

*Lecture 10*

# 스레드 프로그래밍


2018년도 2학기

## 컴퓨터프로그래밍2

김 영 국

충남대학교 컴퓨터공학과

# 이번 주에 학습할 내용

- 
- 스레드의 개요
  - 스레드의 생성과 실행
  - 스레드 상태
  - 스레드의 스케줄링
  - 스레드 간의 조정

스레드는  
동시에 여러  
개의  
프로그램을  
실행하는  
효과를 냅니다.



# 스레드란?

- **멀티 태스킹(multi-tasking)**은 여러 개의 애플리케이션을 동시에 실행하여서 컴퓨터 시스템의 성능을 높이기 위한 기법

음악을 들으면서  
운동을 할 수 있다.

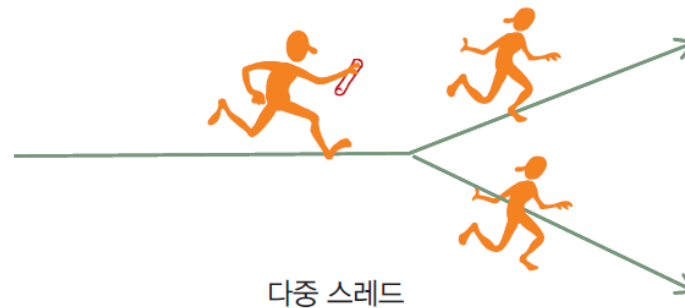


인쇄를 하면서  
문서 편집을 할 수 있다.

< 병렬 처리의 예 >

# 스레드란?

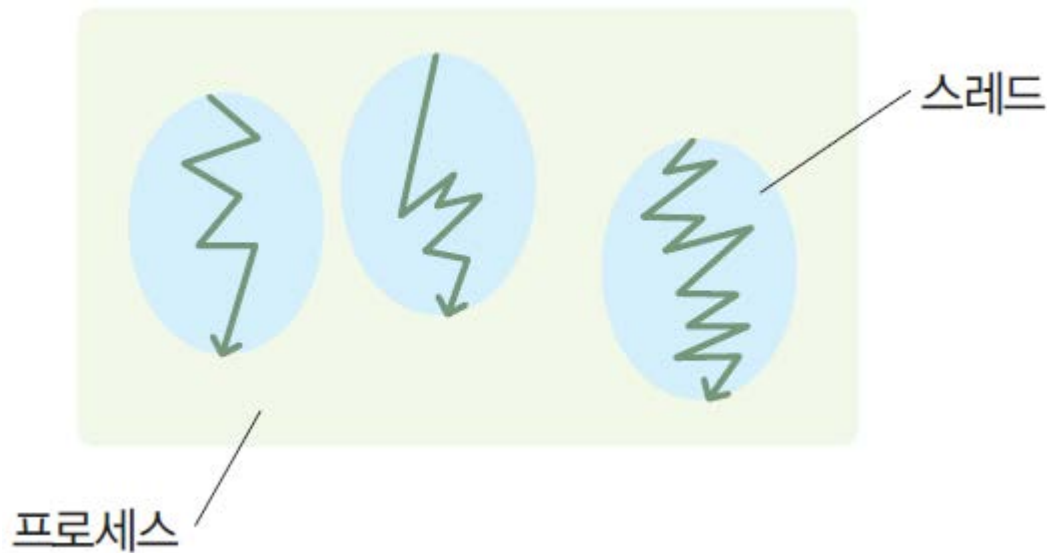
- 다중 스레딩(multi-threading)은 하나의 프로그램이 동시에 여러 가지 작업을 할 수 있도록 하는 것
- 각각의 작업은 **스레드(thread)**라고 불린다.



< 다중 스레드의 개념 >

# 프로세스와 스레드

- 프로세스(process): 자신만의 데이터를 가진다.
- 스레드(thread): 동일한 데이터를 공유한다.



< 스레드는 하나의 프로세스 안에 존재한다. >



# 스레드를 사용하는 이유

---

- 웹 브라우저에서 웹 페이지를 보면서 동시에 파일을 다운로드할 수 있도록 한다.
- 워드 프로세서에서 문서를 편집하면서 동시에 인쇄한다.
- 게임 프로그램에서는 응답성을 높이기 위하여 많은 스레드를 사용한다.
- GUI에서는 마우스와 키보드 입력을 다른 스레드를 생성하여 처리한다.

# 중간 점검 문제



## 중간점검

1. 스레드와 프로세스의 결정적인 차이점은 무엇인가?
2. 스레드를 사용해야만 하는 프로그램을 생각하여 보자.
3. 멀티 스레딩에서 발생할 수 있는 문제에는 어떤 것들이 있을까? 추측하여 보라.



# 스레드 생성과 실행

---

- 스레드는 Thread 클래스가 담당한다.

```
Thread t = new Thread();    // 스레드 객체를 생성한다.  
t.start();                  // 스레드를 시작한다.
```

- 스레드의 작업은 Thread 클래스의 run() 메소드 안에 기술한다.





# 스레드 생성과 실행

## 스레드 생성 방법

Thread 클래스를  
상속하는 방법

Thread 클래스를 상속받은 후에 run() 메소드를 재정의한다.

Runnable 인터페이스를  
구현하는 방법

run() 메소드를 가지고 있는 클래스를 작성하고, 이 클래스의 객체를 Thread 클래스의 생성자를 호출할 때 전달한다.

# Thread 클래스를 상속하기

- Thread를 상속받아서 클래스를 작성한다.
- run() 메소드를 재정의한다.

```
class MyThread extends Thread {  
    public void run() {  
        ...  
    }  
}
```

←----- 여기에 수행하여야 하는 작업을 적어준다.

