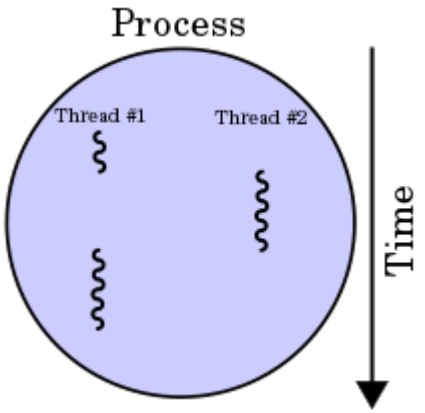
객체지향 프로그래밍 및 실습

13주차. Concurrency

1. About Threads

- Thread
  - **스레드**(thread)는 어떠한 프로그램 내에서, 특히 프로세스 내에서 실행되는 흐름의 단위를 말한다. 일반적으로 한 프로그램은 하나의 스레드를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 스레드를 동시에 실행할 수 있다. 이러한 실행 방식을 멀티스레드(multithread)라고 한다.



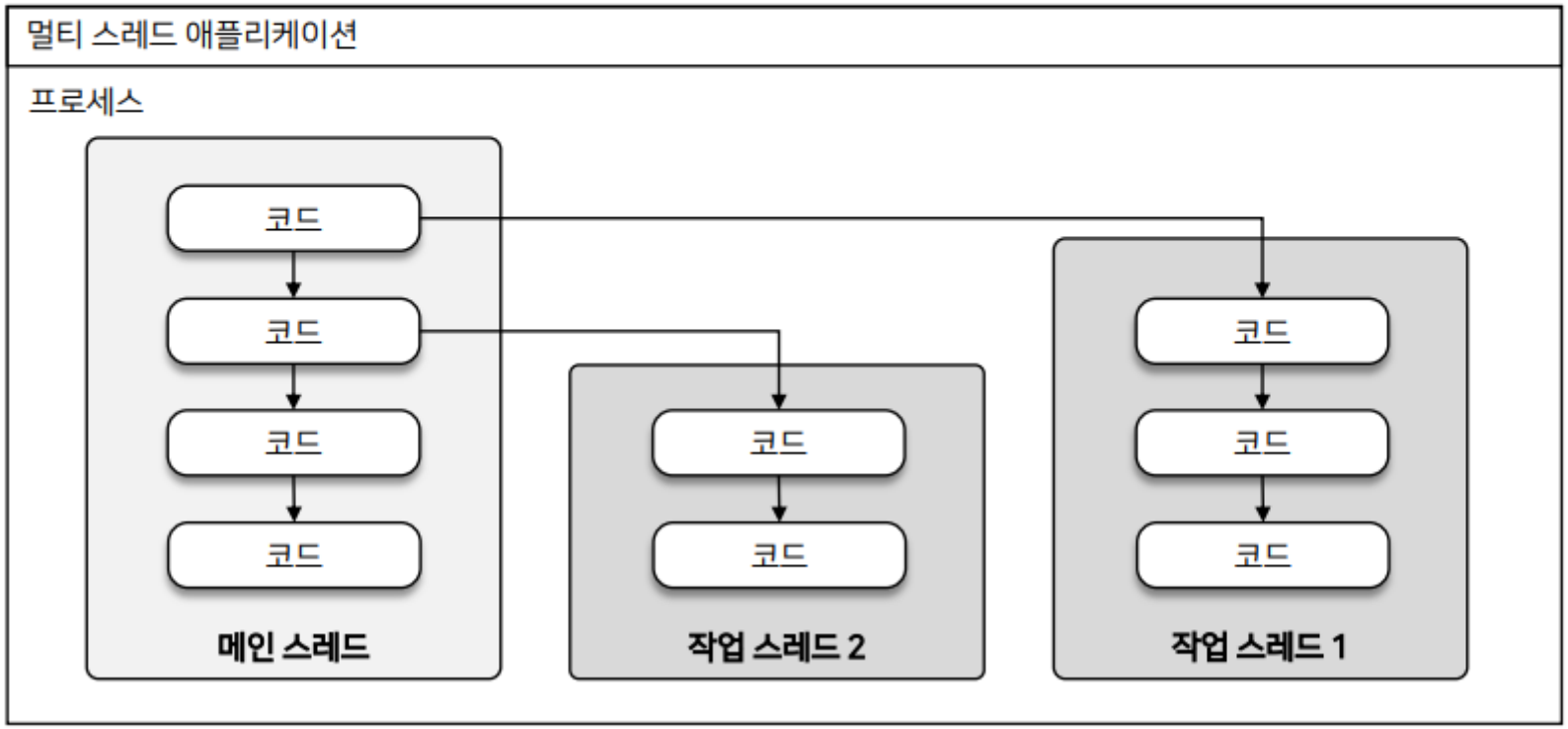
  - 우리는 아직까지 main thread만 다루어 봤다.
  - 따라서 프로그램의 흐름도 하나!

