COMP2500 실습 4

목차

수업 중에 싱글턴 디자인 패턴을 사용하는 법과 싱글턴이 정적(static) 클래스보다 나은 점에 대해 살펴봤습니다. 이 실습에서는 싱글턴 패턴을 확장한 멀티턴(multiton)을 만들어 보면 어떨까요? 멀티턴을 사용하면 인스턴스를 오직 하나만 만들 수 있었던 싱글턴의 제약을 받지 않아도 됩니다. 그 대신 생성될 인스턴스 수를 제어할 수 있죠. 이럴 때 일반적으로 맵(map)을 사용하여 새로 생성되는 인스턴스를 관리한답니다.

멀티턴을 사용하기 적합한 한 가지 예로 하드 디스크 캐시(cache)가 있습니다. 컴퓨터에 하드 디스크가 여럿 설치되어 있다고 생각해보세요. 데이터는 보통 파일로 저장되어 있으며, 이 데이터를 읽으려면 이 파일을 찾아서, 열고, 읽은 뒤, 메모리 어디엔가에 저장할 것입니다. 뭔가 하는 일이 많아 보이지만 이런 일을 아주 가끔씩만 한다면 큰 문제가 없죠. 그러나 이런 일을 빈번한다면 데이터를 메모리에 저장해 두고 재활용하는 게 더 좋을 겁니다. 이렇게 다른 곳에 있는 데이터를 메모리에 저장해 두고 사용하는 걸 캐시로 관리한다고 하는데요. 다음과 같이 멀티턴 패턴을 사용하면 각 하드 디스크마다 전용 캐시를 만들어 줄 수 있습니다.

```
MemoryCache hardDriveC = MemoryCache.getInstance("My Hard Drive C");
MemoryCache usbDrive = MemoryCache.getInstance("My USB");
```

매우 간단해 보이죠?

그러나 메모리는 무한하지 않기에 캐시에 저장된 데이터가 증가하는 것을 어느 정도 제한해야 합니다. 이때 캐시 퇴거 정책(cache eviction policy)이라는 것을 사용합니다. 캐시 퇴거 정책이란 캐시가 가득 찰 때 해야 하는 일을 정의합니다. 새로운 데이터 하나를 캐시에 추가하려면 이미 캐시에 들어있던 데이터를 하나 제거해야겠죠? 근데 어떤 데이터를 지울까요? 제거할 데이터를 고르는 방법도 다양하답니다.

- 선입선출(FIFO): 캐시에 가장 처음에 추가된 데이터부터 제거합니다.
- 후입선출(LIFO): 캐시에 가장 마지막에 추가된 데이터부터 제거합니다.
- LRU(Least Recently Used): 마지막으로 사용한 지 가장 오랜 시간이 지난 데이터부터 제거합니다.

이 실습에서 여러분이 구현할 것이 바로 이 세 정책을 모두 구현하는 멀티턴 캐시 시스템이랍니다. ^o^

1. 프로젝트를 준비한다

1. `Lab4` 폴더로 이동합니다.
2. `src/academy/pocu/comp2500/lab4` 폴더에 `MemoryCache.java` 파일을 추가합니다.
3. 다음 내용을 위 파일에 추가합니다.

```
package academy.pocu.comp2500.lab4;

public class MemoryCache {

}
```

2. MemoryCache 멀티턴 구현하기

전반적인 규칙

- 실습이 너무 어려워지지 않도록 캐시에 저장하는 모든 값은 `String` 이라고 가정하겠습니다.
- `LinkedHashMap` 클래스를 사용할 수 없습니다.
- 어떤 클래스 대신 사용할 수 있는 기본 자료형이 존재한다면(예: Boolean/boolean, Integer/int) 그걸 대신 사용하세요. 기본 자료형 대신 클래스 버전이 허용되는 경우는 제네릭(generic) 클래스의 타입(type) 매개변수로 사용할 때 뿐입니다.

2.1 getInstance() 메서드를 구현한다

- `getInstance()` 메서드는 다음의 인자를 받습니다.
  - 하드 디스크의 이름을 나타내는 `String`
- 이 메서드는 지정된 하드 디스크 전용 `MemoryCache` 인스턴스를 반환합니다.
- 지정된 하드 디스크 전용 `MemoryCache` 인스턴스가 존재하지 않는다면 새로 생성하여 반환합니다.
- 만약 현재 사용 중인 `MemoryCache` 인스턴스의 수가 최대 허용치를 넘으면 마지막으로 사용한 지 가장 오래된(LRU) 인스턴스를 제거합니다.

```
MemoryCache memCacheA = MemoryCache.getInstance("A");
MemoryCache memCacheB = MemoryCache.getInstance("B");
MemoryCache memCacheC = MemoryCache.getInstance("C");
```

