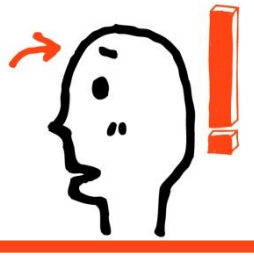
Android Java

23장 쓰레드(Thread)와 동기화

쓰레드(Thread)와 동기화



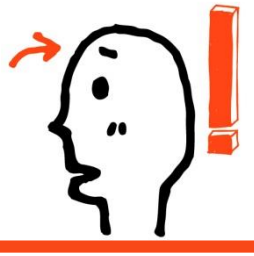
01 쓰레드의 이해와 생성

02 쓰레드의 특성

03 동기화(Synchronization)

04 새로운 동기화 방식

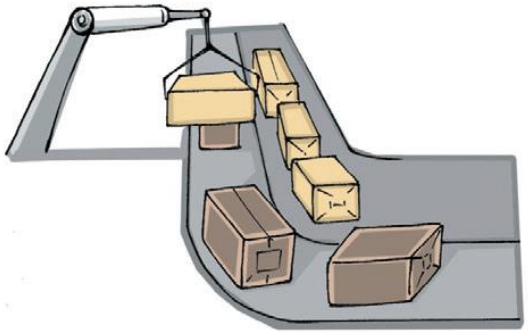
스레드 프로그래밍을 시작하기에 앞서



- 멀티태스킹
  - 하나의 응용프로그램이 여러 개의 작업(태스크)을 동시에 처리

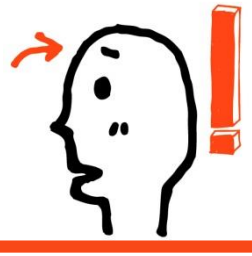
다림질하면서 이어폰으로
전화하는 주부

운전하면서
화장하는 운전자



제품의 판독과 포장 작업의
두 기능을 갖춘 기계

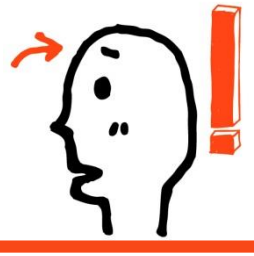
스레드 프로그래밍을 시작하기에 앞서



이해하고 넘어가야 할 용어들

- **멀티 태스킹(Multi-tasking) 환경** : 여러 가지 작업을 동시에 빠르게 처리하는 환경
- **태스크(Task)** : 일 혹은 작업이라고 하며, 프로세스와 스레드까지 의미
- **프로세스(Process)** : OS로부터 자원을 할당받아 동작하는 독립된 프로그램을 의미
- **스레드(Thread)** : 하나의 프로세스에서 실행하는 작업의 단위
- **메인 스레드(Main Thread)** : 기본적으로 여러분이 하나의 프로세스를 실행하면 하나의 스레드가 실행, 프로세스의 시작
- **싱글 스레드 프로세스(Single-Thread Process)** : 프로세스 내부에 하나의 스레드가 동작
- **멀티 스레드 프로세스(Multi-Thread Process)** : 프로세스 내부에 여러 개의 스레드가 동작

쓰레드의 이해와 Thread 클래스의 상속



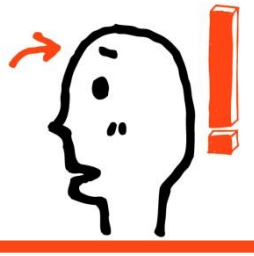
쓰레드와 프로세스의 이해 및 관계

- 프로세스는 실행중인 프로그램을 의미한다.
- 쓰레드는 프로세스 내에서 별도의 실행흐름을 갖는 대상이다.
- 프로세스 내에서 둘 이상의 쓰레드를 생성하는 것이 가능하다.

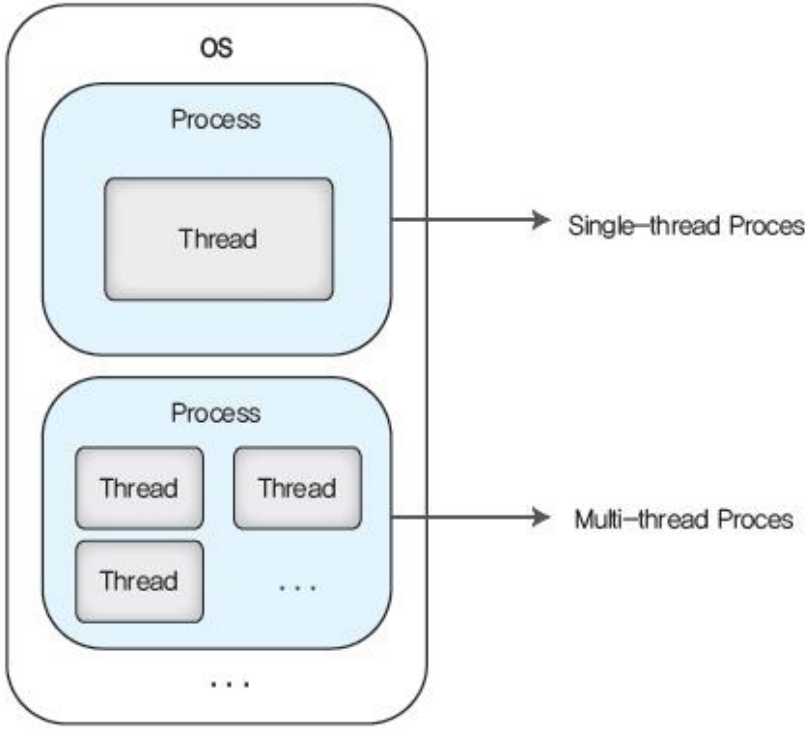
프로그램이 실행될 때 프로세스에 할당된 메모리, 이 자체를 단순히 프로세스라고 하기도 한다.

사실 쓰레드는 모든 일의 기본 단위이다. main 메소드를 호출하는 것도 프로세스 생성시 함께 생성되는 **main 쓰레드**를 통해서 이뤄진다.

스레드 프로그래밍을 시작하기에 앞서

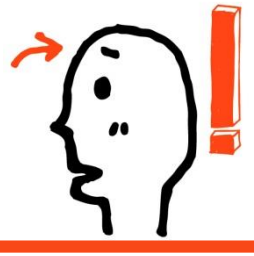


이해하고 넘어가야 할 용어들



△ 그림 9-1 싱글 스레드와 멀티 스레드의 차이

Thread 클래스를 이용한 스레드 생성



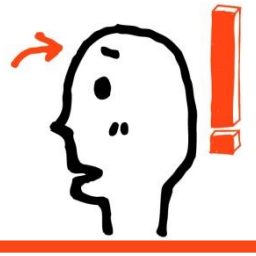
- 스레드 클래스 작성
  - Thread 클래스 상속. 새 클래스 작성
- 스레드 코드 작성
  - run() 메소드 오버라이딩
    - run() 메소드를 스레드 코드라고 부름
    - run() 메소드에서 스레드 실행 시작
- 스레드 객체 생성
- 스레드 시작
  - start() 메소드 호출
    - 스레드로 작동 시작
    - JVM에 의해 스케줄되기 시작함

```
class TimerThread extends Thread {
   .....

   public void run() { // run() 오버라이딩
      .....
   }
}
```

```
TimerThread th = new TimerThread();
```

```
th.start();
```

Thread 클래스를 이용한 스레드 생성



스레드 생성을 위한 Thread 클래스와 Runnable 인터페이스

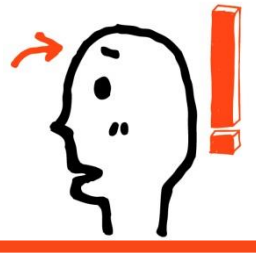
- 스레드를 생성하기 위해서는 Thread 클래스와 **Runnable 인터페이스**를 이용

코드 9-1 package com.gilbut.chapter9;

```
1   public class ThreadSample1
2   {
3       public static void main(String[] args)
4       {
5           Thread t = new Thread();
6           t.start();
7       }
8   }
```

