JAVA 입문 : 이론과 실습

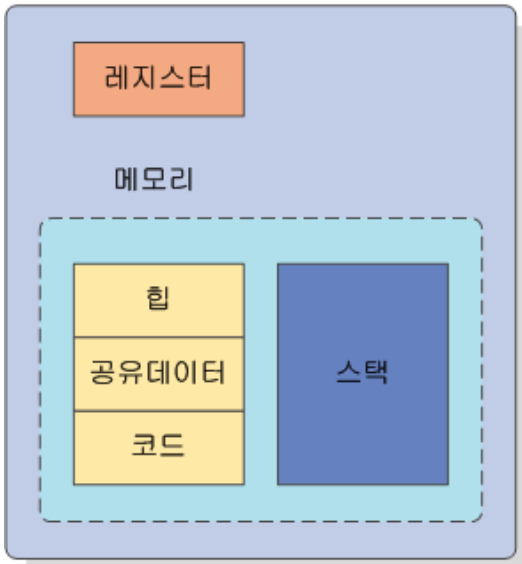
제 11장 스레드

목차

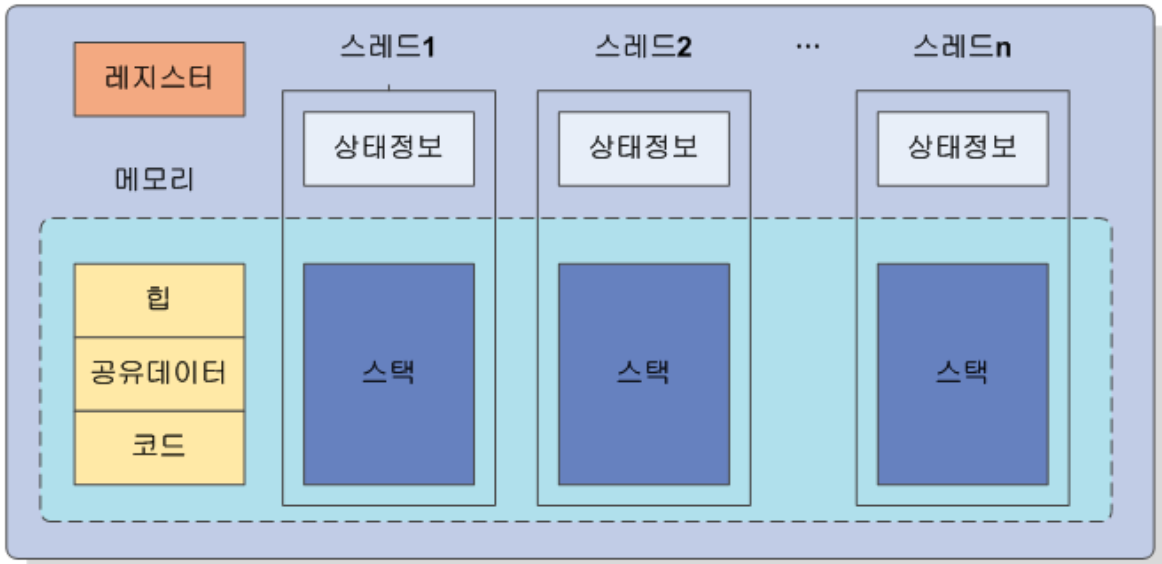
스레드란? [1/2]

- 시작, 실행, 종료의 순서를 가지는 제어 흐름 단위
  - 단일스레딩 시스템: 오직 하나의 실행점
  - 멀티스레딩 시스템: 다중의 실행 점

<단일스레딩 시스템>



<멀티스레딩 시스템>



스레드란? [2/2]

- Concurrent Programming
  - Multiprogramming System
  - Multiprocessing System
  - Multithreading System

- 프로세스에 비해 **문맥 전환**(context switching)에 대한 오버헤드를 줄일 수 있다.
  - 공유 힙, 공유 데이터, 코드를 공유
  - 동일 주소 공간

스레드의 생성[1/6]

- 스레드도 하나의 객체로 처리.
  - 스레드 객체의 생성

```
Thread worker = new Thread();
```

  - 스레드 시작

```
worker.start();
```

  - 스레드 생성 순서
    - **스레드 클래스 정의 => 스레드 객체 생성 => 스레드 시작**

- 스레드 클래스의 정의
  - 방법 2가지 : Thread 클래스 확장, Runnable 인터페이스 구현
  - 스레드 동작 기술 : public void run() 메소드를 재정의

스레드의 생성[2/6]

- Thread 클래스를 확장하여 구현
  - 확장된 클래스를 위한 run 메소드를 작성

```
class SimpleThread extends Thread {
        public void run() {
                // …
        }
}
```

  - 스레드 객체 생성

```
SimpleThread t = new SimpleThread() ;
```

  - 스레드 시작

```
t.start() ;
```

스레드의 생성[3/6]