2.2 clear() 메서드를 구현한다

- `clear()` 메서드는 아무 인자도 받지 않습니다.
- 모든 캐시 인스턴스를 제거합니다.

```
MemoryCache memCacheA = MemoryCache.getInstance("A");
MemoryCache memCacheB = MemoryCache.getInstance("B");
MemoryCache memCacheC = MemoryCache.getInstance("C");

MemoryCache.clear();
```

2.3 setMaxInstanceCount() 메서드를 구현한다

- `setMaxInstanceCount()` 메서드는 다음의 인자를 받습니다.
  - 이 멀티턴 클래스에서 허용하는 최대 `MemoryCache` 인스턴스 수를 나타내는 `int`
- 이 멀티턴 클래스에서 허용하는 최대 `MemoryCache` 인스턴스 수를 설정합니다.
- 이 값은 언제나 0보다 크다고 가정하세요.
- 만약 현재 사용 중인 `MemoryCache` 인스턴스의 수가 최대 허용치를 넘게 되면 LRU에 기초하여 초과분을 제거합니다.

```
MemoryCache.setMaxInstanceCount(3);
MemoryCache memCacheA = MemoryCache.getInstance("A");
MemoryCache memCacheB = MemoryCache.getInstance("B");
MemoryCache memCacheC = MemoryCache.getInstance("C");
MemoryCache memCacheD = MemoryCache.getInstance("D"); // memCacheA가 제거됨
```

2.4 setEvictionPolicy() 메서드를 구현한다

- `setEvictionPolicy()` 메서드는 다음의 인자를 받습니다.
  - 캐시 퇴거 정책(eviction policy)을 나타내는 `EvictionPolicy` 열거형
- `MemoryCache` 인스턴스의 캐시 퇴거 정책을 설정합니다.
- `EvictionPolicy` 에 사용할 수 있는 값은 다음과 같습니다.
  - `FIRST_IN_FIRST_OUT`
  - `LAST_IN_FIRST_OUT`
  - `LEAST_RECENTLY_USED`
- 기본(default) 캐시 퇴거 정책은 `LEAST_RECENTLY_USED` 입니다.

```
MemoryCache memCache = MemoryCache.getInstance("A");

memCache.setEvictionPolicy(EvictionPolicy.FIRST_IN_FIRST_OUT);
memCache.setEvictionPolicy(EvictionPolicy.LAST_IN_FIRST_OUT);
memCache.setEvictionPolicy(EvictionPolicy.LEAST_RECENTLY_USED);
```

2.5 addEntry() 메서드를 구현한다

- `addEntry()` 메서드는 다음의 인자를 받습니다.

    - 키(key)를 나타내는 `String`
    - 값(value)을 나타내는 `String`
- 키-값 쌍을 캐시에 추가합니다.
- 캐시에 이미 키가 존재한다면 그 키에 연결된 값을 업데이트합니다.
- 캐시 속에 저장된 항목수가 최대 허용치를 넘으면 현재 사용 중인 캐시 퇴거 정책에 따라 항목 하나를 제거합니다.

```
MemoryCache memCache = MemoryCache.getInstance("A");

memCache.addEntry("key1", "value1");
memCache.addEntry("key2", "value2");
```

2.6 `getEntryOrNull()` 메서드를 구현한다

- `getEntryOrNull()` 메서드는 다음의 인자를 받습니다.
    - 키(key)를 나타내는 `String`
- 키에 연결된 값을 캐시로부터 찾아 반환합니다.

```
MemoryCache memCache = MemoryCache.getInstance("A");

memCache.addEntry("key1", "value1");

String value = memCache.getEntryOrNull("key1"); // value: value1
```

2.7 `setMaxEntryCount()` 메서드를 구현한다

- `setMaxEntryCount()` 메서드는 다음의 인자를 받습니다.
    - 캐시 속에 저장할 수 있는 최대 항목수를 나타내는 `int`
- 캐시에 저장할 수 있는 최대 항목수를 설정합니다.
- 인자로 전달되는 값은 언제나 0보다 크다고 가정하셔도 좋습니다.
- 현재 캐시에 저장되어 있는 항목수가 최대 허용치보다 크다면 현재 사용 중인 캐시 퇴거 정책에 따라 초과분을 제거합니다.

```
MemoryCache memCache = MemoryCache.getInstance("A");
memCache.setMaxEntryCount(3);

memCache.addEntry("key1", "value1");
memCache.addEntry("key2", "value2");
memCache.addEntry("key3", "value3");

memCache.addEntry("key4", "value4"); // 캐시로부터 key1이 제거됨
```