  - 프로그램이 여러 흐름을 가지게 하도록 튜닝해보자.

1. About Threads

2. 메인 스레드와 작업 스레드

- 모든 자바 애플리케이션은 메인 스레드가 main() 메소드를 실행하면서 시작된다.
- 메인 스레드는 필요에 다라 작업 스레드들을 만들어서 병렬로 코드를 실행할 수 있다.
- 즉, 멀티 스레드를 생성해서 멀티 태스킹을 수행한다.
- 아래의 그림은 메인 스레드가 작업 스레드1을 생성하고 실행한 다음, 곧이어 작업 스레드 2를 생성하고 실행한다.



1. About Threads

- 가장 간단한 Thread 생성 방법
  - 1. Runnable을 구현하는 클래스 생성
  - 스레드에서 실행할 작업에 대한 코드를 run() 메서드 내에 작성한다.

  - 오른쪽 코드의 run()은 계속해서 자신의 이름을 출력하는 코드

```java
package Threads;

public class ThreadClass implements Runnable {
    private String name;
    private boolean runState = false;
    public ThreadClass(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        runState = true;
        while (runState) {
            System.out.println(name);
        }
    }

    public void stop() {
        runState = false;
    }

}
```

1. About Threads

- 가장 간단한 Thread 생성 방법
  - 1. Runnable을 구현하는 클래스 생성

```java
package Threads;

public class ThreadLambda {
    public static void main(String[] args) throws InterruptedException {
        ThreadClass tc1 = new ThreadClass("T1");
        ThreadClass tc2 = new ThreadClass("T2");

        Thread th1 = new Thread(tc1);
        Thread th2 = new Thread(tc2);

        th1.start();
        th2.start();

        Thread.sleep(5000);
        tc1.stop();
        tc2.stop();
    }
}
```

1. About Threads

- 가장 간단한 Thread 생성 방법
  - Thread 클래스의 생성자는 Runnable 객체를 받는다.
  - 그렇다면 익명클래스로도 사용할 수 있지 않을까?

  - 2. 익명클래스 활용

```java
package Threads;

public class ThreadAnn {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new Runnable() {
            @Override
            public void run() {
                while (!Thread.interrupted()) {
                    System.out.println("this is t1");
                }
            }
        });
        Thread t2 = new Thread(new Runnable() {
            @Override
            public void run() {
                while (!Thread.interrupted()) {
                    System.out.println("this is t2");
                }
            }
        });

        t1.start();
        t2.start();

        Thread.sleep(5000);
        t1.interrupt();
        t2.interrupt();
    }
}
```

1. About Threads

- 가장 간단한 Thread 생성 방법
  - Thread 클래스의 생성자는 Runnable 객체를 받는다.
  - 그렇다면 익명클래스로도 사용할 수 있지 않을까?
  - 3. 람다식으로의 사용

```java
package Threads;

public class ThreadAnn {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            while (!Thread.interrupted()) {
                System.out.println("this is t1");
            }
        });
        Thread t2 = new Thread(() -> {
            while (!Thread.interrupted()) {
                System.out.println("this is t2");
            }
        });

        t1.start();
        t2.start();

        Thread.sleep(5000);
        t1.interrupt();
        t2.interrupt();
    }
}
```

1. About Threads

- 가장 간단한 Thread 생성 방법
  - Thread를 상속해도 가능
  - 4. Thread의 상속

```java
// ThreadExtend.java
public class ThreadExtend extends Thread {
    @Override
    public void run() {
        // 명령
    }
}

// ThreadExe.java
public class ThreadExe {
    public static void main(String[] args) {
        ThreadExtend  t1 = new ThreadExtend();
        Thread t2 = new ThreadExtend();
        t1.start();
        t2.start();
    }
}
```

1. About Threads

- **실습 문제 1**
  - 숫자를 0부터 100,000 까지 출력하는 스레드를 3개 생성하여 실행시켜보자
  - 출력은 “Thread 1 (숫자)”, “Thread 2 (숫자)” 형식으로 한다.
  - 3개의 스레드 모두 출력이 다 끝났다면, main스레드에 “Print End”를 출력한다.
  - 참고: 해당 스레드의 실행이 끝나기 까지 잠시 실행을 멈추기 위해서는 Thread.join() 메서드를 활용해야 한다.