- 객체의 생성과 동시에 시작

```
new SimpleThread().start();
```

- run 메소드
  - 스레드 실행의 시작 장소
  - 순차 프로그램의 main()과 동일
  - run 메소드가 종료되면 스레드 종료

스레드의 생성[4/6]

- Thread 클래스 확장을 통한 스레드 클래스 구현 예

[예제 11.1 - SimpleThreadTest.java]

```
class SimpleThread extends Thread  {
    public SimpleThread(String name) {
        super(name);
    }
    public void run() {
        System.out.println(getName() + " is now running.");
    }
}

public class SimpleThreadTest {
    public static void main(String[] args) {
        SimpleThread t = new SimpleThread("SimpleThread");
        System.out.println("Here : 1");
        t.start();
        System.out.println("Here : 2");
    }
}
```

실행 결과 :

```
Here : 1
SimpleThread is now running.
Here : 2
```

스레드의 생성[5/6]

- **Runnable 인터페이스를 구현하여 스레드 클래스를 정의**

```
public interface Runnable {
        public abstract void run();
   }
```

- Runnable 인터페이스 구현 예 :

```
class SimpleThread  implements  Runnable {
        public void run() {
              // ...
        }
   }
   // ...
   Thread t1 = new Thread(new SimpleThread());
```

스레드의 생성[6/6]

- Runnable 인터페이스 구현을 통한 스레드 클래스 구현 예

[예제 11.2 - ImplOfRunnable.java]

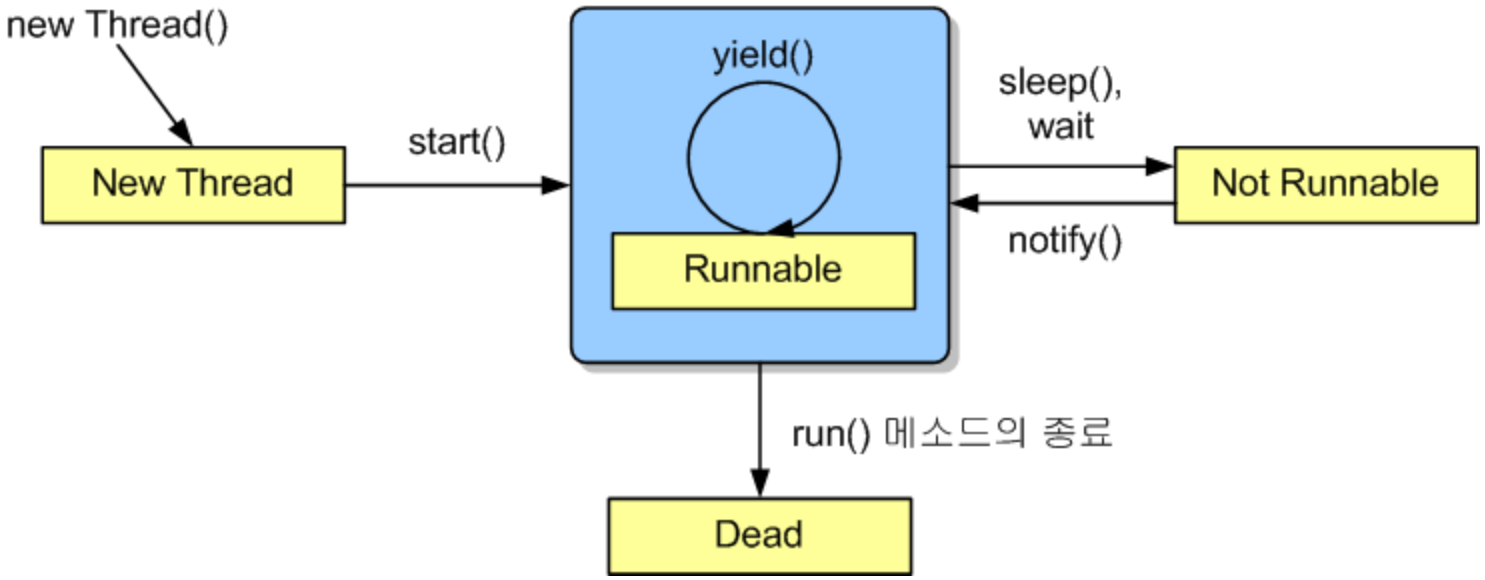
```
class RunnableThread implements Runnable {
    public void run() {
        System.out.println("Implementation of Runnable");
    }
}
public class ImplOfRunnable {
    public static void main(String[] args) {
        Thread t = new Thread (new RunnableThread());
        System.out.println("Here : 1");
        t.start();
        System.out.println("Here : 2");
    }
}
```

실행 결과 :

```
Here : 1
Here : 2
Implementation of Runnable
```

스레드의 상태

- 스레드 상태 전이도와 관련 메소드



스레드의 상태 – 시작

- **New Thread 상태**
  - 스레드 객체만을 생성한 상태
    - Thread myThread = new SimpleThread();
  - 어떤 시스템 자원도 할당 받지 않은 상태
  - start 메소드만 가능

- Runnable 상태
  - 실행 가능한 상태, 스케줄링을 받을 수 있는 상태.
    - myThread.start();
  - start() 메소드
    1. 필요한 시스템 자원을 할당하고
    2. 스레드를 스케줄링한 후,
    3. 스레드 객체의 run() 메소드를 호출한다.