- Thread 클래스를 사용해서 스레드를 만드는 예제
- Thread 객체를 만들고 start( ) 메소드를 실행하면 간단히 새로운 실행 흐름이 생성
- 아무 일도 하지 않는 스레드가 생성되고 바로 소멸
- 원하는 작업을 수행하는 스레드를 만들기 위해서는 Thread 클래스의 run( ) 메소드를 오버라이딩

Thread 클래스를 이용한 스레드 생성

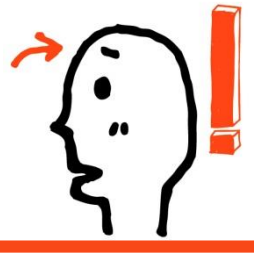


스레드 생성을 위한 Thread 클래스와 Runnable 인터페이스

코드 9-2 package com.gilbut.chapter9;

```
1   public class ThreadSample2 extends Thread
2   {
3       public static void main(String[] args)
4       {
5           ThreadSample2 sample2 = new ThreadSample2();
6           sample2.run();
7       }
8
9       //Thread 클래스의 run() 메소드를 오버라이딩한다.
10      public void run()
11      {
12          System.out.println("New flow, new thread is running");
13      }
14  }
```

Thread 클래스를 이용한 스레드 생성



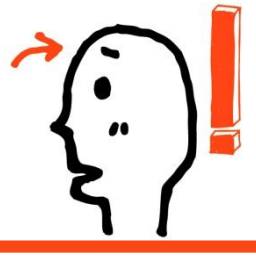
스레드 생성을 위한 Thread 클래스와 Runnable 인터페이스

실행결과

```
New flow, new thread is running
```

- extends 키워드로 Thread 클래스를 상속받고 있으며
  Thread 클래스 의 run( ) 메소드를 오버라이딩해서 새로운 일을 하도록 재정의
- 싱글 스레드 프로그래밍 예제

Thread 클래스를 이용한 스레드 생성



* Thread를 상속받아 1초 단위로 초 시간을 출력하는 TimerThread 스레드 작성

스레드 클래스 정의

스레드 코드 작성

1초에 한 번씩 n을 증가시켜 콘솔에 출력한다.

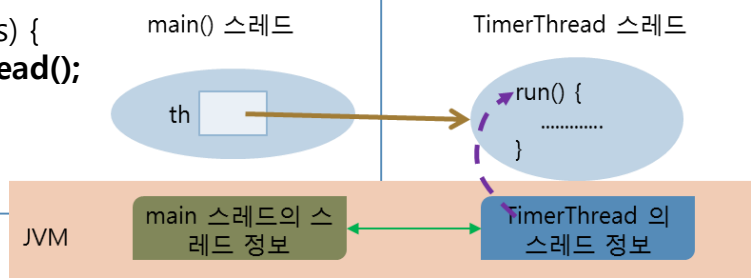
```
class TimerThread extends Thread {
    int n = 0;
    public void run() {
        while(true) { // 무한루프를 실행한다.
            System.out.println(n);
            n++;
            try {
                sleep(1000); //1초 동안 잠을 잔 후 깨어난다.
            }
            catch(InterruptedException e){return;}
        }
    }
}
```

```
0
1
2
3
4
```

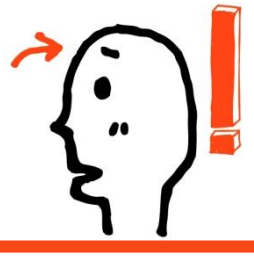
스레드 객체 생성

스레드 시작

```
public class TestThread {
    public static void main(String [] args) {
        TimerThread th = new TimerThread();
        th.start();
    }
}
```

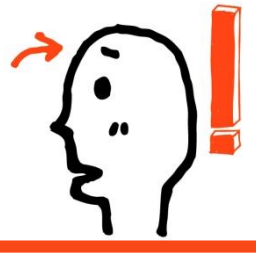


스레드 주의 사항



- run() 메소드가 종료하면 스레드는 종료한다.
  - 스레드가 계속 살아있도록 하려면 run() 메소드 내 무한 루프 구성
- 한번 종료한 스레드는 다시 시작시킬 수 없다.
  - 다시 스레드 객체를 생성하고 스레드로 등록하여야 한다.
- 한 스레드에서 다른 스레드를 강제 종료할 수 있다.
  - 뒤에서 다룸

Runnable 인터페이스로 스레드 만들기



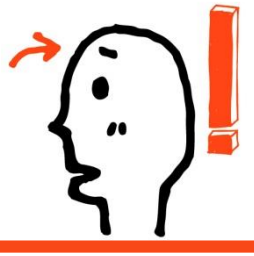
- 스레드 클래스 작성
  - Runnable 인터페이스 구현하는 새 클래스 작성
- 스레드 코드 작성
  - run() 메소드 구현
    - run() 메소드를 스레드 코드라고 부름
    - run() 메소드에서 스레드 실행 시작
- 스레드 객체 생성
- 스레드 시작
  - start() 메소드 호출

```
class TimerRunnable implements Runnable {
   ……
   public void run() { // run() 메소드 구현
      ……
   }
}
```

```
Thread th = new Thread(new TimerRunnable());
```

```
th.start();
```

Runnable 인터페이스로 스레드 만들기



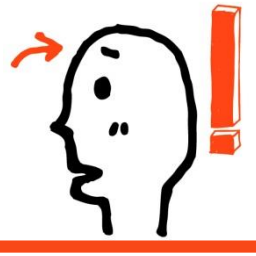
스레드 생성을 위한 Thread 클래스와 Runnable 인터페이스

코드 9-3 package com.gilbut.chapter9;

```
1   public class ThreadSample3 implements Runnable
2   {
3       public static void main(String[] args)
4       {
5           //ThreadSample3는 Runnable 인터페이스의 구현 클래스이므로
6           //아래와 같이 Runnable r로 업캐스팅이 가능하다.
7           Runnable r = new ThreadSample3();
8           Thread t = new Thread(r);
9           t.start();
10      }
11
12      @Override
13      public void run()
14      {
15          System.out.println("New flow, new thread is running");
16      }
17  }
```

스레드를 실행하기 위해 start( ) 메소드를 사용했습니다. 코드 9-2와 비교했을 때 Thread 클래스를 상속하는 것과 Runnable 인터페이스의 구현하는 차이

Runnable 인터페이스로 스레드 만들기



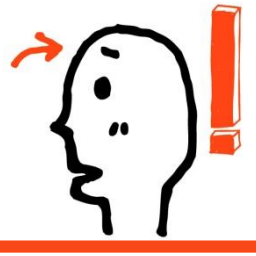
스레드 생성을 위한 Thread 클래스와 Runnable 인터페이스

실행결과

```
New flow, new thread is running
```

- Runnable 인터페이스를 사용하여 스레드를 만드는 예제
- Runnable 인터페이스를 구현한 경우 ThreadSample3 클래스 객체 r을 인자로 넘김
- 스레드 객체는 생성되었지만 start( ) 메소드를 호출하기 전까지 아무 일도 일어나지 않음
- start( ) 메소드를 호출해야 새로운 실행 흐름이 생성되고 구현한 내용이 각각 실행

Runnable 인터페이스로 스레드 만들기



*Runnable 인터페이스를 상속받아 1초 단위로 초 시간을 출력하는 스레드 작성

Runnable 을 클래스로 구현

스레드 코드 작성

1초에 한 번씩 n을 증가시켜 콘솔에 출력한다.