1. About Threads

- 참고사항 – Thread에서 발생한 Exception
  - 모든 스레드는 name 속성을 가지고 있다.
  - 기본이 되는 스레드는 main스레드이며, Thread 객체를 생성할 때 name을 따로 지정할 수도 있다.
  - 만약 지정이 안되었다면 Thread-0, Thread-1 순으로 지정됨

```
1 System.out.println(Thread.currentThread().getName());
2 new Thread(() -> System.out.println(Thread.currentThread().getName()), "Hi Thread").start();
3 new Thread(() -> System.out.println(Thread.currentThread().getName()), "Hello Thread").start();
```

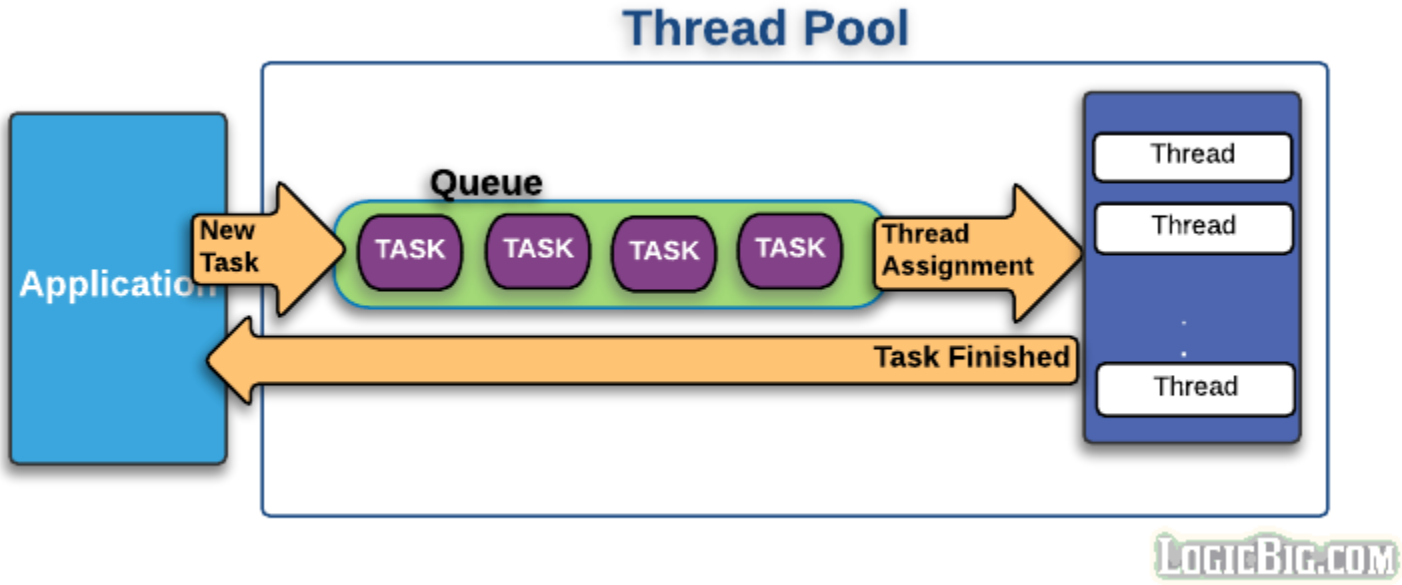
```
1 new Thread(() -> {
2     throw new RuntimeException("This is Exception");
3 }, "Hi Thread").start();
```

```
Exception in thread "Hi Thread" java.lang.RuntimeException: This is Exception
        at Threads.ThreadName.lambda$0(ThreadName.java:7)
        at java.base/java.lang.Thread.run(Thread.java:829)
```

2. Executor Framework

- Thread pool
  - 스레드를 생성하고, 실행이 끝난 뒤 수거하는데 많은 비용이 소모된다.
  - 따라서 계속해서 스레드를 사용해야 하는 환경에서는 new Thread() 키워드를 자주 쓰는 것이 매우 비효율적!