스레드의 상태 – 상태 전이

- Runnable 상태 → Not Runnable 상태
  - sleep()를 호출
  - wait()를 호출
  - 입출력을 위해 블록되었을 때
- Not Runnable 상태 → Runnable 상태
  - sleep() : 지정된 시간 경과
  - wait() : 다른 메소드에 의해 notify(), notifyAll()
  - 입출력 : 해당 입출력의 완료

  ✦ suspend(), resume() 지원 안함

스레드의 상태 – 종료

- **Dead 상태**
  - run 메소드의 종료

교과서 396쪽

```
public void run() {
    int i = 0;
    while ( i < 100) {
        i++;
        System.out.println("i = " + i );
    }
}
```

- isAlive()
  - true: Runnable이거나 Not Runnable 상태
  - false: New Thread이거나 Dead 상태

스레드 스케줄링 [1/7]

- Runnable 상태에 있는 여러 스레드의 실행 순서를 제어
- 고정 우선순위(Fixed Priority) 스케줄링 알고리즘
  - 상대적 우선 순위에 기반을 두고 스레드의 실행 순서를 결정
  - 높은 우선 순위를 갖는 스레드에게 실행의 권한
- 스레드 우선순위
  - MIN_PRIORITY, MAX_PRIORITY
  - 값이 클수록 높은 우선 순위
  - 디폴트 우선 순위 : NORM_PRIORITY

스레드 스케줄링 [2/7]

- 선택된 스레드는 다음 조건 중에 하나가 참이 될 때까지 실행
  - 더 높은 우선 순위의 스레드가 실행 가능하게 될 경우
  - 선택된 스레드가 양보하거나, 그의 run 메소드가 종료된 경우
  - 시분할(time sharing) 시스템에서, 스레드에 할당된 시간이 만료된 경우

- 선점적(preemptive)
  - 다른 Runnable 스레드보다 더 높은 우선 순위를 갖는 스레드가 Runnable하게 되면, 자바 시스템은 새로운 더 높은 우선 순위의 스레드가 실행되도록 선택

스레드 스케줄링 [3/7]

- 스레드를 생성시킨 스레드의 우선 순위 상속
- 스레드 우선 순위 관련 메소드
  - public final void **setPriority**(int newPriority) :
    - 스레드의 우선 순위를 변경한다.
  - public final int **getPriority**() :
    - 현재 스레드의 우선 순위를 반환한다.
- 지속적인 작업을 행하는 스레드의 우선순위는 MIN_PRIORITY로 설정이 바람직
  - 긴급한 작업을 즉각적으로 처리

스레드 스케줄링 [4/7]

- 스케줄링 제어 메소드
  - public static void **sleep**(long millis)
  - public static void **sleep**(long millis, int nanos)
  - public static void **yield**()

- 예제

```
public  void  run() {
        for (int i=0; i < howOften; i++) {
            System.out.println(word);
            if (doYield) yield();
        }
   }
```

스레드 스케줄링 [5/7]

[예제 11.4 - SchedulerTest.java]

```
class RunThread extends Thread {
    public RunThread(String name) {
        super(name);
    }
    public void run() {
        for(int i = 1; i <= 200000; i++) {
            if (i % 50000 == 0)
                System.out.println("Thread [" + getName() + "] is activated => " + i);
        }
    }
}
public class SchedulerTest {
    private final static int NUM = 2;
    public static void main(String[] args) {
        Thread[] p = new RunThread[NUM];

        p[0] = new RunThread("Pear ");
        p[1] = new RunThread("Apple");

        p[0].start();
        p[1].start();
    }
}
```

스레드 스케줄링 [6/7]

- 예제 11.4 결과 1
  - 시분할을 지원하는 시스템의 실행결과

```
Thread [Pear ] is activated => 50000
Thread [Apple] is activated => 50000
Thread [Pear ] is activated => 100000
Thread [Pear ] is activated => 150000
Thread [Apple] is activated => 100000
Thread [Apple] is activated => 150000
Thread [Pear ] is activated => 200000
Thread [Apple] is activated => 200000
```