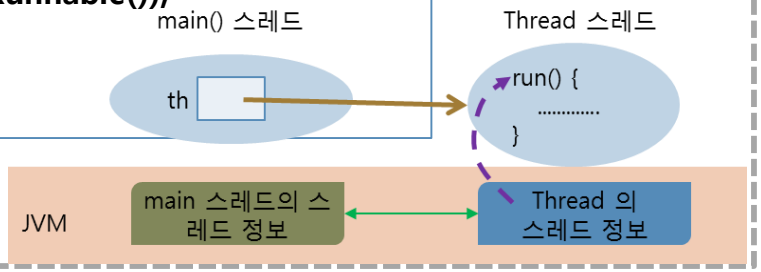
```
class TimerRunnable implements Runnable {
    int n = 0;
    public void run() {
        while(true) { // 무한루프를 실행한다.
            System.out.println(n);
            n++;
            try {
                Thread.sleep(1000); // 1초 동안 잠을 잔 후 깨어난다.
            }
            catch(InterruptedException e) { return; }
        }
    }
}
```

```
0
1
2
3
4
```

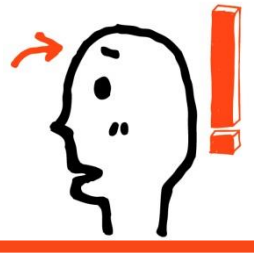
스레드 객체 생성

스레드 시작

```
public class TestRunnable {
    public static void main(String [] args) {
        Thread th = new Thread(new TimerRunnable());
        th.start();
    }
}
```



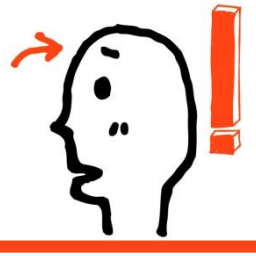
Runnable 인터페이스로 스레드 만들기



스레드 생성을 위한 Thread 클래스와 Runnable 인터페이스

- Runnable은 인터페이스이기 때문에 반드시 implements 키워드를 사용해서 구현
- 내부에는 run( ) 메소드가 선언되어 있어 구현 클래스는 반드시 그 내용을 구현해야 함
  - 자바는 단일 상속만 가능하므로 Thread 클래스를 상속한 경우 확장성이 떨어짐
  - Runnable 인터페이스를 구현한 경우 다형성 측면에서 유리
  - 재사용 및 확장에 좋음
  - 일반적인 경우 Runnable 인터페이스를 구현하는 쪽을 권장

Runnable 인터페이스로 스레드 만들기

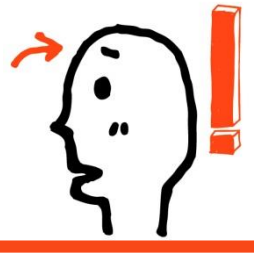


run() 메소드와 start() 메소드

코드 9-4 package com.gilbut.chapter9;

```
1   public class MyThread implements Runnable
2   {
3       @Override
4       public void run()
5       {
6           while (true)
7           {
8               System.out.println("is running...");
9               try
10              {
11                  Thread.sleep(1000);
12              }
13              catch (InterruptedException e)
14              {
15                  e.printStackTrace();
16              }
17          }
18      }
19  }
```

Runnable 인터페이스로 스레드 만들기

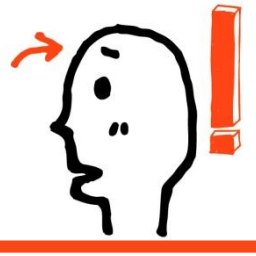


run() 메소드와 start() 메소드

코드 9-5 package com.gilbut.chapter9;

```
1   public class ThreadSample4
2   {
3       public static void main(String[] args)
4       {
5           Thread t = new Thread(new MyThread());
6           t.start();
7           System.out.println("MainThread terminated");
8       }
9   }
```

Runnable 인터페이스로 스레드 만들기



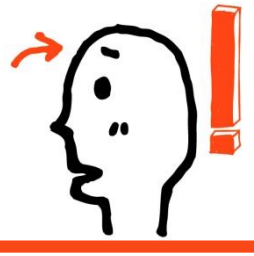
run() 메소드와 start() 메소드

실행결과

```
MainThread terminated
is running...
is running...
is running...
is running...
is running...
is running...
is running...
      ⋮
```

- MyTherad 클래스를 스레드로 실행하기 위한 클래스

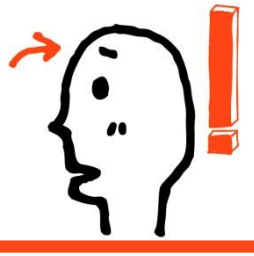
Runnable 인터페이스로 스레드 만들기



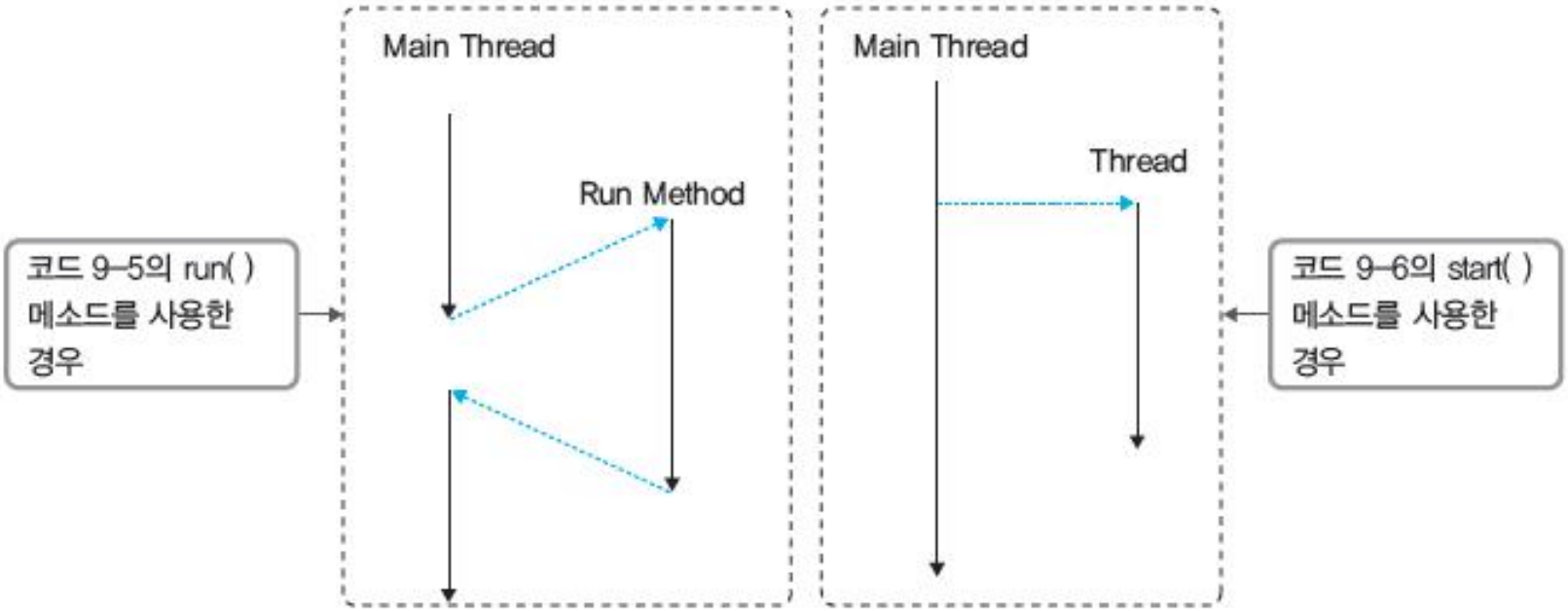
run() 메소드와 start() 메소드

- MyThread 클래스는 Runnable 인터페이스를 구현한 구현 클래스이며 run( ) 메소드 구현
- run( ) 메소드에 구현된 내용은 무한 반복하는 while( ) 구문을 실행하는 것
- Thread 클래스의 sleep( ) 메소드는 일정 시간 작업을 멈추게 하는 메소드
- 밀리 초 단위의 숫자를 매개변수로 받음
- run( ) 메소드를 실행 하면 매초마다 "is running"이라는 구문을 화면에 출력