  - 스레드를 미리 만들어 놓고, 필요할 때 꺼내 쓴다면 어떨까



2. Executor Framework

- Executor Service
  - 자바에서는 java.uitl.concurrency 패키지에서 스레드풀 관련 클래스들을 제공하고 있다.
  - ExecutorService는 인터페이스
  - Executors의 new~ 메서드로 생성되는 객체들은 ThreadPoolExecutor 클래스

```
1 public static void main(String[] args) {
2     ExecutorService a = Executors.newFixedThreadPool(4);
3     ExecutorService b = Executors.newCachedThreadPool();
4     ExecutorService c = Executors.newSingleThreadExecutor();
5 }
```

2. Executor Framework

- FixedThreadPool
  - 풀에 생성될 스레드의 개수를 지정할 수 있다.
  - 만약 지정된 개수보다 많이 들어온다면, 빈 자리가 생길 때 까지 대기한다.
  - 마찬가지로 Runnable 타입을 인자로 받는다.

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class FixedPool {
    public static void main(String[] args) {
        PrintTask[] tasks = new PrintTask[10];
        for (int i = 0; i < tasks.length; i++) {
            tasks[i] = new PrintTask(i);
        }

        ExecutorService pool =
Executors.newFixedThreadPool(4);

        for (PrintTask taskObj : tasks) {
            pool.execute(taskObj);
        }
    }
}
```

2. Executor Framework

- FixedThreadPool
  - 풀에 생성될 스레드의 개수를 지정할 수 있다.
  - 만약 지정된 개수보다 많이 들어온다면, 빈 자리가 생길 때 까지 대기한다.
  - 마찬가지로 Runnable 타입을 인자로 받는다.

```
19 class PrintTask implements Runnable {
20     private int num;
21 
22     public PrintTask(int num) {
23         this.num = num;
24     }
25 
26     @Override
27     public void run() {
28         for (int i = 0; i < 20; i++) {
29             System.out.printf("Thread%d %d\n", num, i);
30             try {
31                 Thread.sleep(500);
32             } catch (InterruptedException e) {
33                 e.printStackTrace();
34             }
35         }
36     }
37 }
```

2. Executor Framework

- CachedThreadPool
  - 스레드 개수에 제한이 없고, 60초 이내에 다른 작업 요청이 없을 때 스레드를 삭제하는 유동적 스레드 풀
  - ((ThreadPoolExecutor) pool).getPoolSize()로 풀의 개수가 변하는 것을 확인해보자

```
1 ExecutorService pool = Executors.newCachedThreadPool();
```

2. Executor Framework

- **실습 문제 2**
  - 실습 문제 1을 Executor Framework 사용하도록 변경해보자.

2. Executor Framework

- Future 객체
  - 스레드를 사용하면 프로그램을 여러 흐름으로 분기할 수 있다는 것을 알았다.
  - 그렇다면 다음같은 경우에서는 어떻게 해야 할까?

  - 무작위로 생성된 3000개 숫자의 합을 알고 싶다.
  - 스레드1은 1~1000까지의 수를, 스레드2는 1001~2000까지의 수를, 스레드3은 2001~3000까지의 수를 각각 구해서 합쳐본다.
  - 스레드가 계산한 합을 내가 전달받을 때는 어떻게 해야 할까..

  - 공유하는 객체를 통해 값을 받는 방법, 핸들러를 사용하는 방법 등 다양하지만..
  - 제일 쉬운 방법은 Future 객체를 사용하는 것.

2. Executor Framework

- Future 객체
  - Future 객체를 받기 위해서는 ExecutorService.sumbit()을 호출해야 한다.
  - 반대로 받지 않기 위해서는 ExecutorService.execute()
  - submit()은 Callable(또는 Runnable)을 받고, execute()는 Runnable을 받는다.