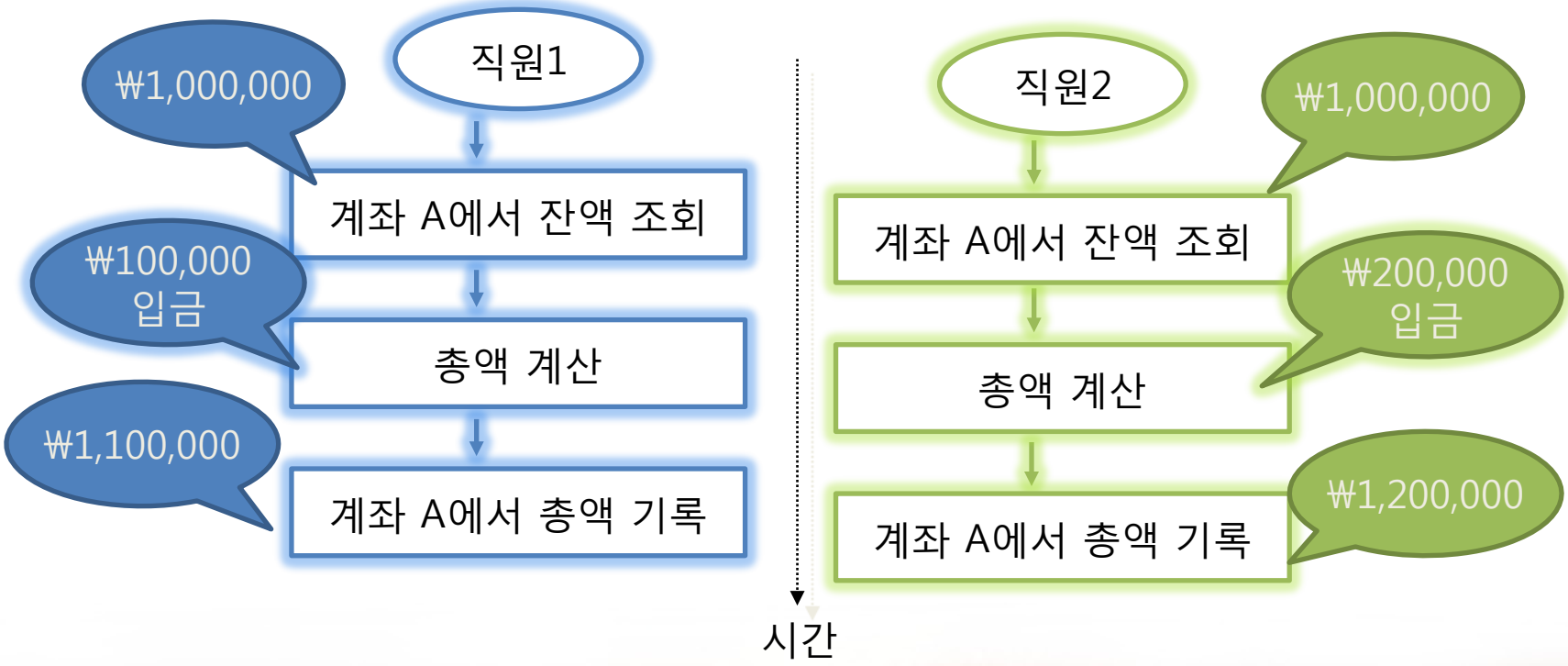
스레드 스케줄링 [7/7]

- 예제 11.4 결과 2
  - 시분할을 지원하지 않는 시스템 실행결과

```
Thread [Pear ] is activated => 50000
Thread [Pear ] is activated => 100000
Thread [Pear ] is activated => 150000
Thread [Pear ] is activated => 200000
Thread [Apple] is activated => 50000
Thread [Apple] is activated => 100000
Thread [Apple] is activated => 150000
Thread [Apple] is activated => 200000
```

동기화(synchronization)