Runnable 인터페이스로 스레드 만들기

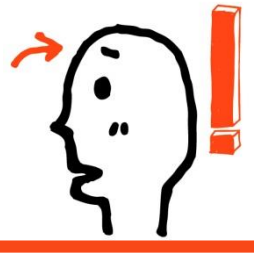


run() 메소드와 start() 메소드



△ 그림 9-2 run( ) 메소드와 start( ) 메소드의 실행 흐름

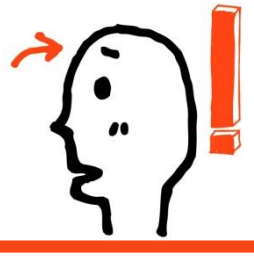
Runnable 인터페이스로 스레드 만들기



run() 메소드와 start() 메소드

- 모든 스레드는 독립적인 실행 흐름으로
  부모 스레드가 종료되어도 자식 스레드의 실행 흐름에 영향을 미치지 않는다는 것
- 프로세스 시작 시 최초 실행되는 static main( ) 메소드는
  JVM에 의해 생성된 main( ) 스레드에서 실행
- 모든 메소드는 각각의 실행 흐름 즉, 스레드 내에서 동작
- 그래서 모든 스레드가 종료되어야 프로그램이 종료

Runnable 인터페이스로 스레드 만들기



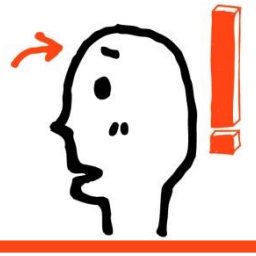
run() 메소드와 start() 메소드

코드 9-7 package com.gilbut.chapter9;

```
1   public class ChildThread implements Runnable
2   {
3       public static void main(String[] args)
4       {
5           Thread th = new Thread(new ChildThread());
6           th.start();
7
8           System.out.println("Main thread is done");
9
10      }
11
```

start( ) 메소드를 사용해서 스레드를 실행하면 그림 9-2처럼 두 개의 흐름이 생긴 것을 확인할 수 있습니다.

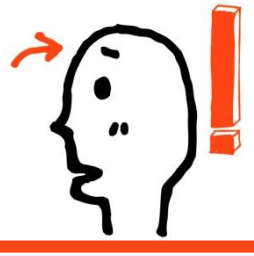
Runnable 인터페이스로 스레드 만들기



run() 메소드와 start() 메소드

```
12      @Override
13      public void run()
14      {
15          while (true)
16          {
17              System.out.println("is running...");
18
19              try
20              {
21                  Thread.sleep(1000);
22              }
23              catch (InterruptedException e)
24              {
25                  e.printStackTrace();
26              }
27          }
28      }
29  }
```

Runnable 인터페이스로 스레드 만들기



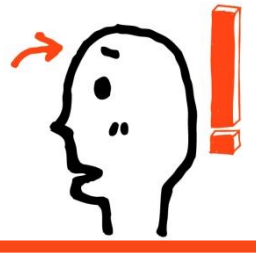
run() 메소드와 start() 메소드

실행결과

```
Main thread is done
is running...
is running...
is running...
```

- main 스레드가 종료되었음에도 ChildThread는 계속 실행
- 프로세스를 강제 종료하지 않는 이상 ChildThread의 run( ) 메소드가 종료될 때까지 실행
- 모든 실행 흐름이 끝나야 비로소 프로세스는 종료

쓰레드의 생성



```
class ShowThread extends Thread
{
    String threadName;
    public ShowThread(String name)
    {
        threadName=name;
    }
    public void run()
    {
        for(int i=0; i<100; i++)
        {
            System.out.println("안녕하세요. "+threadName+"입니다.");
            try
            {
                sleep(100);
            }
            catch(InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

쓰레드의 main 메소드가 run이다!

별도의 쓰레드 생성을 위해서는
별도의 쓰레드 클래스를
정의해야 한다.
쓰레드 클래스는 **Thread**를
상속하는 클래스를 의미한다.

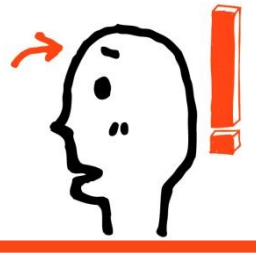
실행결과

```
안녕하세요. 예쁜 쓰레드입니다.
안녕하세요. 멋진 쓰레드입니다.
안녕하세요. 멋진 쓰레드입니다.
안녕하세요. 예쁜 쓰레드입니다.
안녕하세요. 멋진 쓰레드입니다.
안녕하세요. 예쁜 쓰레드입니다.
안녕하세요. 예쁜 쓰레드입니다.
안녕하세요. 멋진 쓰레드입니다.
. . . . . 중 략 . . . . . .
```

```
public static void main(String[] args)
{
    ShowThread st1=new ShowThread("멋진 쓰레드");
    ShowThread st2=new ShowThread("예쁜 쓰레드");
    st1.start();
    st2.start();
}
```

start 메소드가 호출되면 쓰레드가 생성되고, 생성된 쓰레드는 run 메소드를 호출한다.

쓰레드의 생성

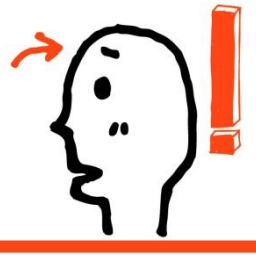


싱글 스레드 vs 멀티 스레드

코드 9-8 package com.gilbut.chapter9;

```
1   public class ThreadSample6
2   {
3      public static void main(String[] args)
4      {
5         Thread th1 = new Thread(new ChildWorker1());
6         Thread th2 = new Thread(new ChildWorker2());
7
8         th1.start();
9         th2.start();
10
11        System.out.println("main thread terminated");
12     }
13  }
14
```

쓰레드의 생성

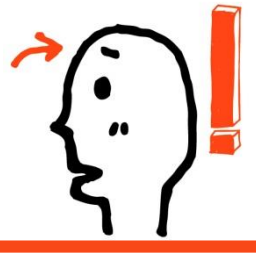


싱글 스레드 vs 멀티 스레드

```
15  class ChildWorker1 implements Runnable
16  {
17      @Override
18      public void run()
19      {
20          while (true)
21          {
22              System.out.println("Working <<<<<<<<");
23
24              try
25              {
26                  // 작업 지연
27                  Thread.sleep(1000);
28              }
29              catch (InterruptedException e)
30              {
31                  e.printStackTrace();
32              }
33          }
34      }
35  }
```

1초마다 "Working <<<<" 메시지를 출력하는 기능을 수행할 클래스. 5라인에서 실행합니다.

쓰레드의 생성

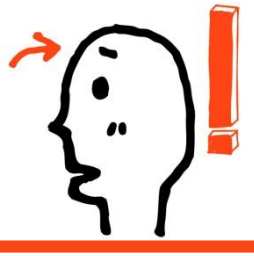


싱글 스레드 vs 멀티 스레드

```
36
37  class ChildWorker2 implements Runnable
38  {
39      @Override
40      public void run()
41      {
42          while (true)
43          {
44              System.out.println("<<<<<<<< Working");
45
46              try
47              {
48                  // 작업 지연
49                  Thread.sleep(1000);
50              }
51              catch (InterruptedException e)
52              {
53                  e.printStackTrace();
54              }
55          }
56      }
57  }
```

1초마다 "<<<< Working" 메시지를 출력합니다. main( ) 메소드의 6라인에서 스레드로 실행시키는 것을 확인해보세요.

쓰레드의 생성



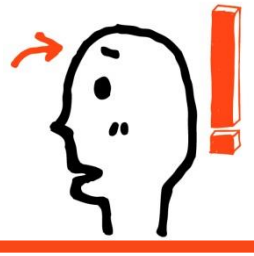
싱글 스레드 vs 멀티 스레드

실행결과

```
main thread terminated
Working <<<<<<<<
<<<<<<<< Working
<<<<<<<< Working
Working <<<<<<<<
<<<<<<<< Working
Working <<<<<<<<
<<<<<<<< Working
Working <<<<<<<<
```

- Working 키워드를 왼쪽, 오른쪽에 각각 출력하는 스레드
- 사용자가 멀티 스레드 프로그램을 작성할 때 각 스레드의 실행 순서를 결정할 수 없음

쓰레드의 생성



❖ 스레드의 이름

- 메인 스레드 이름: main
- 작업 스레드 이름 (자동 설정) : Thread-n

```
thread.getName();
```

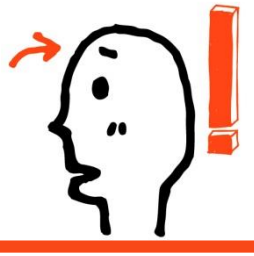
- 작업 스레드 이름 변경

```
thread.setName("스레드  이름");
```

- 코드 실행하는 현재 스레드 객체의 참조 얻기

```
Thread thread = Thread.currentThread();
```

쓰레드를 생성하는 두 번째 방법



RunnableThread.java

```
class Sum
{
    int num;
    public Sum() { num=0; }
    public void addNum(int n) { num+=n; }
    public int getNum() { return num; }
}

class AdderThread extends Sum implements Runnable
{
    int start, end;

    public AdderThread(int s, int e)
    {
        start=s;
        end=e;
    }
    public void run()
    {
        for(int i=start; i<=end; i++)
            addNum(i);
    }
}
```

Runnable 인터페이스를 구현하는 클래스의 인스턴스를 대상으로 Thread 클래스의 인스턴스를 생성한다. 이 방법은 상속할 클래스가 존재할 때 유용하게 사용된다.