```java
package executor;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FutureTest {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService pool = Executors.newFixedThreadPool(5);
        Future<Integer> future = pool.submit(new Counter());
        System.out.println(future.get());
        pool.shutdown();
    }

}

class Counter implements Callable<Integer> {
    @Override
    public Integer call() {
        int sum = 0;
        for (int i = 0; i < 100000; i++) {
            sum += 1;
        }
        return sum;
    }
}
```

2. Executor Framework

- Future 객체
  - 간략한 예시
  - https://github.com/lani009/Ajou-OOP-Practice_22/blob/main/Lecture%20Codes/week13/Week13/src/executor/Prob2.java

3. Synchronization

- 멀티스레딩에 의해서 발생할 수 있는 문제

```
35 /**
36  * 조회수 카운터
37  */
38 class ViewCounter {
39     private int count = 0;
40 
41     /**
42      * 누군가가 영상을 조회했을 경우
43      */
44     public void viewCount() {
45         count = count + 1;
46     }
47 
48     public int getCount() {
49         return count;
50     }
51 }
```

3. Synchronization

- 멀티스레딩에 의해서 발생할 수 있는 문제
  - 여러 서버에서 동시에 조회수가 올라간다.
  - 이 때 카운트 되어야 하는 조회수는 당연히 100000 * 3이 맞는데? 실행해보면 그렇지 않다.
  - count = count + 1연산이 사실은 하나의 연산이 아니기 때문이다.

```java
package sync;

public class SyncExample {
    static ViewCounter vc = new ViewCounter();

    public static void main(String[] args) throws InterruptedException {
        // 유튜브는 쾌적한 스트리밍 환경을 제공하기 위해서
        // 여러 나라에 각각의 서버를 두고 있다.
        // 이 때 여러 서버에서 동시에 조회수가 계속
        // 업데이트 되는 상황을 가정해보자.

        Thread koreaServer = new Thread(SyncExample::forCount);

        Thread japanServer = new Thread(SyncExample::forCount);

        Thread europeServer = new Thread(SyncExample::forCount);

        koreaServer.start();
        japanServer.start();
        europeServer.start();

        koreaServer.join();
        japanServer.join();
        europeServer.join();

        System.out.println(vc.getCount());
    }

    public static void forCount() {
        for (int i = 0; i < 100000; i++) {
            vc.viewCount();
        }
    }
}
```

3. Synchronization

- 멀티스레딩에 의해서 발생할 수 있는 문제

3. Synchronization

- 멀티스레딩에 의해서 발생할 수 있는 문제
  - 공유 자원에 대해서 여러 스레드가 동시에 접근할 가능성이 있을 때는 동기화를 해야 한다.
  - 가장 대표적인 방법은 *synchronized* 키워드
  - Synchronized 키워드가 붙은 메서드는 한번에 하나의 스레드만 접근 가능해진다.

```java
/**
 * 조회수 카운터
 */
class ViewCounter {
    private int count = 0;

    /**
     * 누군가가 영상을 조회했을 경우
     */
    public synchronized void viewCount() {
        count = count + 1;
    }

    public int getCount() {
        return count;
    }
}
```

3. Synchronization

- 멀티스레딩에 의해서 발생할 수 있는 문제
  - synchronized 블록을 사용하는 것도 방법 중의 하나.
  - 메서드 전체에 lock을 걸지 않고, 특정 코드만 lock하는 것도 성능을 높이는 방법

```
1 /**
2  * 조회수 카운터
3  */
4 class ViewCounter {
5     private int count = 0;
6 
7     /**
8      * 누군가가 영상을 조회했을 경우
9      */
10    public void viewCount() {
11        synchronized (this) {
12            count = count + 1;
13        }
14    }
15 
16    public int getCount() {
17        return count;
18    }
19 }
```

3. Synchronization

- **실습 문제 3**
  - 100000개의 실수 난수를 8분할하여 8개의 스레드로 그 합을 구해보자
  - 이 때, SumCounter 클래스에 add메서드를 구현하여 이를 모든 스레드가 호출하도록 한다.
  - SumCounter::add(double a)
  - -0.5 ~ 0.5사이의 실수 난수 배열(또는 리스트) 생성은 아래 코드를 사용
  - Random rand = new Random(System.currentTimeMillis());
  - double[] numList = DoubleStream.generate(() -> rand.nextDouble() - 0.5).limit(100000000).toArray();
  - List<Double> numList = DoubleStream.generate(() -> rand.nextDouble() - 0.5).limit(100000000).boxed().collect(Collectors.toList());

3. Synchronization

- Producer Consumer 문제
  - 간단한 예시
    - Producer는 해야할 일을 만들어내고 (ex. 1+5+3+5)
    - Consumer는 해야할 일을 수행한다 (ex. 1+5+3+5=14)
    - 만약 producer가 일을 생성하는 속도에 비해 consumer가 느리다면 일이 쌓여서 메모리가 넘쳐나게 될 것
    - 만약 consumer가 일을 수행하는 속도에 비해 producer가 느리다면 불필요한 연산을 계속 수행할 것