- Thread 객체를 생성한다.

```
Thread t = new MyThread();
```

- start()를 호출하여서 스레드를 시작한다.

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

# Thread 클래스를 상속하기

MyThreadTest.java

```
01 class MyThread extends Thread {
02     public void run() {
03         for (int i = 10; i >= 0; i--)
04             System.out.print(i + " ");
05     }
06 }
07
08 public class MyThreadTest {
09     public static void main(String args[]) {
10         Thread t = new MyThread();
11         t.start();
12     }
13 }
```

MyThread 클래스는 Thread를 상속받는다. Thread 클래스는 java.lang 패키지에 들어 있어서 따로 import할 필요가 없다. MyThread 클래스는 하나의 메소드 run()만을 가지고 있는데 run()은 이 스레드가 시작되면 자바 런타임 시스템에 의하여 호출된다. 스레드가 실행하는 모든 작업은 이 run() 메소드 안에 있어야 한다. 현재는 단순히 10부터 0까지를 화면에 출력한다.

스레드를 실행시키려면 Thread에서 파생된 클래스 MyThread의 인스턴스를 생성한 후 start()를 호출한다. Thread 타입의 변수 t가 선언되고 MyThread의 객체가 생성하였다. 객체가 생성되었다고 스레드가 바로 시작되는 것은 아니다. start() 메소드를 호출해야만 스레드가 실행된다.

실행결과

10 9 8 7 6 5 4 3 2 1 0



# Runnable 인터페이스를 구현하는 방법

- Runnable 인터페이스를 구현한 클래스를 작성한다.
- run() 메소드를 재정의한다.

```
class MyRunnable implements Runnable {  
    public void run() {  
        ...  
    }  
}
```

- Thread 객체를 생성하고 이때 MyRunnable 객체를 인수로 전달한다.

```
Thread t = new Thread(new MyRunnable());
```

- start()를 호출하여서 스레드를 시작한다.

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

# Runnable 인터페이스를 구현하는 방법

MyRunnableTest.java

```
01 class MyRunnable implements Runnable {
02     public void run() {
03         for (int i = 10; i >= 0; i--)
04             System.out.print(i + " ");
05     }
06 }
07
08 public class MyRunnableTest {
09     public static void main(String args[]) {
10         Thread t = new Thread(new MyRunnable());
11         t.start();
12     }
13 }
```

Runnable을 구현하는 클래스를 작성한다.  
run() 메소드를 재정의하여 작업에 필요한  
코드를 넣는다.

Thread 클래스의 인스턴스를 생성하고,  
Runnable 객체를 Thread 생성자의 매개  
변수로 넘긴다. Thread 객체의 start()  
메소드를 호출하여야 한다.

실행결과

10 9 8 7 6 5 4 3 2 1 0

# 예제



## Q & A

**Q:** 그렇다면 스레드를 생성하기 위해서는 어떤 방법을 사용하는 것이 좋은가?

**A:** `Runnable` 인터페이스를 사용하는 편이 더 일반적이다. `Runnable` 객체는 `Thread`가 아닌 다른 클래스를 상속받을 수 있다. `Thread` 클래스에서 상속받으면 다른 클래스를 상속받을 수 없다. 따라서 인터페이스 방법은 유연할 뿐 아니라 고수준의 스레드 관리 API도 사용할 수 있는 장점이 있다.

# 예제 #1

TestThread.java

```
01 class MyRunnable implements Runnable {
02     String myName;
03     public MyRunnable(String name) { myName = name; }
04     public void run() {
05         for (int i = 10; i >= 0; i--)
06             System.out.print(myName + i + " ");
07     }
08 }
09 public class TestThread {
10     public static void main(String[] args) {
11         Thread t1 = new Thread(new MyRunnable("A"));
12         Thread t2 = new Thread(new MyRunnable("B"));
13         t1.start();
14         t2.start();
15     }
16 }
```

스레드를 구분하기 위하여 이름을 설정한다.

이름이 "A"와 "B"인 스레드 2개를 생성하고 시작한다.

# 예제 #1

## 실행결과

A10 B10 A9 B9 B8 A8 B7 B6 A7 B5 A6 B4 A5 B3 A4 A3 A2 B2 A1 B1 A0 B0

2개의 스레드가 실행되면서 스레드의  
출력이 섞이는 것을 알 수 있다.



# 예제 #2

CountDownTest.java

```
01 ...
02 public class CountDownTest extends JFrame {
03     private JLabel label;
04
05     class MyThread extends Thread {
06         public void run() {
07             for (int i = 10; i >= 0; i--) {
08                 try {
09                     Thread.sleep(1000);
10                 } catch (InterruptedException e) {
11                     e.printStackTrace();
12                 }
13                 label.setText(i + "");
14             }
15         }
16     }
17
18     public CountDownTest() {
19         setTitle("카운트다운");
20         setSize(300, 200);
21         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22     }
23 }
```

스레드를 내부 클래스로 만들면  
필드에 접근하기가 쉬워진다.