```
public static void main(String[] args)
{
    AdderThread at1=new AdderThread(1, 50);
    AdderThread at2=new AdderThread(51, 100);
    Thread tr1=new Thread(at1);
    Thread tr2=new Thread(at2);
    tr1.start();
    tr2.start();

    try
    {
        tr1.join();
        tr2.join();
    }
    catch(InterruptedException e)
    {
        e.printStackTrace();
    }
    System.out.println("1~100까지의 합 : "+(at1.getNum()+at2.getNum()));
}
```

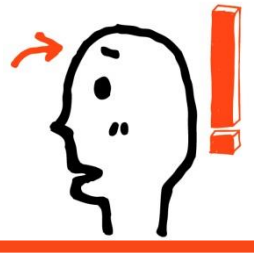
join 메소드가 호출되면, 해당 쓰레드의 종료를 기다리게 된다!

실행결과

```
1~100까지의 합 : 5050
```

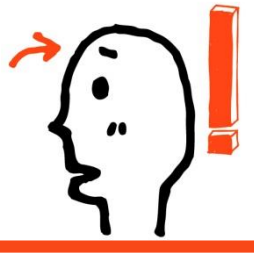
위 예제에서 main 쓰레드가 join 메소드를 호출하지 않았다면, 추가로 생성된 두 쓰레드가 작업을 완료하기 전에 값을 참조하여 쓰레기 값이 출력될 수 있다.

스레드 상태

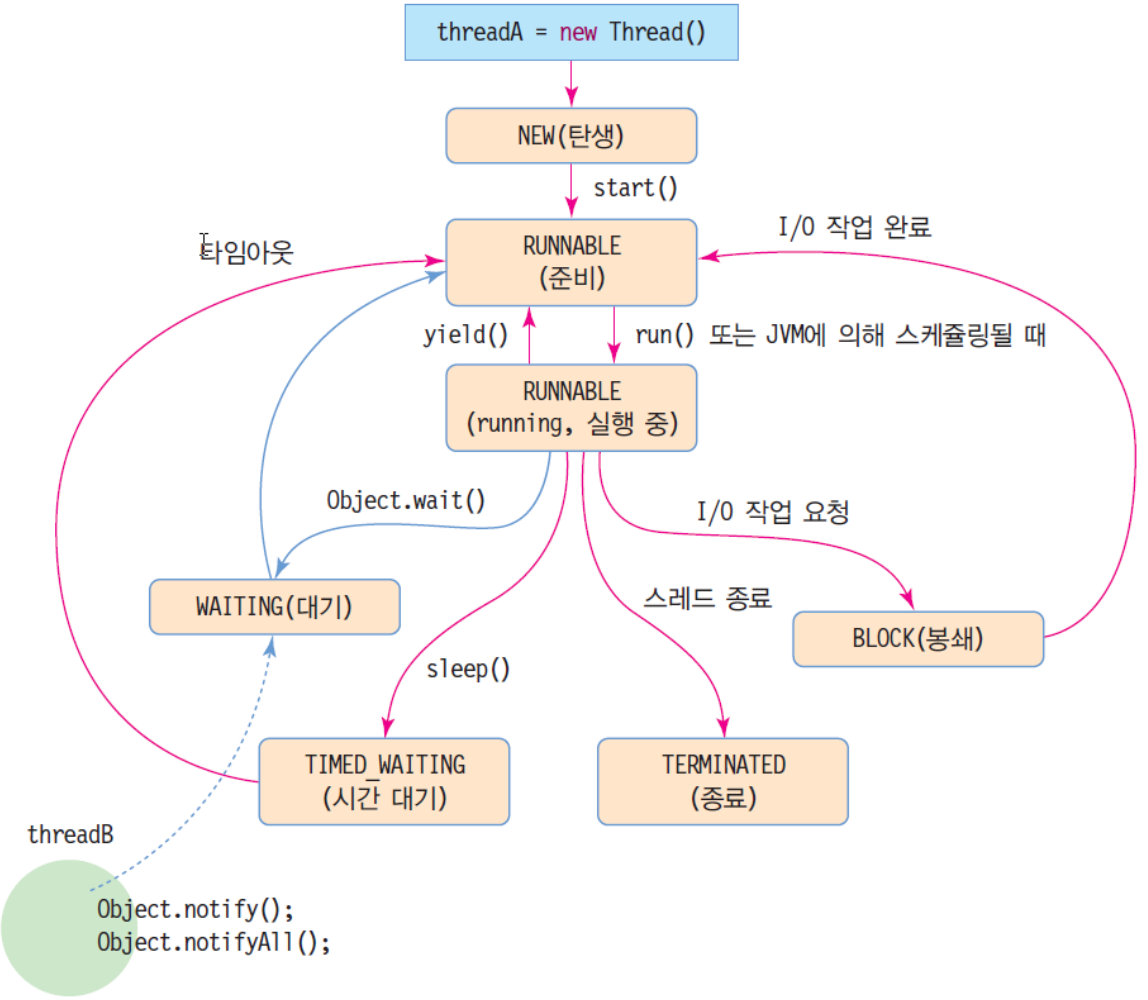


- 스레드 상태 6 가지
  - NEW
    - 스레드가 생성되었지만 스레드가 아직 실행할 준비가 되지 않았음
  - RUNNABLE
    - 스레드가 JVM에 의해 실행되고 있거나 실행 준비되어 스케쥴링을 기다리는 상태
  - WAITING
    - 다른 스레드가 notify(), notifyAll()을 불러주기를 기다리고 있는 상태
    - 스레드 동기화를 위해 사용
  - TIMED_WAITING
    - 스레드가 sleep(n)을 호출하여 n 밀리초 동안 잠을 자고 있는 상태
  - BLOCK
    - 스레드가 I/O 작업을 요청하면 JVM이 자동으로 이 스레드를 BLOCK 상태로 만듦
  - TERMINATED
    - 스레드가 종료한 상태
- 스레드 상태는 JVM에 의해 기록 관리됨

스레드 상태와 생명 주기

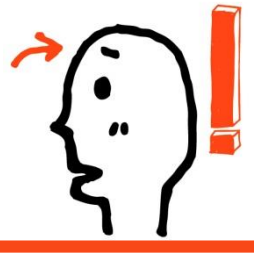


스레드 상태 6 가지
- NEW
- RUNNABLE
- WAITING
- TIMED_WAITING
- BLOCK
- TERMINATED



** wait(), notify(), notifyAll()은 Thread의 메소드가 아니며 Object의 메소드임

스레드 상태와 생명 주기

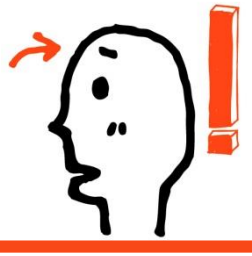


❖ 주어진 시간 동안 일시 정지 - sleep()