```
P pool-1-thread-1 put 0,  sum 0
C pool-1-thread-2 got 0,  sum 0
C pool-1-thread-2 got 0,  sum 0
P pool-1-thread-1 put 1,  sum 0
C pool-1-thread-2 got 1,  sum 1
P pool-1-thread-1 put 2,  sum 1
C pool-1-thread-2 got 2,  sum 3
P pool-1-thread-1 put 3,  sum 3
C pool-1-thread-2 got 3,  sum 6
P pool-1-thread-1 put 4,  sum 6
C pool-1-thread-2 got 4,  sum 10
P pool-1-thread-1 put 5,  sum 10
P pool-1-thread-1 put 6,  sum 15
C pool-1-thread-2 got 6,  sum 16
P pool-1-thread-1 put 7,  sum 21
C pool-1-thread-2 got 7,  sum 23
P pool-1-thread-1 put 8,  sum 28
P pool-1-thread-1 put 9,  sum 36
C pool-1-thread-2 got 9,  sum 32
C pool-1-thread-2 got 9,  sum 41
```

3. Synchronization

- Producer Consumer 문제
  - 간단한 예시
    - Producer는 해야할 일을 만들어내고 (ex. 1+5+3+5)
    - Consumer는 해야할 일을 수행한다 (ex. 1+5+3+5=14)
    - 만약 producer가 일을 생성하는 속도에 비해 consumer가 느리다면 일이 쌓여서 메모리가 넘쳐나게 될 것
    - 만약 consumer가 일을 수행하는 속도에 비해 producer가 느리다면 불필요한 연산을 계속 수행할 것
  - 해결방법
    - Producer가 Consumer에 비해 너무 빠르게 생성하면 Producer를 조금 쉬도록 하자
    - Consumer가 Producer에 비해 너무 빠르게 처리하면 Consumer를 조금 쉬도록 하자

3. Synchronization

- Producer Consumer 문제
  - 일을 생성한 순서대로 처리한다. -> Queue 구조
  - 여기에 동기화를 적용할 수 있도록 스레드를 blocking 한다. -> BlockingQueue

BlockingQueue는 결국 Producer가 작업을 빨리 처리하거나, Consumer가 작업을 빨리 처리하는 경우의 오류를 피할수 있도록 돕고, 작업의 대한 스레드 간의 동기화 및 작업량을 조절해 Thread-Safe 하게 프로그램을 작성 할수 있습니다.

put()는 큐의 크기에 제한이 되어 있을 경우, 큐에 빈 공간이 생길 때 까지 대기 하고, 빈 공간이 생기면 작업을 큐의 넣어 줍니다. take()는 큐의 작업이 없을 경우, 작업이 생길때 까지 대기 하고, 작업이 생기면, 큐에서 작업을 가져 갑니다.

BlockingQueue의 종류에는 자바 클래스 라이브러리에서 LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue가 존재 합니다.

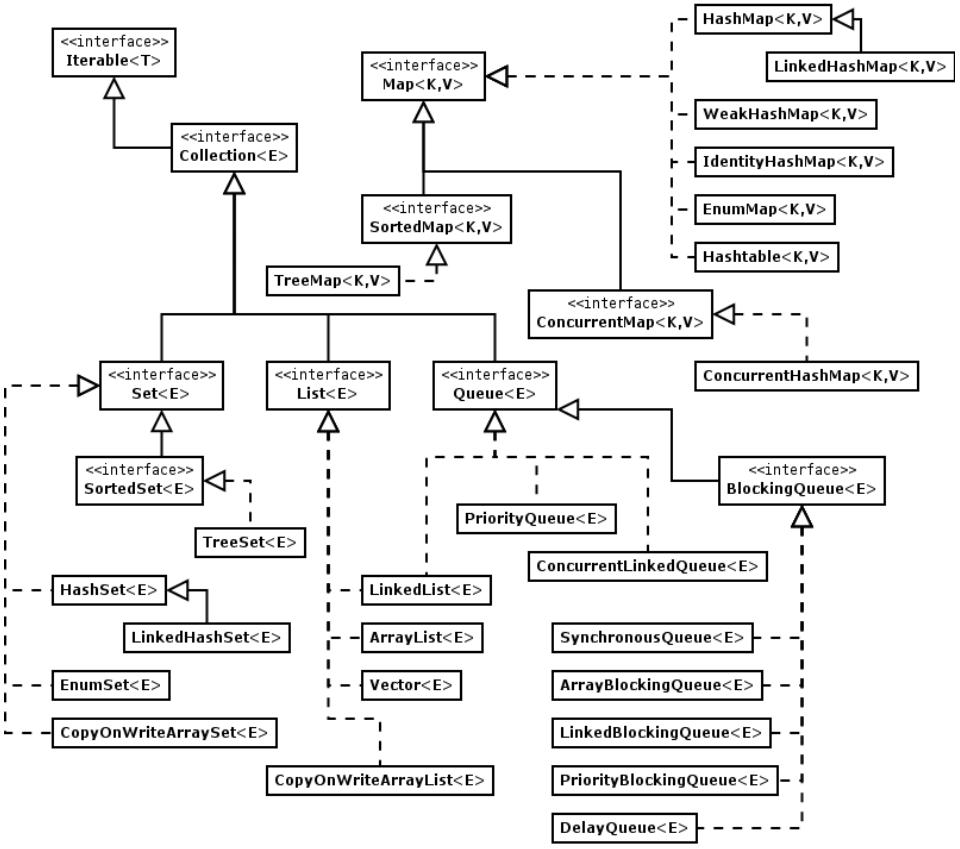
3. Synchronization

- **실습 문제 4**
  - Consumer Producer 예제로 보여준 코드를 Synchronized 하게 수정해보자.
  - BlockingQueue를 사용하여 구현한다.

4. Concurrent Collections

- Synchronized vs Concurrent Collections
  - Synchronized Collections는 읽기 또는 쓰기 작업 시 인스턴스 자체에 lock이 걸린다.
  - 따라서 멀티스레드 환경에서 thread-safe 하지만, 성능은 저하된다.

  - Concurrent Collections에서는 필요한 부분만 최대한 작은 범위를 lock하여 thread-safe 함은 물론 동시성을 보장한다.



4. Concurrent Collections

- putIfAbsent로 알아보는 예시 – 실습문제5