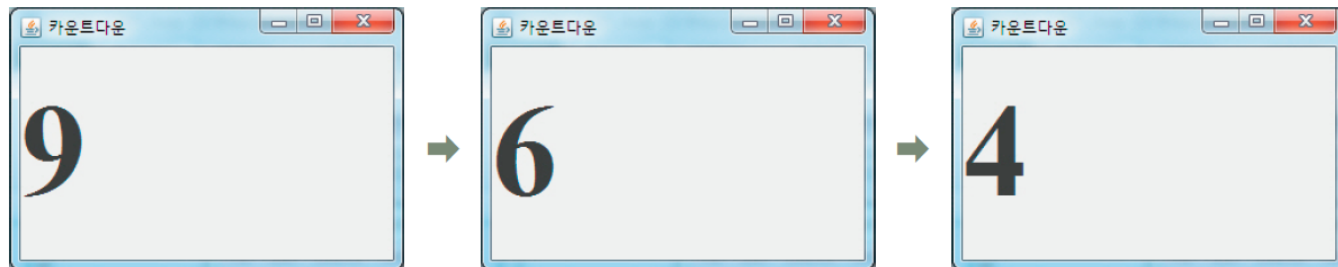
InterruptedException 도중에 예외가  
발생할 가능성이 있다.

1초가 지나가면 레이블  
의 텍스트를 변경한다.

## 예제 #2

```
23     label = new JLabel("Start");
24     label.setFont(new Font("Serif", Font.BOLD, 100));
25     add(label);
26     (new MyThread()).start(); ←----- 스레드를 시작한다.
27     setVisible(true);
28 }
29
30 public static void main(String[] args) {
31     CountdownTest t = new CountdownTest();
32 }
33 }
```

### 실행결과



# 예제#3

CarGame.java

```
01  ...
02  public class CarGame extends JFrame {
03      private JLabel label1;
04      private JLabel label2;
05      private JLabel label3;
06      int x1 = 100, x2 = 100, x3 = 100;
07
08      class MyThread extends Thread {
09          public void run() {
10              for (int i = 0; i < 120; i++) {
11                  try {
12                      Thread.sleep(100);
13                  } catch (InterruptedException e) {
14                      e.printStackTrace();
15                  }
16                  x1 += (int) (Math.random() * 10);
17                  label1.setBounds(x1, 0, 100, 100);
18                  x2 += (int) (Math.random() * 10);
19                  label2.setBounds(x2, 50, 100, 100);
20                  x3 += (int) (Math.random() * 10);
21                  label3.setBounds(x3, 100, 100, 100);
22              }
```

←----- 레이블 하나가 자동차 한 대를 나타낸다.

←----- 자동차의 x좌표

0.1초에 한 번씩 자동차의 위치를  
변경한다. 난수를 발생하여서 그 값  
만큼 자동차의 x좌표를 변경한다.

# 예제#3

```
23     }
24 }
25
26 public CarGame() {
27     setTitle("CarRace");
28     setSize(600, 200);
29
30     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31     setLayout(null);
32     label1 = new JLabel();
33     label1.setIcon(new ImageIcon("car1.gif"));
34     label2 = new JLabel();
35     label2.setIcon(new ImageIcon("car2.gif"));
36     label3 = new JLabel();
37     label3.setIcon(new ImageIcon("car3.gif"));
38     add(label1);
39     add(label2);
40     add(label3);
41     label1.setBounds(100, 0, 100, 100);
42     label2.setBounds(100, 50, 100, 100);
43     label3.setBounds(100, 100, 100, 100);
44     (new MyThread()).start();
45     setVisible(true);
46 }
```

절대 위치를 사용하기 위하여  
배치 관리자를 삭제한다.

이미지 레이블을 만든다.

오른쪽에 정렬시킨다.

## 예제#3

```
47  
48     public static void main(String[] args) {  
49         CarGame t = new CarGame();  
50     }  
51 }
```

### 실행결과



# 중간 점검 문제



## 중간점검

1. 스레딩을 담당하는 클래스의 이름은?
2. Thread를 상속받는 방법의 문제점은 무엇인가?



# Thread 클래스

메소드	설명
<code>Thread()</code>	매개 변수가 없는 기본 생성자
<code>Thread(String name)</code>	이름이 <code>name</code> 인 <code>Thread</code> 객체를 생성한다.
<code>Thread(Runnable target, String name)</code>	<code>Runnable</code> 을 구현하는 객체로부터 스레드를 생성한다.
<code>static int activeCount()</code>	현재 활동 중인 스레드의 개수를 반환한다.
<code>String getName()</code>	스레드의 이름을 반환
<code>int getPriority()</code>	스레드의 우선순위를 반환
<code>void interrupt()</code>	현재의 스레드를 중단한다.
<code>boolean isInterrupted()</code>	현재의 스레드가 중단될 수 있는지를 검사
<code>void setPriority(int priority)</code>	스레드의 우선순위를 지정한다.
<code>void setName(String name)</code>	스레드의 이름을 지정한다.
<code>static void sleep(int milliseconds)</code>	현재의 스레드를 지정된 시간만큼 재운다.
<code>void run()</code>	스레드가 시작될 때 이 메소드가 호출된다. 스레드가 하여야하는 작업을 이 메소드 안에 위치시킨다.
<code>void start()</code>	스레드를 시작한다.
<code>static void yield()</code>	현재 스레드를 다른 스레드에 양보하게 만든다.

# 예제: sleep()

SleepTest.java

```
01 public class SleepTest {  
02     public static void main(String args[]) throws InterruptedException {  
03         String messages[] = { "Pride will have a fall.",  
04             "Power is dangerous unless you have humility.",  
05             "Office changes manners.",  
06             "Empty vessels make the most sound." };  
07  
08         for (int i = 0; i < messages.length; i++) {  
09             Thread.sleep(1000);  
10             System.out.println(messages[i]);  
11         }  
12     }  
13 }
```

sleep()가 다른 메소드에 의하여 중단되면 발생하는 예외, 여기서 처리하지 않고 상위 메소드로 전달한다. 사실 여기서는 다른 메소드가 sleep()을 방해할 일이 없다.

1000밀리 초 동안 실행을 중지한다.