- 여러 스레드의 중첩 실행을 방지

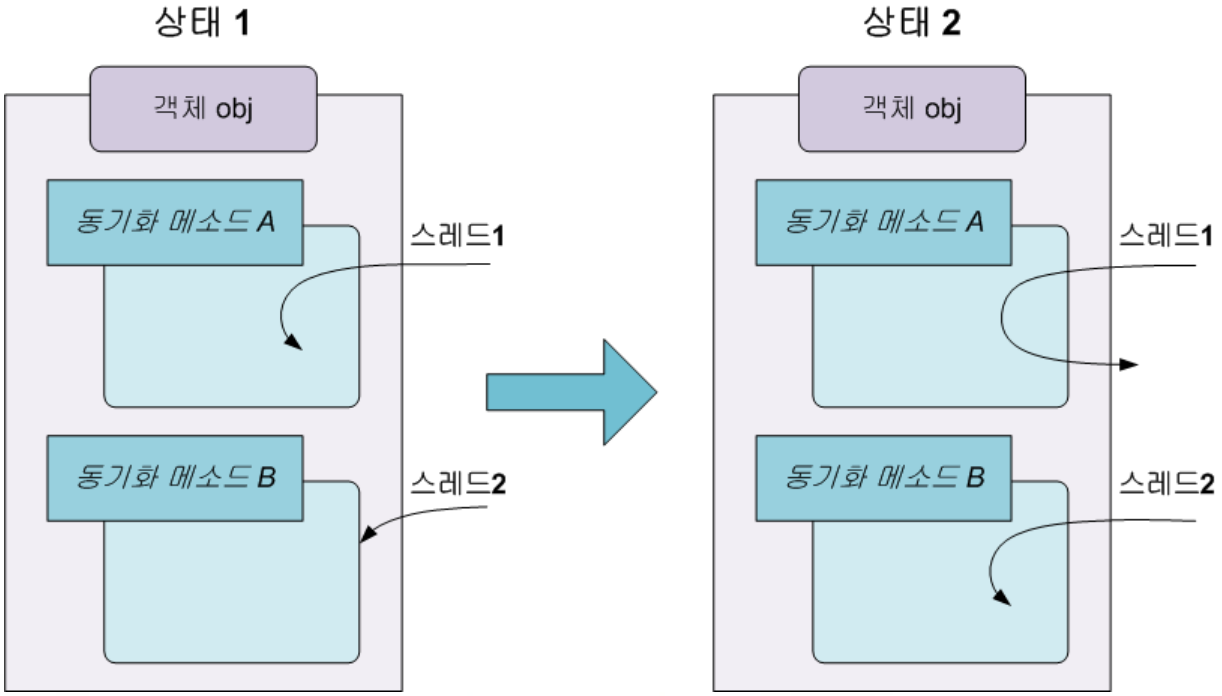


동기화 방법

- 메소드 단위로 --- 동기화 메소드
  - 객체를 Lock
  - synchronized method
- 클래스 단위로 --- 동기화 정적 메소드
  - synchronized static method
- 문장 단위로 --- 동기화 문장
  - synchronized statement

동기화 메소드[1/2]

- 동일한 객체에 대하여 여러 스레드의 중첩 실행을 방지할 때 사용
- 객체의 락(lock)



동기화 메소드[2/2]

- 연산의 순서는 보장되지 않음

```
class Account {
        private double balance;

        public Account(double initialDeposit) {
                balance = initialDeposit;
        }
        public synchronized double getBalance()  {          // 잔액 조회
                return balance;
        }
        public synchronized void deposit(double amount) {   // 총액 계산 및 기록
                balance += amount;
        }
}
```

동기화 정적 메소드

- 클래스 단위로 동기화
- 동시에 같은 클래스에 속한 동기화 정적 메소드를 실행 할 수 없음
- 클래스 단위의 락(lock)은 객체에 영향 없음

- 동기화 메소드 재정의
  - 동기화 속성은 상속되지 않음

동기화 문장

- 문장 단위로 동기화
  - 동기화 메소드는 메소드 전체에 대한 락(lock)을 갖기 때문에 병렬성이 저하
- 문장 형태

```
synchronized  (expr)
            <문장>
```

Lock하고자 하는 객체

```
/* 배열 내의 모든 원소를 양수로 만든다. */
public static void abs(int[] values) {
    synchronized (values) {
        for(int i = 0; i < values.length; i++) {
            if (value[i] < 0)
                values[i] = -values[i];
        }
    }
}
```

멀티스레드 환경 처리

- 멀티스레드 환경을 고려하지 않은 메소드의 처리

```
class ExistingClass {
        // ...
        void method() {
            // ...
        }
}
class ExtendedClass extends ExistingClass {
        // ...
        synchronized void method() {
            super.method();
        }
}
```

- 마치 슈퍼클래스에 있는 메소드가 동기화 메소드인 것처럼 동작

wait와 notify

- 스레드간의 통신이 필요할 때 사용
- wait()
  - 어떤 조건의 변화가 있을 대까지 기다리는 메소드
- notify()
  - 어떤 조건이 변경되었다는 사실을 대기 중인 스레드에게 통지하는 메소드

wait 메소드

- wait 메소드가 사용 가능한 위치
  - 동기화 메소드 내
  - 동기화 문장 내
  - 동기화 정적 메소드 내

```
synchronized void doWhenCondition() {
        while (!condition) wait();
        // Do what needs doing when the condition is true
}
```

- Not Runnable 상태로 전이되고 notify(또는, notifyAll) 메시지를 받을 때까지 기다린다.

