**========================= SimpleCache =================================**

* This is a manual cache created using HashMap to add elements, delete elements, and get elements from the stored cache.

**Program:**

package practice;

import java.util.\*;

public class SimpleCache<k,v>{

private final Map<k,v> cache;

public SimpleCache() {

this.cache = new HashMap<>();

}

public void put(k key,v value) {

cache.put(key,value);

}

public v get(k key) {

return cache.get(key);

}

public void remove(k key) {

cache.remove(key);

}

public void clear() {

cache.clear();

}

public int size() {

return cache.size();

}

public static void main(String[] args) {

SimpleCache<String, String> cache = new SimpleCache<>();

cache.put("key1", "value1");

cache.put("key2", "value2");

cache.put("key3", "value3");

System.out.println(cache.get("key1"));

cache.remove("key2");

System.out.println(cache.size());

cache.clear();

System.out.println(cache.size());

}

}

**================================= LRUCache============================**

* LRU Cache - Least Recently Used Cached.
* It is used to remove the least recently used element when the cache is filled beyond the given capacity.
* Here we have overridden the removeEldestEntry to check if the capacity is filled, if filled the eldest element is removed.

**Program:**

package practice;

import java.util.\*;

public class PracticingLRUCache<K,V> extends LinkedHashMap<K,V>{

private final int capacity;

public PracticingLRUCache(int capacity) {

super(capacity,0.75f,true);

this.capacity = capacity;

}

@Override

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

return size()>capacity;

}

public static void main(String[] args) {

PracticingLRUCache<String, String> cache = new PracticingLRUCache<>(3);

cache.put("key1", "value1");

cache.put("key2", "value2");

cache.put("key3", "value3");

System.out.println(cache);

cache.get("key2");

cache.put("key4", "value4");

cache.put("key5", "value5");

System.out.println(cache);

}

}

**Synchronized keyword and threads:**

* Synchronized keyword is used to maintain synchronization between the thread itself, i.e., a thread can run parallely to the same thread itself. If we use synchronized keyword the parallel thread for itself is not created and only thread will be run at a time.
* Packages should be mentioned correctly otherwise there will be errors raised.
* The class names should not be the same when created in the same package. Whether it is a main class or any other class but the names should be different.
* When we start threads for execution we cannot control the flow of execution of threads. The threads can run in any way, and that is the real-time use of threads.
* Threads are used for efficiency but consistency is not the priority.
* Volatile keyword is used to fetch data from the memory. Whenever it is used the data fetched will always be the latest data from the memory.

**Program:**

package practice;

class Account{

private volatile int balance=0;

Account(){

balance=0;

}

public synchronized void deposit(int amount){

balance+=amount;

System.out.println("Deposited: "+amount+" New Balance: "+balance);

}

public synchronized void withdraw(int amount){

if(balance>=amount){

balance-=amount;

System.out.println("Withdrawn: "+amount+" New Balance: "+balance);

}else{

System.out.println("Insufficient balance");

}

}

public int getBalance(){

return balance;

}

}

public class BankDemo {

public static void main(String[] args) throws InterruptedException {

System.out.println("Bank Demo");

Account account = new Account();

Thread depositThread = new Thread(() -> {

for(int i=0;i<5;i++){

account.deposit(100);

try{

Thread.sleep(100);

}catch(InterruptedException e){

e.printStackTrace();

}

}

},"Deposit Thread");

Thread withdrawThread = new Thread(() -> {

for(int i=0;i<5;i++){

account.withdraw(80);

try{

Thread.sleep(100);

}catch(InterruptedException e){

e.printStackTrace();

}

}

},"Withdraw Thread");

depositThread.start();

Thread.sleep(100);

withdrawThread.start();

try {

Thread.sleep(1000);

}

catch(InterruptedException e) {

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

}

System.out.println("Final Balance: "+account.getBalance());

}

}

**============================GuavaCaching==============================**

* Guava Cache is a caching library provided by Google's Guava that allows in-memory caching with features like eviction policies, time-based expiration, and size limitations.
* We perform the Cache operations by using the existing library, to increase performance.