```
try {
   Thread.sleep(1000);
} catch(InterruptedException  e) {
   // interrupt()  메소드가  호출되면  실행
}
```

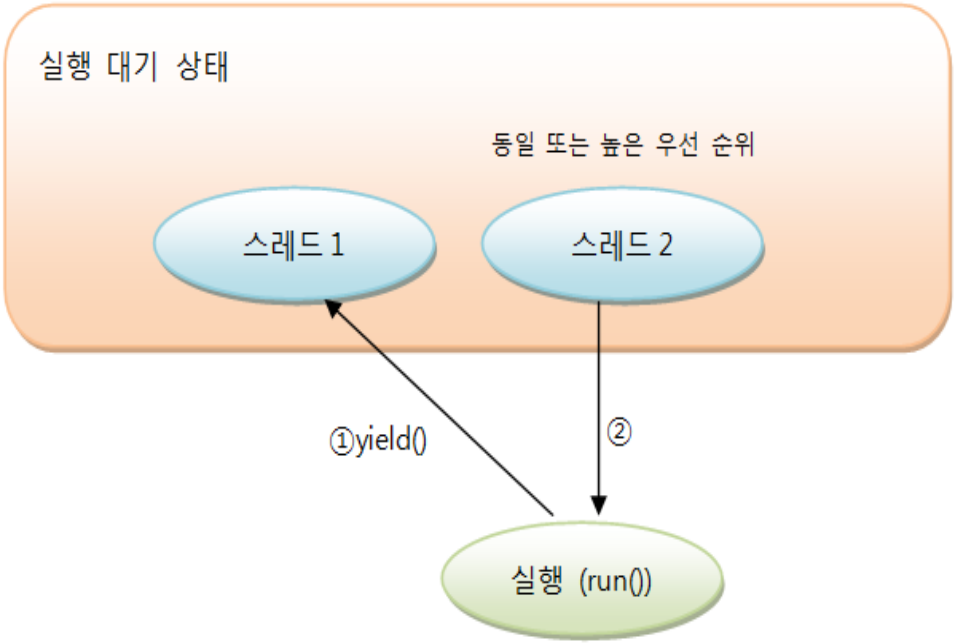
- 얼마 동안 일시 정지 상태로 있을 것인지 밀리 세컨드(1/1000) 단위로 지정
- 일시 정지 상태에서 interrupt() 메소드 호출
  - InterruptedException 발생

스레드 상태와 생명 주기

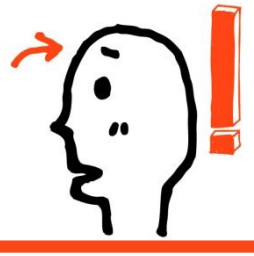


❖ 다른 스레드에게 실행 양보 – yield()

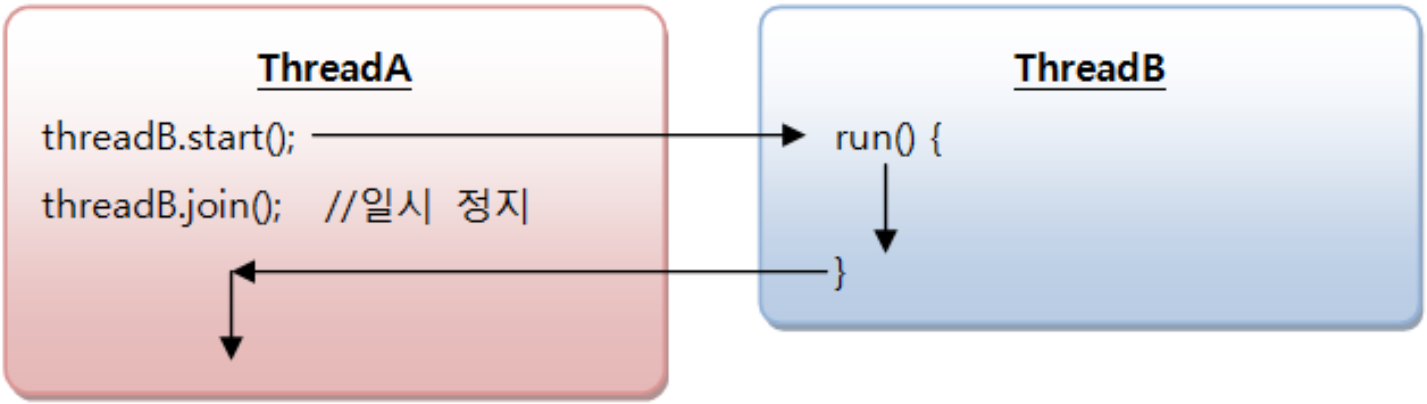
- Ex) 무의미한 반복하는 스레드일 경우



스레드 상태와 생명 주기

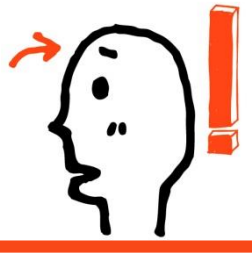


❖ 다른 스레드의 종료를 기다림 – join()



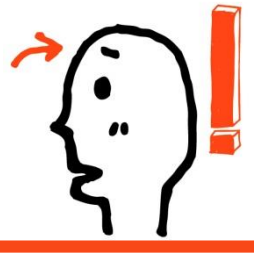
- 계산 작업을 하는 스레드가 모든 계산 작업 마쳤을 때, 결과값을 받아 이용하는 경우 주로 사용

스레드 우선순위와 스케쥴링



- 스레드의 우선순위
  - 최대값 = 10(MAX_PRIORITY)
  - 최소값 = 1(MIN_PRIORITY)
  - 보통값 = 5(NORMAL_PRIORITY)
- 스레드 우선순위는 응용프로그램에서 변경 가능
  - void setPriority(int priority)
  - int getPriority()
- main() 스레드의 우선순위 값은 초기에 5
- 스레드는 부모 스레드와 동일한 우선순위 값을 가지고 탄생
- JVM의 스케쥴링 정책
  - 철저한 우선순위 기반
    - 가장 높은 우선순위의 스레드가 우선적으로 스케쥴링
    - 동일한 우선순위의 스레드는 돌아가면서 스케쥴링(라운드 로빈)

main()은 자바의 main 스레드

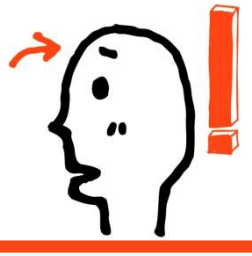


- main 스레드와 main() 메소드
  - JVM은 응용프로그램이 실행될 때 스레드 생성 : main 스레드
  - main 스레드에 의해 main() 메소드 실행
    - main() 메소드가 종료하면 main 스레드 종료

```
public class ThreadMainEx {
    public static void main(String [] args) {
        long id = Thread.currentThread().getId();
        String name = Thread.currentThread().getName();
        int priority = Thread.currentThread().getPriority();
        Thread.State s = Thread.currentThread().getState();

        System.out.println("현재 스레드 이름 = " + name);
        System.out.println("현재 스레드 ID = " + id);
        System.out.println("현재 스레드 우선순위 값 = " + priority);
        System.out.println("현재 스레드 상태 = " + s);
    }
}
```

```
현재 스레드 이름 = main
현재 스레드 ID = 1
현재 스레드 우선순위 값 = 5
현재 스레드 상태 = RUNNABLE
```

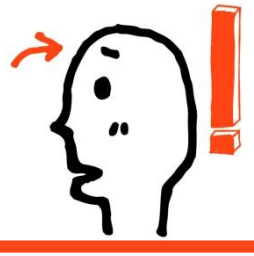
스레드 스케줄러



스레드 스케줄러가 하는 일

- **스레드 스케줄러** : 스레드를 관리하는 프로세스
- 여러 스레드들이 서로 공평하게 시스템 리소스를 사용하여
  동작할 수 있도록 조정하는(스케줄링) 역할을 담당
- 스레드 스케줄러는 Runnable 상태로 대기하고 있는 스레드 중
  우선 순위와 대기 시간을 고려 하여 다음 실행 권한을 얻을 스레드를 결정

스레드 직접 만들고 다뤄보기



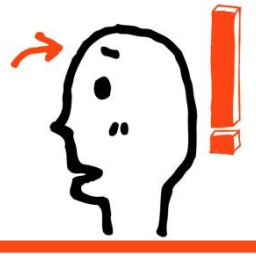
스레드 스케줄러가 하는 일

코드 9-10 package com.gilbut.chapter9;