notify 메소드

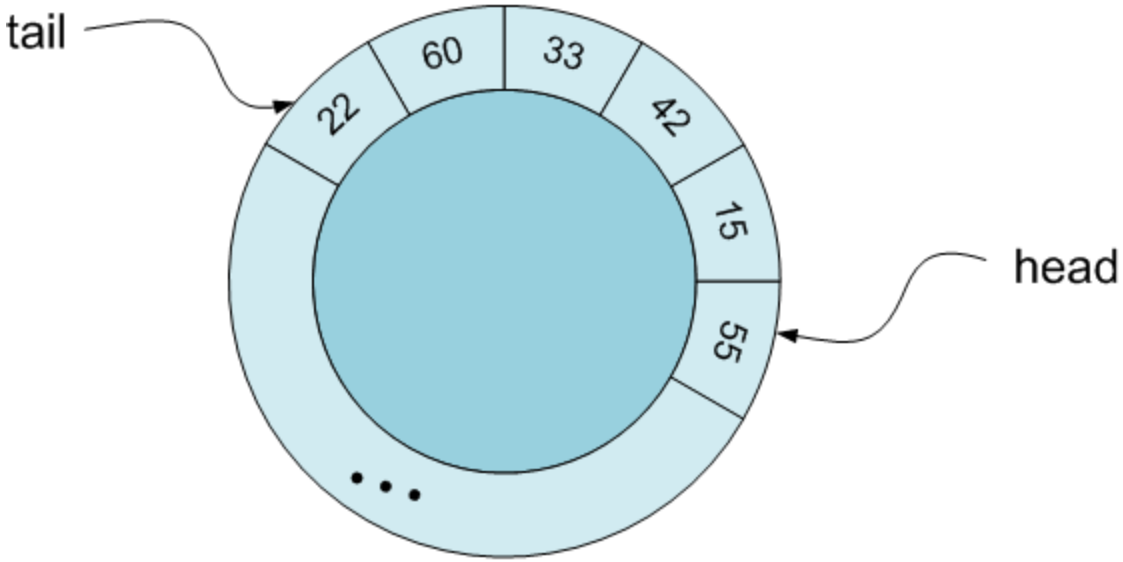
- 대기중인 스레드의 실행을 재개

```
synchronized void changeCondition() {
        // change some value used in a condition test
        notify();
}
```

- notifyAll()
  - 여러 스레드가 같은 객체의 모니터에서 기다리는 경우
  - 모든 대기 스레드를 깨움

Circular Queue 구현 [1/3]

- 환형 큐 - head에 삽입, tail에서 제거



Circular Queue 구현 [2/3]

- get 메소드

```
public synchronized int get() {
        int value;
        while (count == 0)
            try {
                wait();
            } catch (InterruptedException e)  {}
        value = contents[tail];
        tail = (tail + 1) % size;
        count--;
        notifyAll();
        return value;
}
```

Circular Queue 구현 [3/3]

- put 메소드

```
public synchronized void put(int value) {
        while ( count == size )
            try  {
                wait();
            } catch (InterruptedException e)
        contents[head] = value;
        head = (head + 1) % size;
        count++;
        notifyAll();
    }
```

생산자/소비자 문제 [1/5]

- 병행 프로그램의 대표적인 예
- 생산자(producer)
  - 자료의 스트림을 생성하여 버퍼에 삽입
- 소비자(consumer)
  - 버퍼로부터 자료를 제거하여 소비

생산자/소비자 문제 [2/5]

- 고려 사항
  - 소비자는 버퍼가 비어 있으면 블록
  - 생산자는 버퍼가 가득 차 있으면 블록
  - 생산자와 소비자는 버퍼의 용량이 허락하는 한 독립적으로 실행
  - 생산자와 소비자는 동일 시점에 버퍼의 내용을 갱신해서는 안됨
  - 다수의 생산자와 소비자가 존재하는 경우를 대비

생산자/소비자 문제 [3/5]

- Producer 클래스

```
class Producer extends Thread {
        private CircularQueue boundedBuffer;
        private int number;
        public Producer(CircularQueue c, int number) {
            super("Producer #" + number);
            boundedBuffer = c;
            this.number = number;
        }
        public void run() {
            for (int i = 0; i < 10; i++) {
                boundedBuffer.put(i);
                System.out.println(getName() + " put: " + i);
                try {
                    sleep((int)(Math.random() * 100));
                } catch (InterruptedException e)  {}
            }
        }
    }
```

생산자/소비자 문제 [4/5]

- Consumer 클래스

```
class Consumer extends Thread {
        private CircularQueue boundedBuffer;
        private int number;
        public Consumer(CircularQueue c, int number) {
                super("Consumer #" + number);
                boundedBuffer = c;
                this.number = number;
        }
        public void run() {
                int value = 0;
                for (int i = 0; i < 10; i++) {
                        value = boundedBuffer.get();
                        System.out.println(getName() + " got: " + value);
                }
         }
    }
```

생산자/소비자 문제 [5/5]