**Program:**

package com.mycompany.app;

import com.google.common.cache.\*;

import java.util.Random;

import java.util.concurrent.TimeUnit;

public class GuavaCachingExample {

public static void main(String[] args) {

// Replaced LoadingCache with Cache

Cache<String, String> cache = CacheBuilder.newBuilder()

.maximumSize(200000)

.expireAfterWrite(10, TimeUnit.MINUTES)

.build();

try {

String[] genres = {"Action", "Comedy", "Drama", "Horror", "Romance", "Sci-Fi", "Thriller"};

// Generate 200000 movies with random genres and push into cache

for (int i = 0; i < 200000; i++) {

String movie = "Movie" + i;

String genre = genres[new Random().nextInt(genres.length)];

cache.put(movie, genre);

}

// Fetch movie multiple times to see time taken

for (int i = 0; i < 100; i++) {

long startTime = System.nanoTime();

String movie = cache.getIfPresent("Movie92600");

long endTime = System.nanoTime();

System.out.println("Time taken to fetch the movie: " + (endTime - startTime) + " nanoseconds");

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

**====================== Caching Performance ============================**

* Here we have Product class which contains the details of the product. We create objects for that product class and store them in the database.
* Then we perform the operations on that database and cache.
* Here we also calculate the performance by calculating the time taken by the operations for completion.

**Program:**

package com.mycompany.app;

import java.util.HashMap;

import java.util.Map;

import java.util.Random;

class Product {

private String id;

private String name;

private double price;

public Product(String id, String name, double price) {

this.id = id;

this.name = name;

this.price = price;

}

// Getters and setters omitted for brevity

@Override

public String toString() {

return "Product{id='" + id + "', name='" + name + "', price=" + price + "}";

}

}

class DatabaseSimulator {

private Map<String, Product> products = new HashMap<>();

private Random random = new Random();

public DatabaseSimulator(int numProducts) {

for (int i = 0; i < numProducts; i++) {

String id = "PROD" + i;

products.put(id, new Product(id, "Product " + i, 10 + random.nextDouble() \* 90));

}

}

public Product getProduct(String id) {

// Simulate database access delay

try {

Thread.sleep(100); // 100ms delay to simulate DB access

} catch (InterruptedException e) {

e.printStackTrace();

}

return products.get(id);

}

}

//—--------------------------------Another class—----------------------------------------------

package com.mycompany.app;

import com.google.common.cache.CacheBuilder;

import com.google.common.cache.CacheLoader;

import com.google.common.cache.LoadingCache;

import java.util.concurrent.TimeUnit;

public class ProductService {

private final DatabaseSimulator database;

private final LoadingCache<String, Product> cache;

public ProductService(int numProducts, int cacheSize) {

this.database = new DatabaseSimulator(numProducts);

this.cache = CacheBuilder.newBuilder()

.maximumSize(cacheSize)

.expireAfterWrite(10, TimeUnit.MINUTES)

.recordStats() // This allows us to collect cache statistics

.build(new CacheLoader<String, Product>() {

@Override

public Product load(String id) {

return database.getProduct(id);

}

});

}

public Product getProduct(String id) throws Exception {

return cache.get(id);

}

public void printCacheStats() {

System.out.println("Cache stats: " + cache.stats());

}

}

//—-----------------------------Another class—------------------------------------------------

package com.mycompany.app;

import java.util.Random;

public class CachingPerformanceTest {

public static void main(String[] args) throws Exception {

testPerformance(10\_000, 1\_000); // 10,000 products, 1,000 cache size

testPerformance(1\_000\_000, 100\_000); // 1 million products, 100,000 cache size

}

private static void testPerformance(int numProducts, int cacheSize) throws Exception {

ProductService service = new ProductService(numProducts, cacheSize);

Random random = new Random();

System.out.println("\nTesting with " + numProducts + " products and cache size " + cacheSize);

// Warm up the cache

for (int i = 0; i < 1000; i++) {

service.getProduct("PROD" + random.nextInt(numProducts));

}

// Test performance

long startTime = System.currentTimeMillis();

for (int i = 0; i < 10000; i++) {

service.getProduct("PROD" + random.nextInt(numProducts));

}

long endTime = System.currentTimeMillis();

System.out.println("Time taken for 10,000 random product retrievals: " + (endTime - startTime) + "ms");

service.printCacheStats();

}

}

**======================= Hierarchical Caching ============================**

* Here we create 3 Caches, for the first cache we give less capacity and time limit, for the second cache we give more capacity and time limit than the first cache, and for the third cache we give a lot of capacity and time limit.
* Whenever we want an object we first search in the first cache, if found we return it else we go search in second cache. If the object is present in the second cache we first add that object to first cache and then return that object.
* If the object is not present in either first or second cache we go for third cache, if found we add that object to first and second cache and then return the object. If not found we return null.

