System.out.printf("You've selected a %s meal",

selectedSize.name());

System.out.printf("Price: $%d%n", selectedSize.getPrice());

} catch (IllegalArgumentException e) {

System.out.println("Invalid size selected. Please try again.");

}

finally {

scanner.close();

}

}

}

**Cache:**

We create a cache here that will delete the least recent accessed element from our collection when the capacity of our collection is exceeded.

This helps in memory management and acts like an actual cache.

**Program**

package practice;

import java.util.\*;

public class JavaCache{

public static void main(String[] args) {

final int maxCapacity = 10;

final Map<String,String> moviesCache = new LinkedHashMap<String,String>(100,0.75f,true){

protected boolean removeEldestEntry(Map.Entry<String,String> eldest) {

return size()>maxCapacity;

}

};

String[] genres = {"Comedy","Action","Romcom","Sci-fi","Thriller","Horror","Crime"};

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

String movie = "Movie "+i;

moviesCache.put(movie,genres[new Random().nextInt(genres.length)]);

}

System.out.println(moviesCache);

moviesCache.put("Movie 11","Comedy");

System.out.println(moviesCache);

}

}

**Collections:**

**1.List**

- **ArrayList**: A resizable array that allows duplicates and preserves the insertion order.

- **LinkedList**: A doubly linked list that is more efficient for insertion and deletion than ArrayList.

- **Stack**: A last-in, first-out (LIFO) collection where the last element added is the first to be removed.

- **Vector**: A thread-safe, synchronised version of ArrayList.

**2. Set**

- **HashSet**: Stores unique elements in an unordered manner using a hash table.

- **LinkedHashSet**: Preserves the insertion order of elements while maintaining uniqueness.

- **TreeSet**: Stores unique elements in a sorted (ascending) order.

**3. Queue**

- **LinkedList as Queue**: Acts as a FIFO queue, where elements are added at the end and removed from the front.

- **PriorityQueue**: Orders elements based on their natural ordering or a custom comparator.

- **ArrayDeque as Deque**: A double-ended queue that allows elements to be added or removed from both ends.

- **BlockingQueue**: a thread-safe queue that blocks when trying to add elements to a full queue or retrieve from an empty one.

**4. Map**

- **HashMap**: Stores key-value pairs without any guarantee of order.

- **LinkedHashMap**: Maintains the order in which key-value pairs were inserted.

- **TreeMap**: Stores key-value pairs in sorted order based on the keys.

- **Hashtable**: A synchronised, thread-safe map that does not allow null keys or values.

- **ConcurrentHashMap**: a thread-safe implementation of HashMap that allows concurrent read and write operations without locking the entire map, improving performance in multi-threaded environments.

**Thread Safe Collections:**

Thread-safe collections are collections designed to be safely accessed and modified by multiple threads concurrently without causing data corruption or inconsistency. Examples include Vector, Hashtable, ConcurrentHashMap, and BlockingQueue.

**Program**

package practice;

import java.util.\*;

public class JavaCollections{

public static void main(String[] args) {

forList();

forSet();

forQueue();

forMap();

}

private static void forMap() {

System.out.println("====================Types of Maps===========================");

System.out.println(" 1. HashMap\n 2. PriorityQueue\n 3. DeQueue\n 4. BlockingQueue\n");

Map<String, Integer> hashMap = new HashMap<>();

hashMap.put("One", 1);

hashMap.put("Two", 2);

hashMap.put("Three", 3);

System.out.println("HashMap: " + hashMap);

Map<String, String> linkedHashMap = new LinkedHashMap<>();

linkedHashMap.put("USA", "Washington D.C.");

linkedHashMap.put("UK", "London");

linkedHashMap.put("Japan", "Tokyo");

System.out.println("LinkedHashMap: " + linkedHashMap);

Map<String, Double> treeMap = new TreeMap<>();

treeMap.put("Pi", 3.14159);

treeMap.put("Phi", 1.61803);

treeMap.put("e", 2.71828);