## 실행결과

```
Pride will have a fall.  
Power is dangerous unless you have humility.  
Office changes manners.  
Empty vessels make the most sound.
```



# 인터럽트

- 인터럽트(interrupt)는 하나의 스레드가 실행하고 있는 작업을 중지하도록 하는 메커니즘이다.

```
for (int i = 0; i < messages.length; i++) {  
    try {  
        Thread.sleep(1000);  
    } catch (InterruptedException e) {  
        // 인터럽트를 받은 것이다. 단순히 리턴한다.  
        return;   
    }  
    System.out.println(messages[i]);  
}
```

←----- 인터럽트 처리는 여기에서 해준다.

- 그런데 만약 스레드가 실행 중에 한번도 sleep()을 호출하지 않는다면 InterruptedException를 받지 못한다.

```
if (Thread.interrupted()) {  
    // 인터럽트를 받은 것이다. 단순히 리턴한다.  
    return;  
}
```



# 조인

---

- join() 메소드는 하나의 스레드가 다른 스레드의 종료를 기다리게 하는 메소드이다.

```
t.join();
```

# 예제

ThreadControl.java

```
01 public class ThreadControl {
02
03     static void print(String message) {
04         String threadName = Thread.currentThread().getName();
05         System.out.format("%s: %s\n", threadName, message);
06     }
07
08     private static class MessageLoop implements Runnable {
09         public void run() {
10             String messages[] = { "Pride will have a fall.",
11                                   "Power is dangerous unless you have humility.",
12                                   "Office changes manners.",
13                                   "Empty vessels make the most sound." };
14
15             try {
16                 for (int i = 0; i < messages.length; i++) {
```

메시지를 스레드 이름과  
함께 출력한다.

# 예제

```
17         print(messages[i]);
18         Thread.sleep(2000);
19     }
20     } catch (InterruptedException e) {
21         print("아직 끝나지 않았어요!");
22     }
23 }
24 }
25
26 public static void main(String args[]) throws InterruptedException {
27     int tries = 0;
28
29     print("추가적인 스레드를 시작합니다.");
30     Thread t = new Thread(new MessageLoop());
31     t.start();
32
33     print("추가적인 스레드가 끝나기를 기다립니다.");
34     while (t.isAlive()) {
35         print("아직 기다립니다.");
36         t.join(1000);
37         tries++;
```

← 인터럽트되면 메시지를 출력한다.

← 스레드 t가 종료하기를 1초 동안 기다린다.

# 실행 결과

```
38         if (tries > 2) {
39             print("참을 수 없네요!");
40             t.interrupt();
41             t.join();
42         }
43     }
44     print("메인 스레드 종료!");
45 }
46 }
```

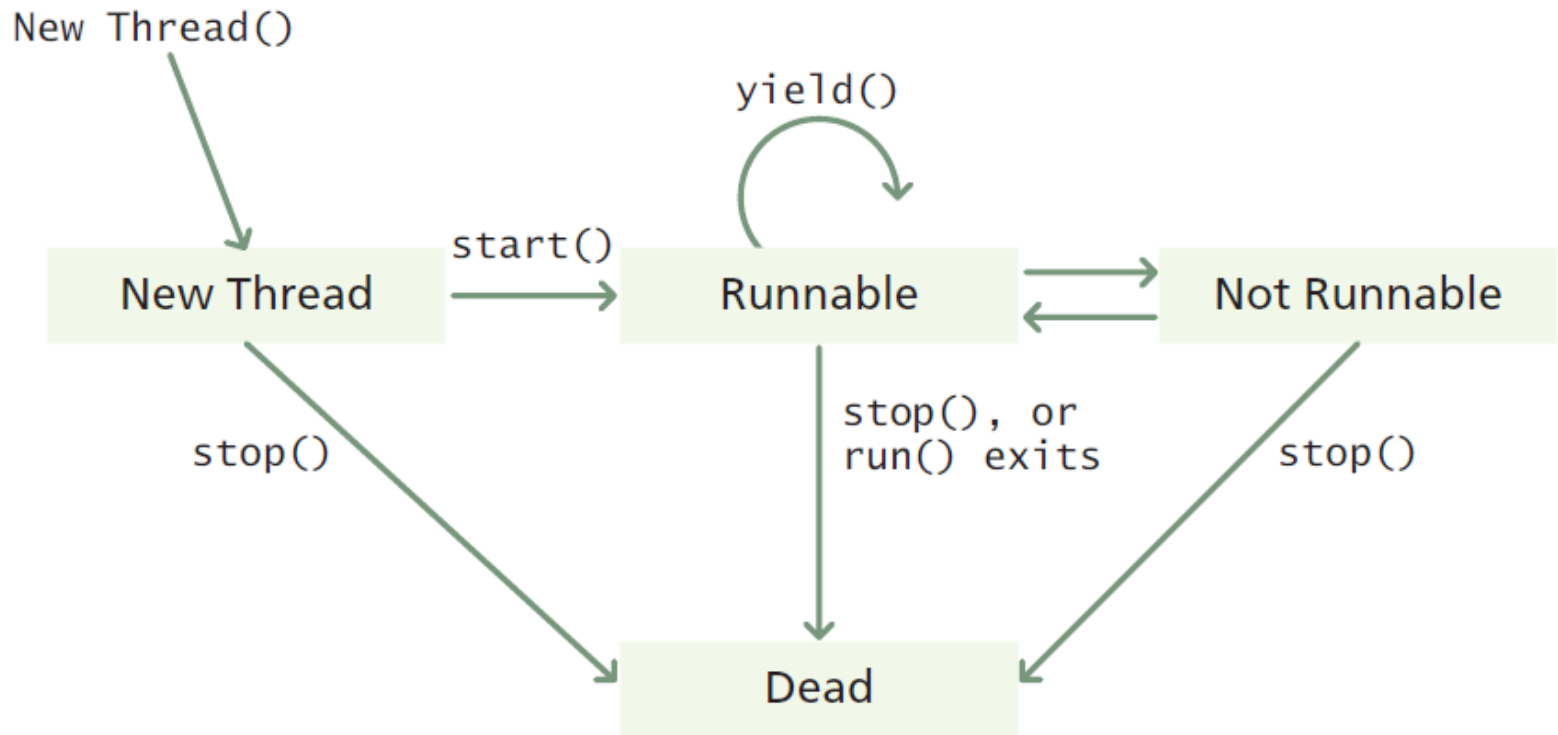
스레드 t를 강제로 중단시킨다.