- 테스트 클래스

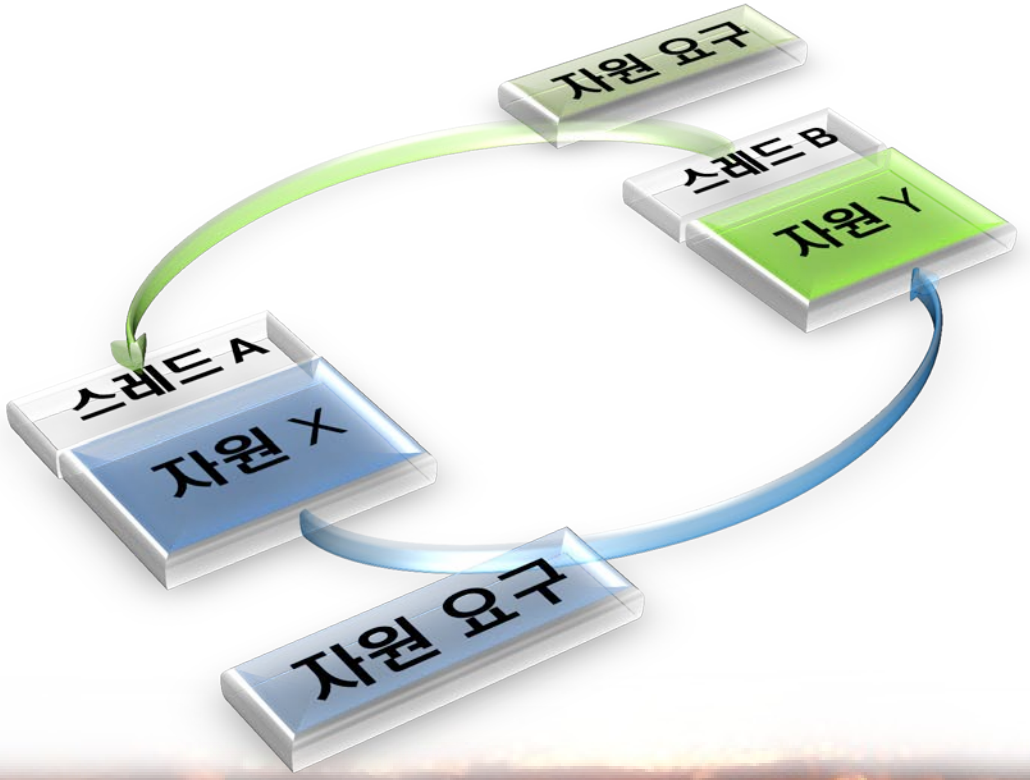
```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        CircularQueue c = new CircularQueue(1);    // 버퍼의 크기는 1로 한정
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
```

결과

```
Producer   #1 put : 0
Consumer #1 got : 0
Producer   #1 put : 1
Consumer #1 got : 1
. . .
```

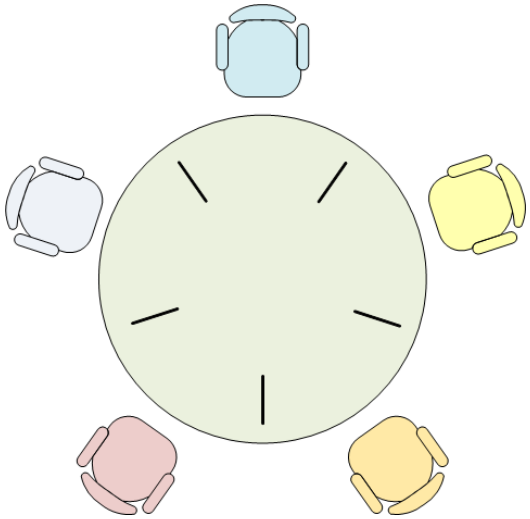
데드락(Deadlock) [1/2]

- 자원을 가지고 있는 스레드와 자원을 요구하는 스레드들 상호간에 **환형 의존**으로 인하여 더 이상의 처리가 불가능한 상태



데드락(Deadlock) [2/2]

- 식사하는 철학자 문제



- 데드락의 해결
  - 데드락 **예방**과 **탐지**(prevention and detection)

스레드 그룹 [1/3]

- 스레드 그룹
  - 모든 스레드는 스레드 그룹의 일원
  - 여러 개의 스레드를 하나의 객체로 관리하는 방법 제공
  - 한번에 여러 개의 스레드를 다룰 수 있는 방법 제공
- 스레드 그룹의 생성

```
ThreadGroup  myThreadGroup =
                        new ThreadGroup(“ThreadGroupName”);

ThreadGroup myThreadGroup =
                        new ThreadGroup(ThreadGroupParent, "ThreadGroupName");
```

- 그룹 안의 스레드 객체 생성

```
Thread myThread = new Thread(myThreadGroup, "a thread for my group");
```