System.out.println("TreeMap: " + treeMap);

Hashtable<Integer, String> hashtable = new Hashtable<>();

hashtable.put(1, "First");

hashtable.put(2, "Second");

System.out.println("Hashtable: " + hashtable);

// ConcurrentMap<String, Integer> concurrentMap = new ConcurrentHashMap<>();

// concurrentMap.put("Concurrent", 1);

// concurrentMap.put("Hash", 2);

// concurrentMap.put("Map", 3);

// System.out.println("ConcurrentHashMap: " + concurrentMap);

System.out.println("Keys in HashMap: " + hashMap.keySet());

System.out.println("Values in HashMap: " + hashMap.values());

System.out.println("Entries in HashMap: " + hashMap.entrySet());

}

private static void forQueue() {

System.out.println("====================Types of Queues===========================");

System.out.println(" 1. Queue\n 2. PriorityQueue\n 3. DeQueue\n 4. BlockingQueue\n");

Queue<Integer> llQueue = new LinkedList<>();

llQueue.offer(7);

llQueue.offer(8);

llQueue.offer(9);

System.out.println("Queue: " + llQueue);

System.out.println("Poll from Queue: " + llQueue.poll());

Queue<Integer> PriorityQueue = new PriorityQueue<>();

PriorityQueue.offer(2);

PriorityQueue.offer(2);

PriorityQueue.offer(3);

PriorityQueue.offer(7);

System.out.println("Queue: " + PriorityQueue);

System.out.println("Poll from PriorityQueue: " + PriorityQueue.poll());

Deque<Integer> deque = new ArrayDeque<>();

deque.addFirst(5);

deque.addLast(6);

System.out.println("Deque: " + deque);

System.out.println("Poll first from Deque: " + deque.pollFirst());

System.out.println("Poll last from Deque: " + deque.pollLast());

// BlockingQueue<String> blockingqueue = new LinkedBlockingQueue<>();

// blockingqueue.offer("Task 1");

// blockingqueue.offer("Task 2");

// System.out.println("BlockingQueue: " + blockingqueue);

}

private static void forSet() {

System.out.println("====================Types of Sets===========================");

System.out.println("1. HashSet\n 2. LinkedHashList\n 3. TreeSet\n");

Set<String> hsFruits = new HashSet<>();

hsFruits.add("Apple");

hsFruits.add("Tomato");

hsFruits.add("Kiwi");

System.out.println("HashSet: "+hsFruits);

Set<String> lhsFruits = new LinkedHashSet<>(hsFruits);

lhsFruits.add("Banana");

System.out.println("LinkedHashSet: "+lhsFruits);

Set<String> tsFruits = new TreeSet<>(lhsFruits);

tsFruits.add("Cherry");

System.out.println("TreeSet: "+tsFruits);

Set<String> set1 = new HashSet<>(Arrays.asList("B","c","D"));

Set<String> set2 = new HashSet<>(Arrays.asList("a","B","c"));

set1.retainAll(set2);

System.out.println(set1);

}

private static void forList() {

System.out.println("====================Types of Lists===========================");

System.out.println(" 1. ArrayList\n 2. LinkedList\n 3. Stack\n 4. Vector\n");

List<String> alvehicles = new ArrayList<>();

alvehicles.add("Bike");

alvehicles.add("Car");

alvehicles.add("Aeroplane");

System.out.println("ArrayList of Vehicles is: "+alvehicles);

List<String> llvehicles = new LinkedList<>(alvehicles);

llvehicles.add(1,"Train");

System.out.println("LinkedList of vehicles is: "+llvehicles);

Stack<String> stvehicles = new Stack<>();

stvehicles.push("Bus");

System.out.println("Stack of Vehicles: "+stvehicles);

System.out.println("Popped element from the Stack is: "+stvehicles.pop());

stvehicles.push("Car");

stvehicles.push("Bike");

Vector<String> vctrvehicles = new Vector<>();

vctrvehicles.add("Chopper");

System.out.println("Vector of vehicles: "+vctrvehicles);

Collections.sort(alvehicles);

Collections.sort(llvehicles);

Collections.sort(stvehicles);

Collections.sort(vctrvehicles);

System.out.println(alvehicles+"\n"+llvehicles+"\n"+stvehicles+"\n"+vctrvehicles);

}

}

**THREADS:**

- A Thread is a lightweight sub-process that allows multiple tasks to run concurrently within a program, improving performance by executing code in parallel.