**Program:**

package com.mycompany.app;

import com.google.common.cache.Cache;

import com.google.common.cache.CacheBuilder;

import java.io.\*;

import java.nio.file.\*;

import java.util.concurrent.TimeUnit;

public class HierarchicalCache {

private final Cache<String, Product> l1Cache;

private final Cache<String, Product> l2Cache;

private final Path l3CacheDir;

public HierarchicalCache(int l1Size, int l2Size, String l3Path) throws IOException {

this.l1Cache = CacheBuilder.newBuilder()

.maximumSize(l1Size)

.expireAfterWrite(1, TimeUnit.MINUTES)

.recordStats()

.build();

this.l2Cache = CacheBuilder.newBuilder()

.maximumSize(l2Size)

.expireAfterWrite(5, TimeUnit.MINUTES)

.recordStats()

.build();

this.l3CacheDir = Paths.get(l3Path);

Files.createDirectories(l3CacheDir);

}

public Product get(String key) throws IOException, ClassNotFoundException {

// Try L1 Cache

Product product = l1Cache.getIfPresent(key);

if (product != null) {

return product;

}

// Try L2 Cache

product = l2Cache.getIfPresent(key);

if (product != null) {

l1Cache.put(key, product);

return product;

}

// Try L3 Cache

Path filePath = l3CacheDir.resolve(key);

if (Files.exists(filePath)) {

try (ObjectInputStream ois = new ObjectInputStream(Files.newInputStream(filePath))) {

product = (Product) ois.readObject();

l2Cache.put(key, product);

l1Cache.put(key, product);

return product;

}

}

return null;

}

public void put(String key, Product value) throws IOException {

l1Cache.put(key, value);

l2Cache.put(key, value);

// Write to L3 Cache

Path filePath = l3CacheDir.resolve(key);

try (ObjectOutputStream oos = new ObjectOutputStream(Files.newOutputStream(filePath))) {

oos.writeObject(value);

}

}

public void printStats() {

System.out.println("L1 Cache Stats: " + l1Cache.stats());

System.out.println("L2 Cache Stats: " + l2Cache.stats());

}

}

//—---------------------------------------------Another class—----------------------------------------

package com.example;

import java.util.Random;

public class DataGenerator {

private static final Random random = new Random();

public static Product generateProduct(String id) {

return new Product(id, "Product " + id, 10 + random.nextDouble() \* 990);

}

public static String generateRandomId(int maxId) {

return "PROD" + random.nextInt(maxId);

}

}

//—--------------------------------Another class—----------------------------------------------

package com.mycompany.app;

import java.io.IOException;

public class HierarchicalCacheTest {

private static final int TOTAL\_PRODUCTS = 100\_000;

private static final int TEST\_ITERATIONS = 1\_000\_000;

public static void main(String[] args) throws IOException, ClassNotFoundException {

HierarchicalCache cache = new HierarchicalCache(100, 1000, "l3cache");

// Populate cache

System.out.println("Populating cache...");

for (int i = 0; i < TOTAL\_PRODUCTS; i++) {

String id = "PROD" + i;

cache.put(id, DataGenerator.generateProduct(id));

}

// Test random access

System.out.println("Testing random access...");

long startTime = System.currentTimeMillis();

for (int i = 0; i < TEST\_ITERATIONS; i++) {

String randomId = DataGenerator.generateRandomId(TOTAL\_PRODUCTS);

Product product = cache.get(randomId);

if (product == null) {

System.out.println("Product not found: " + randomId);

}

if (i % 100000 == 0) {

System.out.println("Completed " + i + " iterations");

cache.printStats();

}

}

long endTime = System.currentTimeMillis();

System.out.println("Total time for " + TEST\_ITERATIONS + " random accesses: " + (endTime - startTime) + "ms");

cache.printStats();

}

}