스레드 t가 종료하기를 기다린다.

## 실행결과

main: 추가적인 스레드를 시작합니다.  
main: 추가적인 스레드가 끝나기를 기다립니다.  
main: 아직 기다립니다.  
Thread-0: Pride will have a fall.  
main: 아직 기다립니다.  
main: 아직 기다립니다.  
Thread-0: Power is dangerous unless you have humility.  
main: 참을 수 없네요!  
Thread-0: 아직 끝나지 않았어요!  
main: 메인 스레드 종료!

# 스레드의 상태



< 스레드의 상태 >

# 중간 점검 문제



## 중간점검

1. `setPriority()`와 `getPriority()`의 역할은?
2. `sleep()` 메소드는 어떤 경우에 사용되는가?
3. 어떤 스레드가 가장 우선적으로 실행되는가?
4. Thread의 `run()` 메소드의 역할은?
5. Thread의 `start()`, `stop()` 메소드의 역할은?
6. 어떤 일이 발생하면 스레드가 실행 중지 상태로 가는가?

# 동기화

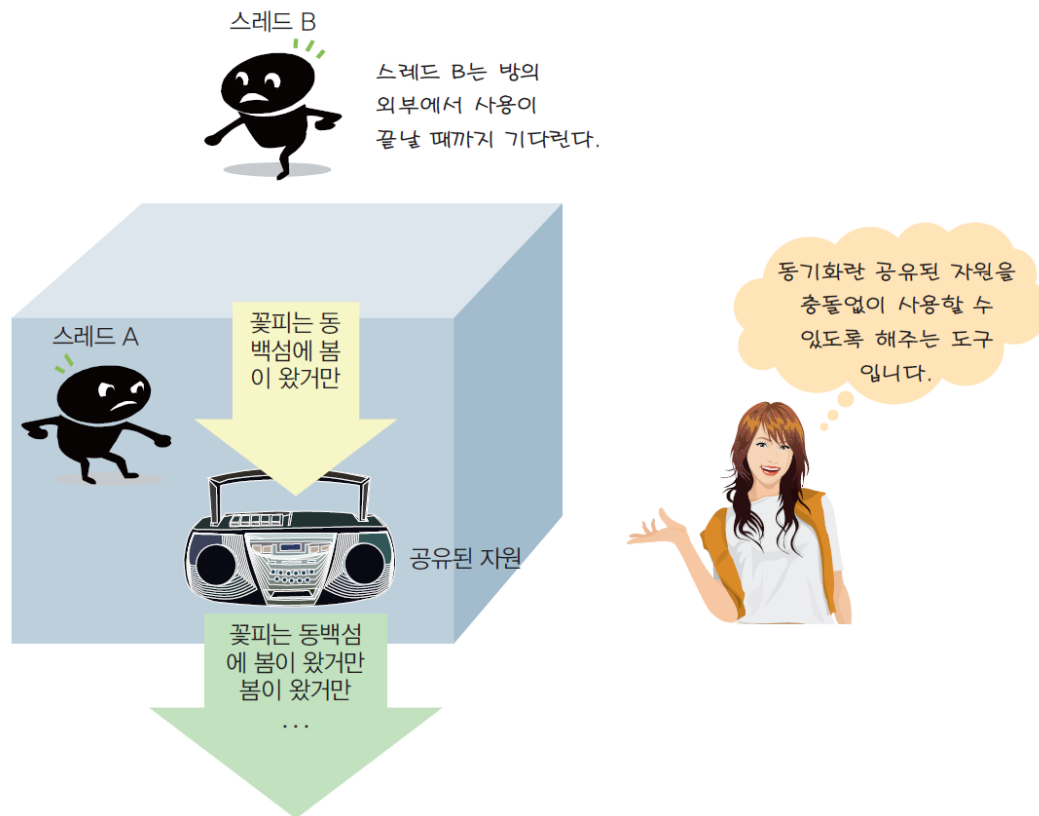
- 동기화(synchronization): 한 번에 하나의 스레드 만이 공유 데이터를 접근할 수 있도록 제어하는 것이 필요





# 동기화 기법

- 동기화란 쉽게 설명하면 공유된 자원 중에서 동시에 사용하면 안 되는 자원을 보호하는 도구이다.



# 스레드 간섭

- 스레드 간섭(thread interference)이란 서로 다른 스레드에서 실행되는 두 개의 연산이 동일한 데이터에 적용되면서 서로 겹치는 것
- (예) 카운터

```
class Counter {  
    private int value = 0;  
    public void increment() { value++; }  
    public void decrement() { value--; }  
    public void printCounter() { System.out.println(value); }  
}
```

- 하나의 Counter 객체를 두개의 스레드가 공유하면서 카운터 값을 변경한다고 가정하자.