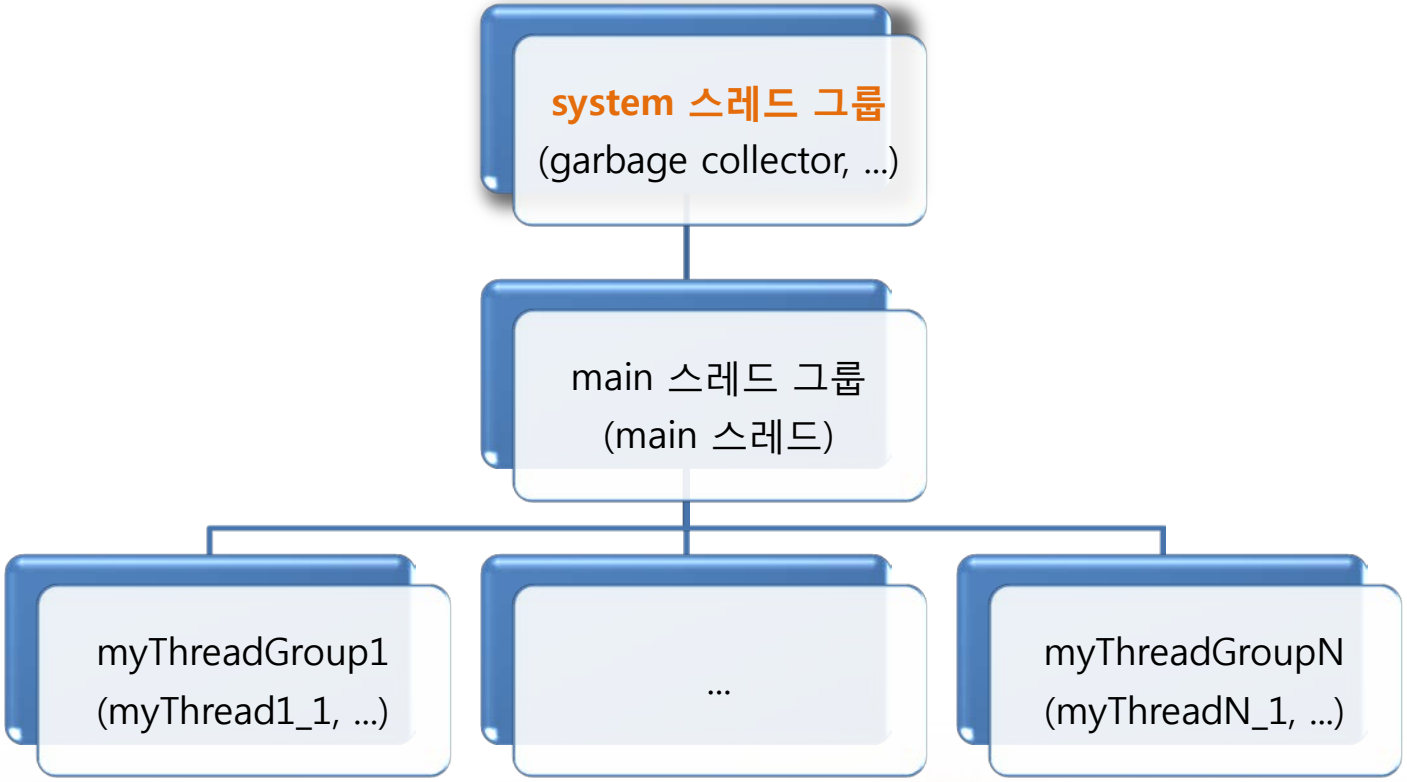
스레드 그룹 [2/3]

- 특정 스레드 그룹에 속하는 스레드 생성
  - public Thread(ThreadGroup group, Runnable target)
  - public Thread(ThreadGroup group, String name)
  - public Thread(ThreadGroup group, Runnable target, String name)
- 자신이 속한 스레드 그룹 확인

```
myGroup = myThread.getThreadGroup();
```

스레드 그룹 [3/3]

- 스레드 그룹 계층 구조



단원 요약 [1/3]

- 스레드
  - 시작, 실행, 종료의 순서를 가지는 제어 흐름 단위
    - 단일스레딩 시스템 : 오직 하나의 실행점을 가짐
    - 멀티스레딩 시스템 : 다중의 실행 점을 가짐
  - 장점
    - 동일 주소 공간을 사용하고 공유 힙, 공유 데이터, 코드를 공유하기 때문에 프로세스에 비해 문맥 전환(context switching)에 대한 오버헤드를 줄일 수 있음
- 스레드 클래스의 정의
  - Thread 클래스 확장
  - Runnable 인터페이스 구현
- 스레드 동작은 public void run() 메소드를 재정의하여 기술함

단원 요약 [2/3]

- 스케줄링
  - Runnable 상태에 있는 여러 스레드의 실행 순서를 제어하는 것을 의미
  - 고정 우선순위(Fixed Priority) 스케줄링 알고리즘
    - 상대적 우선 순위에 기반을 두고 스레드의 실행 순서를 결정
    - 높은 우선 순위를 갖는 스레드에게 실행의 권한
  - 선점적(preemptive)
    - 다른 Runnable 스레드보다 더 높은 우선 순위를 갖는 스레드가 Runnable하게 되면, 자바 시스템은 새로운 더 높은 우선 순위의 스레드가 실행되도록 선택

단원 요약 [3/3]

- 동기화
  - 여러 스레드의 중첩 실행을 방지
    - 동기화 메소드
      - 메소드 단위로 객체를 Lock
    - 동기화 정적 메소드
      - 클래스 단위로 객체를 Lock
    - 동기화 문장
      - 문장 단위로 객체를 Lock
- 스레드 그룹
  - 모든 스레드는 스레드 그룹의 일원
  - 여러 개의 스레드를 하나의 객체로 관리하는 방법 제공
  - 한번에 여러 개의 스레드를 다룰 수 있는 방법 제공