- Threads can be created by either extending the Thread class or implementing the Runnable interface.

- With Thread class we can also write a lambda function to override run function.

- Thread logic should only be written inside run() method.

- **Synchronized Block**

- A synchronised block ensures that only one thread can execute this block at a time.

- It prints the name of the current thread when inside the synchronised block.

- **join()**: The main thread waits for thread1, thread2, and thread3 to finish before proceeding.

**wait-notify**

- **wait()**: Causes the current thread to release the monitor lock and go into a waiting state until another thread calls `notify()` or `notifyAll()` on the same object. The thread waits until it is either notified or interrupted.

- **notify()**: Wakes up a single thread that is waiting on the object's monitor. The awakened thread will continue execution once it regains the monitor lock.

- **notifyAll()**: Wakes up all threads waiting on the object's monitor (those that called `wait()` on the object). Only one of the awakened threads will proceed once it gains the lock, while the others will continue waiting.

- **sleep()**: Pauses the execution of the current thread for a specified duration (in milliseconds or nanoseconds) without releasing any locks it holds. Unlike `wait()`, it does not interact with thread synchronisation.

**Program:**

package practice;

public class JavaThreads{

class MyRunnableThread implements Runnable{

public void run() {

System.out.println(Thread.currentThread().getName()+" is running");

}

}

class MyThreadThread extends Thread{

public void run() {

System.out.println(Thread.currentThread().getName()+" is running");

}

}

public void demonstration() {

MyRunnableThread t1 = new MyRunnableThread();

Thread thread1 = new Thread(t1);

thread1.start();

MyThreadThread t2 = new MyThreadThread();

Thread thread2 = new Thread(t2);

thread2.start();

Thread thread3 = new Thread(()->System.out.println(Thread.currentThread().getName()+" is running"),"lambaThread");

thread3.start();

thread1.setName("ImplementedThread");

thread2.setName("ExtendedThread");

synchronized(this) {

System.out.println(Thread.currentThread().getName()+" in synchronized block");

}

try {

thread1.join();

System.out.println(thread1.getName()+": "+thread1.getState());

thread2.join();

thread3.join();

}

catch(InterruptedException e) {

System.out.println("There is an Interruption");

}

final Object lock = new Object();

Thread waiter = new Thread(()->{

synchronized(lock) {

try {

System.out.println("Waiter is created");

lock.wait();

System.out.println("Waiter is notified");

}

catch(InterruptedException e){

System.out.println("There is an interruption");

}

}

});

Thread notifier = new Thread(()->{

synchronized(lock) {

System.out.println("Notifier is notifying");

lock.notify();

}

});

waiter.start();

try {

Thread.sleep(1000);

}

catch(InterruptedException e) {

System.out.println("There is an exception");

}

notifier.start();

Thread sleeper = new Thread(()->{

try {

System.out.println("Sleeper is running and sent to sleep for 5000 milliseconds");

Thread.sleep(5000);

}

catch(InterruptedException e) {

System.out.println("Sleeper was interrupted");

}

});

sleeper.start();

sleeper.interrupt();

}

public synchronized void SynchronizedMethod() {

System.out.println(Thread.currentThread().getName()+" entered Synchronized Block");

try {

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName()+": "+Thread.currentThread().getState());

}

catch(InterruptedException e){

System.out.println("There is an interruption");

}

System.out.println(Thread.currentThread().getName()+" is exiting the synchronized block");

}

public static void main(String[] args) {

System.out.println("Welcome to Threads");

new JavaThreads().demonstration();

}

}