- ① 변수 value의 현재값을 가져온다.
- ② 현재값을 1만큼 증가시킨다.
- ③ 증가된 값을 다시 변수 value에 저장한다.



# 스레드 간섭

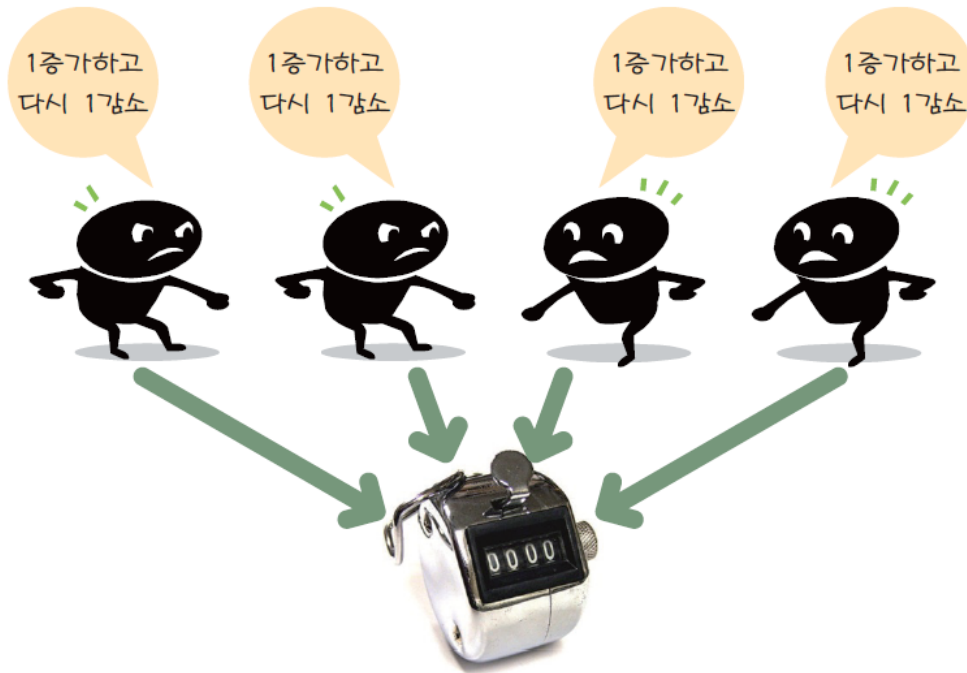
---

- 만약 스레드 A가 `increment()`를 호출하고 동시에 스레드 B가 `decrement()`를 호출하였다고 가정하자.

- ① 스레드 A: 변수 `value`의 현재값을 가져온다.
- ② 스레드 B: 변수 `value`의 현재값을 가져온다.
- ③ 스레드 A: 가져온 값을 1증가한다. 증가된 값은 1이 된다.
- ④ 스레드 B: 가져온 값을 1감소한다. 감소된 값은 -1이 된다.
- ⑤ 스레드 A: `value`에 값을 저장한다. `value`는 1이 된다.
- ⑥ 스레드 B: `value`에 값을 저장한다. `value`는 -1이 된다.

# 문제 발생

- 다음과 같은 상황을 가정하자.
- 4개의 스레드가 하나의 카운터를 증가했다가 다시 감소한다.



# 예제

CounterTest.java

```
01 class Counter {
02     private int value = 0;
03     public void increment() { value++; }
04     public void decrement() { value--; }
05     public void printCounter() { System.out.println(value); }
06 }
07
08 class MyThread extends Thread {
09     Counter sharedCounter;
10
11     public MyThread(Counter c) {
12         this.sharedCounter = c;
13     }
14
15     public void run() {
16         int i = 0;
17         while (i < 20000) {
18             sharedCounter.increment();
19             sharedCounter.decrement();
20             if (i % 40 == 0)
21                 sharedCounter.printCounter();
22             try {
```

Counter를 정의한다.

공유된 Counter 객체의 참조값을 저장한다.

증가했다가 감소시키기 때문에 카운터의 값은 변화가 없어야 한다.

가끔 카운터의 값을 출력하여 본다.

# 예제

```
23         sleep((int) (Math.random() * 2)); ←----- 난수 시간만큼 스레드를 중지한다.
24     } catch (InterruptedException e) { }
25     i++;
26 }
27 }
28 }
29
30 public class CounterTest {
31     public static void main(String[] args) {
32         Counter c = new Counter(); ←----- 공유 카운터 객체를 생성한다.
33         new MyThread(c).start();
34         new MyThread(c).start(); ←----- 확실하게 잘못된 결과를 내기 위하여
35         new MyThread(c).start(); ←----- 스레드를 4개나 생성하여 실행한다.
36         new MyThread(c).start();
37     }
38 }
```

# 실행 결과

## 실행결과

...  
-7  
-7  
-7  
-8  
-7  
-7  
...

실행 결과는 컴퓨터와 상황에  
따라서 상당히 달라진다.  
스레드 간섭이 없다면 모두 0  
이 출력되어야 한다.





# 메모리 불일치 오류

---

- 메모리 불일치 오류는 서로 다른 스레드가 동일한 데이터의 값을 서로 다르게 볼 때, 발생한다.

```
int counter = 0;
```

- 스레드 A가 counter를 다음과 같이 증가하였다.

```
counter++;
```

- 잠시 후에 스레드 B가 counter의 값을 출력한다.

```
System.out.println(counter);
```



# 동기화된 메소드

- 동기화된 메소드를 만들기 위해서는 `synchronized` 키워드를 메소드 선언에 붙이면 된다.
- `synchronized` 키워드가 붙어 있으면 하나의 스레드가 공유 메소드를 실행하는 동안에 다른 스레드는 공유 메소드를 실행할 수 없다.