3. 본인 컴퓨터에서 테스트하는 법

- `Program.java`를 아래처럼 바꾼 뒤 실행하세요.

```java
package academy.pocu.comp2500.lab4.app;

import academy.pocu.comp2500.lab4.EvictionPolicy;
import academy.pocu.comp2500.lab4.MemoryCache;

public class Program {

    public static void main(String[] args) {
        {
            MemoryCache memCacheA = MemoryCache.getInstance("A");

            MemoryCache memCacheB = MemoryCache.getInstance("B");
            MemoryCache memCacheC = MemoryCache.getInstance("C");

            assert memCacheA == MemoryCache.getInstance("A");
            assert memCacheB == MemoryCache.getInstance("B");
            assert memCacheC == MemoryCache.getInstance("C");

            MemoryCache.setMaxInstanceCount(3);

            MemoryCache memCacheD = MemoryCache.getInstance("D");

            assert memCacheA != MemoryCache.getInstance("A");
            assert memCacheC == MemoryCache.getInstance("C");
            assert memCacheB != MemoryCache.getInstance("B");
            assert memCacheD != MemoryCache.getInstance("D");
        }

        {
            MemoryCache memCache = MemoryCache.getInstance("A");
            memCache.addEntry("key1", "value1");
            memCache.addEntry("key2", "value2");
            memCache.addEntry("key3", "value3");
            memCache.addEntry("key4", "value4");
            memCache.addEntry("key5", "value5");

            memCache.setMaxEntryCount(3);

            assert memCache.getEntryOrNull("key1") == null;
            assert memCache.getEntryOrNull("key2") == null;
            assert memCache.getEntryOrNull("key3") != null;
            assert memCache.getEntryOrNull("key4") != null;
            assert memCache.getEntryOrNull("key5") != null;

            memCache.addEntry("key6", "value6");

            assert memCache.getEntryOrNull("key3") == null;

            memCache.getEntryOrNull("key4");
            memCache.getEntryOrNull("key5");
            memCache.getEntryOrNull("key4");

            memCache.addEntry("key7", "value7");

            assert memCache.getEntryOrNull("key6") == null;

            memCache.addEntry("key5", "value5_updated");
            memCache.addEntry("key8", "value8");

            assert memCache.getEntryOrNull("key4") == null;

            assert memCache.getEntryOrNull("key5").equals("value5_updated");

            memCache.setEvictionPolicy(EvictionPolicy.FIRST_IN_FIRST_OUT);

            memCache.addEntry("key9", "value9");
            assert memCache.getEntryOrNull("key5") == null;

            memCache.addEntry("key10", "value10");
            assert memCache.getEntryOrNull("key7") == null;

            memCache.setMaxEntryCount(1);

            assert memCache.getEntryOrNull("key8") == null;
            assert memCache.getEntryOrNull("key9") == null;
            assert memCache.getEntryOrNull("key10").equals("value10");

            memCache.setMaxEntryCount(5);
            memCache.setEvictionPolicy(EvictionPolicy.LAST_IN_FIRST_OUT);

            memCache.addEntry("key11", "value11");
```

```
            memCache.addEntry("key12", "value12");
            memCache.addEntry("key13", "value13");
            memCache.addEntry("key14", "value14");

            assert memCache.getEntryOrNull("key10") != null;
            assert memCache.getEntryOrNull("key11") != null;
            assert memCache.getEntryOrNull("key12") != null;
            assert memCache.getEntryOrNull("key13") != null;
            assert memCache.getEntryOrNull("key14") != null;

            memCache.addEntry("key15", "value15");

            assert memCache.getEntryOrNull("key14") == null;

            assert memCache.getEntryOrNull("key13") != null;
            assert memCache.getEntryOrNull("key11") != null;
            assert memCache.getEntryOrNull("key12") != null;
            assert memCache.getEntryOrNull("key10") != null;
            assert memCache.getEntryOrNull("key15") != null;

            memCache.setEvictionPolicy(EvictionPolicy.LEAST_RECENTLY_USED);

            memCache.addEntry("key16", "value16");

            assert memCache.getEntryOrNull("key13") == null;
            assert memCache.getEntryOrNull("key10") != null;
            assert memCache.getEntryOrNull("key11") != null;
            assert memCache.getEntryOrNull("key12") != null;
            assert memCache.getEntryOrNull("key15") != null;
            assert memCache.getEntryOrNull("key16") != null;
        }
    }
}
```

4. 커밋, 푸시 그리고 빌드 요청

이건 어떻게 하는지 이제 다 아시죠? :)

5. 도전 정신이 강하신 분들을 위해...

`getInstance()`, `getEntryOrNull()`, 그리고 `addEntry()` 메서드가 O(1) 시간 복잡도 안에 실행되게 만들어보세요. 그런다고 빌드봇이 보너스 점수를 주진 않지만, 실습이 너무 쉽다고 느끼셨다면 한 단계 더 도약해보는 것도 좋겠죠?