```
44 class PutNum implements Runnable {
45     private Map<Integer, String> map;
46     private Set<Integer> putSet;
47 
48     public PutNum(Map<Integer, String> map, Set<Integer> putSet) {
49         this.map = map;
50         this.putSet = putSet;
51     }
52 
53     @Override
54     public void run() {
55         for (int i = 0; i < 10000; i++) {
56             if (map.putIfAbsent(i, String.format("%c", 'A' + i)) == null) {
57                 putSet.add(i);
58             }
59         }
60     }
61 }
```

4. Concurrent Collectio

- putIfAbsent로 알아보는 예시

```java
package concurrent;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ConTest {
    public static void main(String[] args) throws InterruptedException {
        Map<Integer, String> map = new HashMap<>();
        List<Set<Integer>> putSetList = new ArrayList<>();

        ExecutorService pool = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
            Set<Integer> putSet = new HashSet<>();
            putSetList.add(putSet);
            pool.execute(new PutNum(map, putSet));
        }
        pool.shutdown();
        pool.awaitTermination(100000, TimeUnit.MINUTES);

        for (Set<Integer> set1 : putSetList) {
            for (Set<Integer> set2 : putSetList) {
                if (set1 == set2) {
                    continue;
                }
                Set<Integer> duplicateSet = new HashSet<>();
                duplicateSet.addAll(set1);
                duplicateSet.retainAll(set2);
                for (int dupNum : duplicateSet) {
                    System.out.println(dupNum);
                }
            }
        }
    }
}
```

4. Concurrent Collections

- putIfAbsent로 알아보는 예시
  - Map.putIfAbsent() 메서드는 해당하는 key가 없을 때, value를 넣고 null을 반환한다.
  - if (~.putIfAbsent() == null) -> 값이 map에 들어 갔음을 의미

  - 따라서 putSet에 중복되는 값이 있다는 것은 동시에 put이 되었다는 것을 의미한다
  - 하지만, 이는 옳지 않은 연산

  - HashMap을 ConcurrentHashMap으로 바꾸고 달라진 결과값을 확인해보자