공유 데이터를 조작하는 메소드 앞에 `synchronized`를 붙인다.

```
class Counter {  
    private int value = 0;  
    public synchronized void increment() { value++; }  
    public synchronized void decrement() { value--;}  
    public synchronized void printCounter() { System.out.println(value); }  
}
```

# 실행 결과

## 실행결과

0  
0  
0  
0  
0  
0  
1  
0  
0  
...

가끔 1이 나오지만 다시 0이 출력된다. 1이  
가끔 나오는 이유는 하나의 스레드가 출력하  
기 직전에 다른 스레드가 1로 증가시켰기 때  
문이다. 즉 증가된 값이 출력되는 것이다.  
다시 0으로 감소된다.

# 중간 점검 문제



## 참고사항

중요한 예외는 `final` 필드이다. `final` 변수는 객체가 생성된 뒤에는 변경이 불가능하다. 따라서 동기화되지 않은 메소드를 사용해서도 안전하게 읽거나 쓸 수 있다.



## 중간점검

1. 동기화 문제는 근본적으로 왜 발생하는가?
2. 동기화 문제를 해결하는 키워드는 무엇인가?

# 스레드 간의 조정

- 두 개의 스레드가 데이터를 주고 받는 경우에 발생



생산자



버퍼



소비자

< 생산자와 소비자 문제 >

# 스레드 간의 조정

- polling

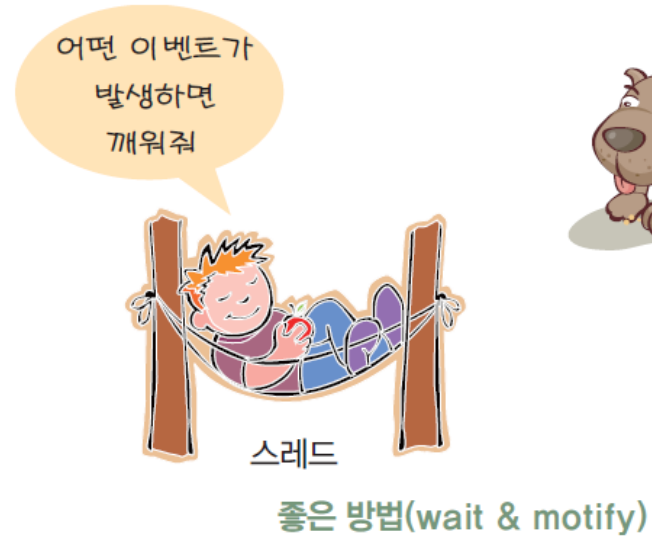
```
public void badMethod() {  
    // CPU 시간을 엄청나게 낭비한다.  
    // 절대 해서는 안 된다!  
    while(!condition) {  
        System.out.println("조건이 만족되었습니다!");  
    }  
}
```

- Event-driven

```
public synchronized goodMethod() {  
    while(!condition) {  
        try {  
            wait();  
        } catch (InterruptedException e) { }  
    }  
    System.out.println("조건이 만족되었습니다!");  
}
```

이벤트가 발생할 때까지 리턴하지 않는다.  
이벤트가 발생하면 깨어나서 다시 조건을  
체크한다.

# 스레드 간의 조정 방법





# 스레드 간의 조정 방법

---

- wait()가 호출되면 스레드는 가지고 있던 락(lock)을 해제하고 실행을 일시 중지한다.

```
public synchronized notifyCondition() {  
    condition = true;  
    notifyAll();  
}
```

# wait()와 notify()

Wait()하고 있을테니  
끝나면 notifyAll()해줘



synchronized



알았어...  
잊지 않고  
notifyAll()할게

< wait()와 notify() >





# 생산자/소비자 문제

- 생산자는 데이터를 생산하고 소비자는 데이터를 가지고 어떤 작업을 한다.
- 생산자-소비자 문제에서 중요한 것은 생산자가 생산하기 전에 소비자가 물건을 가져가면 안된다.
- 반대로 이전 물건을 소비하기 전에 생산하면 안 된다.
  - ☞ 동기화된 메소드를 사용하여 두 개의 스레드가 동시에 버퍼 객체에 접근하는 것을 막는다. 동기화된 메소드는 `synchronized` 키워드를 메소드 앞에 붙여서 만든다.
  - ☞ 케익을 생산하고 가져가는 동작을 일치시키기 위하여 두 개의 스레드를 동기화할 수 있는 어떤 방법이 필요하다. 스레드 간의 동작을 일치시키기 위해서 사용하는 메소드들이 **wait()**와 **notify()**이다. 이들 메소드를 이용하여 생산이 되었음을 소비자에게 명시적으로 알리고 또 한 소비가 되었음을 명시적으로 생산자에게 알릴 수 있다.