```
1   public class ThreadSample8 implements Runnable
2   {
3       public static void main(String[] args)
4       {
5           Thread th = new Thread(new ThreadSample8());
6           th.start();
7           th.start();
8       }
9
```

이렇게 두 번 start( ) 메소드를 호출하면 IllegalThreadStateException이 발생합니다!

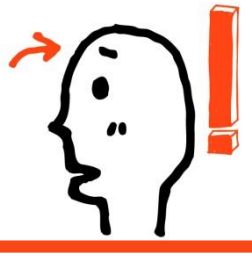
스레드 직접 만들고 다뤄보기



스레드 스케줄러가 하는 일

```
10      @Override
11      public void run()
12      {
13          try
14          {
15              Thread.sleep(5000);
16              System.out.println("Working now");
17          }
18          catch (InterruptedException e)
19          {
20              e.printStackTrace();
21          }
22      }
23  }
```

스레드 직접 만들고 다뤄보기



스레드 스케줄러가 하는 일

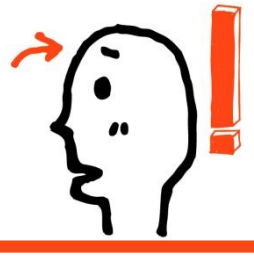
실행결과

```
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Thread.java:682)
    at com.gilbut.chapter9.ThreadSample8.main(ThreadSample8.java:9)
```

- main( ) 메소드에서 start( ) 메소드를 두 번 호출
- 첫 번째 호출한 start( ) 메소드에 의해서 스레드의 상태가 Runnable로 변경된 후 다시 start( ) 메소드가 호출되었기 때문에

  → IllegalThreadState Exception 예외가 발생

스레드 직접 만들고 다뤄보기

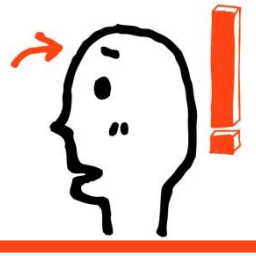


스레드의 우선 순위와 실행 시간

코드 9-12 package com.gilbut.chapter9;

```
1   public class ThreadSample10 implements Runnable
2   {
3       public static void main(String[] args)
4       {
5           Thread th = new Thread(new ThreadSample10());
6           th.setPriority(3);
7           th.start();
8
9           try
10          {
11              Thread.sleep(500);
12          }
13          catch (InterruptedException e)
14          {
15              e.printStackTrace();
16          }
17
18          th.setPriority(10);
19      }
20
```

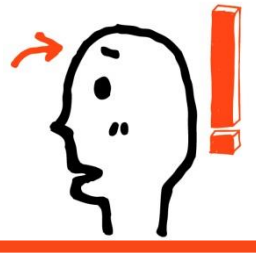
스레드 직접 만들고 다뤄보기



스레드의 우선 순위와 실행 시간

```
21      @Override
22      public void run()
23      {
24          while (true)
25          {
26              System.out.println("Priority : " + Thread.currentThread().getPriority());
27
28              try
29              {
30                  Thread.sleep(100);
31              }
32              catch (InterruptedException e)
33              {
34                  e.printStackTrace();
35              }
36          }
37      }
38  }
```

스레드 직접 만들고 다뤄보기



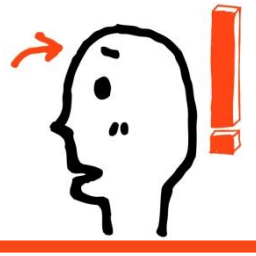
스레드의 우선 순위와 실행 시간

실행결과

```
Priority : 3
Priority : 3
Priority : 3
Priority : 3
Priority : 3
Priority : 10
Priority : 10
Priority : 10
      ⋮
```

- sleep( ) 메소드를 사용해서 sleep할 시간까지 스레드의 상태를 정확하게 제어할 수 있음

쓰레드의 스케줄링과 우선순위 컨트롤



쓰레드 스케줄링의 두 가지 기준

- ✓ 우선순위가 높은 쓰레드의 실행을 우선시한다.
- ✓ 우선순위가 동일할 때는 CPU의 할당시간을 나눈다.

PriorityTestOne.java

```
class MessageSendingThread extends Thread
{
    String message;

    public MessageSendingThread(String str)
    {
        message=str;
    }
    public void run()
    {
        for(int i=0; i<1000000; i++)
            System.out.println(message+"("+getPriority()+")");
    }
}
```

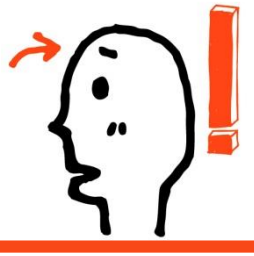
```
public static void main(String[] args)
{
    MessageSendingThread tr1=new MessageSendingThread("First");
    MessageSendingThread tr2=new MessageSendingThread("Second");
    MessageSendingThread tr3=new MessageSendingThread("Third");
    tr1.start();
    tr2.start();
    tr3.start();
}
```

실행결과

```
First(5)
First(5)
Second(5)
. . . . . .
Third(5)
First(5)
. . . . . .
Third(5)
```

메소드 getPriority의 반환값을 통해서 쓰레드의 우선순위를 확인할 수 있다. 왼쪽의 실행결과에서 보이듯이, 우선순위와 관련해서 별도의 지시를 하지 않으면, 동일한 우선순위의 쓰레드들이 생성된다.

우선순위가 다른 쓰레드들의 실행



PriorityTestTwo.java

```
class MessageSendingThread extends Thread
{
    String message;
    public MessageSendingThread(String str, int prio)
    {
        message=str;
        setPriority(prio);
    }
    public void run()
    {
        for(int i=0; i<1000000; i++)
            System.out.println(message+"("+getPriority()+")");
    }
}
```

```
public static void main(String[] args)
{
    MessageSendingThread tr1
        =new MessageSendingThread("First", Thread.MAX_PRIORITY);
    MessageSendingThread tr2
        =new MessageSendingThread("Second", Thread.NORM_PRIORITY);
    MessageSendingThread tr3
        =new MessageSendingThread("Third", Thread.MIN_PRIORITY);
    tr1.start();
    tr2.start();
    tr3.start();
}
```

실행 결과

```
First(10)
First(10)
. . . . . .
Second(5)
Second(5)
. . . . . .
Third(1)
```

Thread.MAX_PRIORITY는 상수로 10,

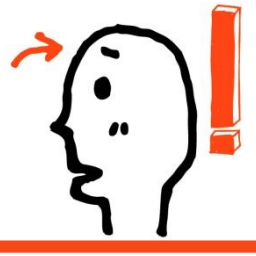
Thread.NORM_PRIORITY는 상수로 5,

Thread.MIN_PRIORITY는 상수로 1

실행결과에서 보이듯이 쓰레드의 실행시간은 우선순위의 비율대로 나뉘지 않는다.

높은 우선순위의 쓰레드가 종료되어야 낮은 우선순위의 쓰레드가 실행된다.

낮은 우선순위의 쓰레드 실행



PriorityTestThree.java

```
class MessageSendingThread extends Thread
{
    String message;

    public MessageSendingThread(String str, int prio)
    {
        message=str;
        setPriority(prio);
    }
    public void run()
    {
        for(int i=0; i<1000000; i++)
        {
            System.out.println(message+"("+getPriority()+")");
            try
            {       CPU를 양보!
                sleep(1);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

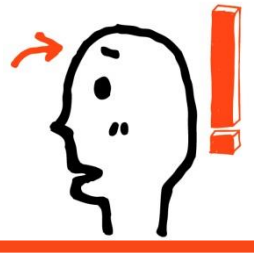
실행결과

```
Third(1)
First(10)
Third(1)
Second(5)
First(10)
Third(1)
Second(5)
. . . . . .
```