# 생산자/소비자 문제에 적용

- 먼저 케익을 임시적으로 보관하는 Buffer 클래스를 작성한다.

```
class Buffer {  
    private int data;  
    private boolean empty = true;  
  
    public synchronized int get() {  
        while (empty) {  
            try {  
                wait();  
            } catch (InterruptedException e) {  
            }  
        }  
        empty = true;  
        notifyAll();  
        return data;  
    }  
  
    public synchronized void put(int data) {  
        while (!empty) {  
            try {  
                wait();  
            } catch (InterruptedException e) {  
            }  
        }  
    }  
}
```

생산자로부터 소비자  
로 전해지는 데이터 → `private int data;`

소비가자가 기다리고 있으면 true,  
생산자가 기다리고 있으면 false ← `private boolean empty = true;`

케익이 생산될 때까지 기다린다. ← `while (empty) { ... }`

상태를 토글한다. → `empty = true;`

생산자를 깨운다. → `notifyAll();`

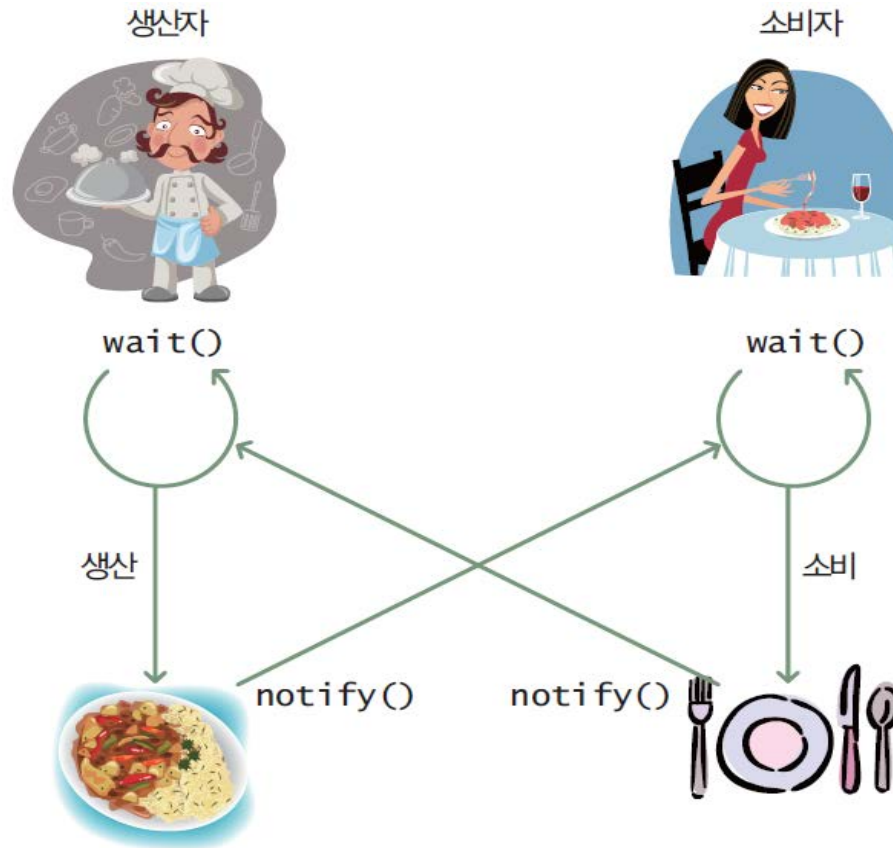


# 생산자/소비자 문제에 적용

---

```
    }  
    empty = false;  
    this.data = data;  
    notifyAll();  
  }  
}
```

# 생산자/소비자 문제에 적용



< 생산자와 소비자 문제 >

# 예제

- 생산자를 코딩하여 보자.

```
class Producer implements Runnable {  
    private Buffer buffer; ←----- 버퍼 참조 변수를 저장한다.  
  
    public Producer(Buffer buffer) {  
        this.buffer= buffer;  
    }  
  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            buffer.put(i); ←----- 버퍼에 케익을 가져다 놓는다.  
            System.out.println("생산자: " + i + "번 케익을 생산하였습니다.");  
            try {  
                Thread.sleep((int) (Math.random() * 100));  
            } catch (InterruptedException e) {  
            }  
        }  
    }  
}
```

- 이번에는 같은 방식으로 소비자 스레드를 작성한다.

```
class Consumer implements Runnable {  
    private Buffer buffer; ←----- 버퍼 참조 변수를 저장한다.
```

# 예제

```
public Consumer(Buffer drop) {
    this.buffer= drop;
}

public void run() {
    for (int i = 0; i < 10; i++) {
        int data = buffer.get();
        System.out.println("소비자: " + data + "번 케익을 소비하였습니다.");
        try {
            Thread.sleep((int) (Math.random() * 100));
        } catch (InterruptedException e) {
        }
    }
}
```

버퍼에서 케익을 가져온다.

- 마지막으로 메인 스레드를 작성하여 준다.

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        (new Thread(new Producer(buffer))).start();
        (new Thread(new Consumer(buffer))).start();
    }
}
```

# 실행 결과

## 실행결과

생산자: 0번 케익을 생산하였습니다.  
소비자: 0번 케익을 소비하였습니다.  
생산자: 1번 케익을 생산하였습니다.  
소비자: 1번 케익을 소비하였습니다.  
...  
생산자: 9번 케익을 생산하였습니다.  
소비자: 9번 케익을 소비하였습니다.

- 만약 wait(), notifyall()을 사용하지 않고 무조건 케익을 가져가고 생산한다면 다음과 같이 잘못된 결과가 나온다.

```
class Buffer {  
    private int data;  
  
    public synchronized int get() {  
        return data;  
    }  
  
    public synchronized void put(intdata) {  
        this.data = data;  
    }  
}
```

# 실행 결과

## 실행결과

생산자: 0번케익을생산하였습니다.

소비자: 0번케익을소비하였습니다.

소비자: 0번케익을소비하였습니다.

소비자: 0번케익을소비하였습니다.

생산자: 1번케익을생산하였습니다.

소비자: 1번케익을소비하였습니다.

...

←----- 똑같은 케익을 3번이나 가져간다.

## 중간점검



1. wait()와 notifyAll() 메소드는 왜 필요한가?
2. wait()는 어떤 역할을 하는가?
3. notifyAll()는 어떤 역할을 하는가?



# Q & A