쓰레드가 CPU의 할당을 필요로 하지 않을 경우, CPU를 다른 쓰레드에게 양보한다.

```
public static void main(String[] args)
{
    MessageSendingThread tr1
        =new MessageSendingThread("First", Thread.MAX_PRIORITY);
    MessageSendingThread tr2
        =new MessageSendingThread("Second", Thread.NORM_PRIORITY);
    MessageSendingThread tr3
        =new MessageSendingThread("Third", Thread.MIN_PRIORITY);

    tr1.start();
    tr2.start();
    tr3.start();
}
```

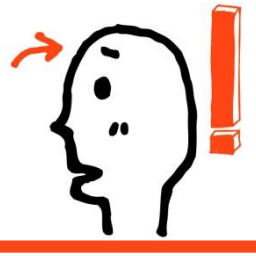
스레드 종료하기



인터럽트가 발생했을 경우

- 인터럽트 메소드를 호출할 경우 스레드의 인터럽트 상태가 변경되고
  이를 감지하여 스레드가 종료되도록 함
- 일반적인 관행에 따라 인터럽트 발생 시 스레드가 종료되도록 구현
- Thread.interrupted( ) 메소드를 통해 현재 스레드에 인터럽트가 발생했는지 알 수 있음
- 메소드가 호출되면 인터럽트 상태는 초기화되어 다음 번 호출 시 false를 반환

스레드 종료하기

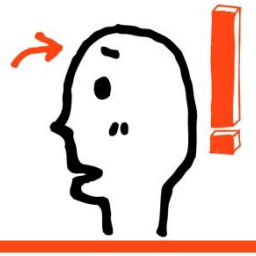


인터럽트가 발생했을 경우

코드 9-15 package com.gilbut.chapter9;

```
1   public class ThreadSample11 implements Runnable
2   {
3       public static void main(String[] args) throws InterruptedException
4       {
5           Thread th = new Thread
6           (new ThreadSample11());
7           th.start();
8
9           Thread.sleep(50);
10          // 작업 지연
11
12          th.interrupt();
13      }
```

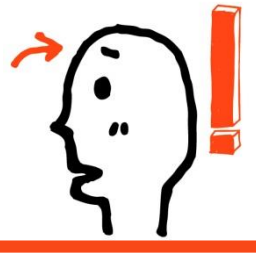
스레드 종료하기



인터럽트가 발생했을 경우

```
14
15      @Override
16      public void run()
17      {
18          while(true)
19          {
20              System.out.println("is running... " + Thread.interrupted());
21              // 작업 지연
22              for (int i=0; i < 10000000; i++)
23                  ;
24          }
25      }
26  }
```

스레드 종료하기



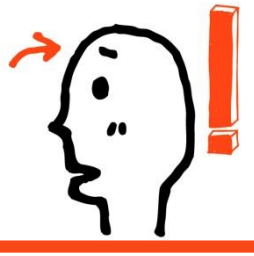
인터럽트가 발생했을 경우

실행결과

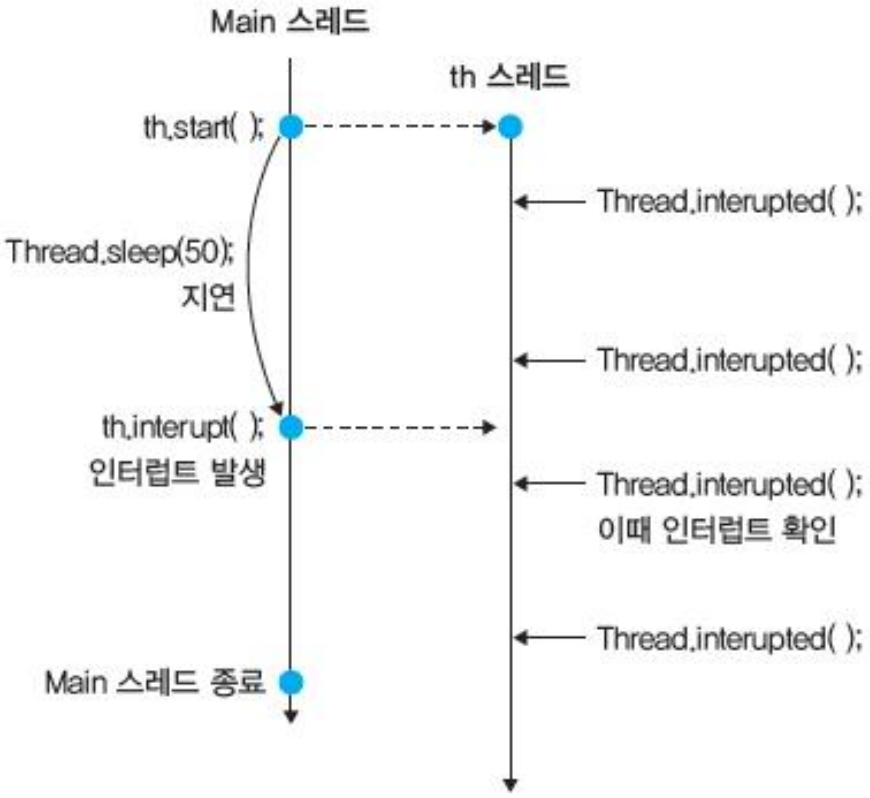
```
is running... false
is running... false
is running... false
is running... true
is running... false
is running... false
is running... false
```

- interrupt( ) 메소드를 사용해서 0.05초 뒤에 인터럽트를 발생하는 예제

스레드 종료하기

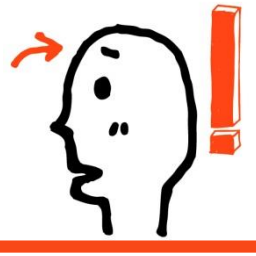


인터럽트가 발생했을 경우



△ 그림 9-6 ThreadSample11 클래스의 실행 흐름

스레드 종료하기

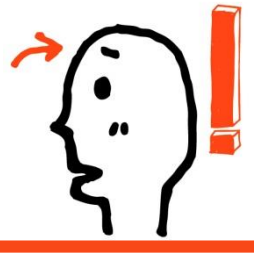


인터럽트가 발생했을 경우

코드 9-16 package com.gilbut.chapter9;

```
1   public class ThreadSample12 implements Runnable
2   {
3       public static void main(String[] args) throws InterruptedException
4       {
5           Thread th = new Thread(new ThreadSample12());
6           th.start();
7
8           Thread.sleep(50); // 작업 지연
9
10          th.interrupt();
11      }
12
```

스레드 종료하기

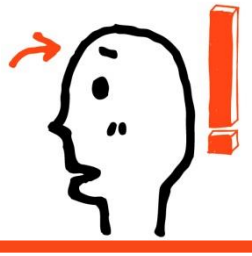


인터럽트가 발생했을 경우

```
13      @Override
14      public void run()
15      {
16          while (!Thread.interrupted())
17          {
18              System.out.println("is running... ");
19              // 작업 지연
20              for (int i = 0; i < 10000000; i++)
21                  ;
22          }
23
24          System.out.println("terminated…");
25      }
26  }
```

코드 9-14와 비교해보세요. Thread에 인터럽트가 걸리면 종료되는 구문입니다.

스레드 종료하기



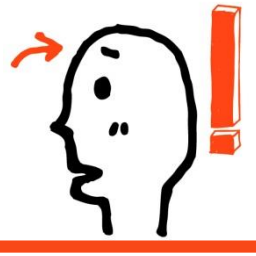
인터럽트가 발생했을 경우

실행결과

```
is running...
is running...
is running...
is running...
is running...
is running...
is running...
terminated…
```

- 인터럽트가 발생했을 때 스레드가 정상 종료하도록 구현한 것
- sleep( )이나 wait( ), join( ) 같은 메소드는 스레드가 다시 깨어날 때까지 인터럽트 상태를 확인 불가
- 따라서 인터럽트가 발생했을 때 깨어날 수 있도록 InterruptedException을 발생

스레드 종료하기

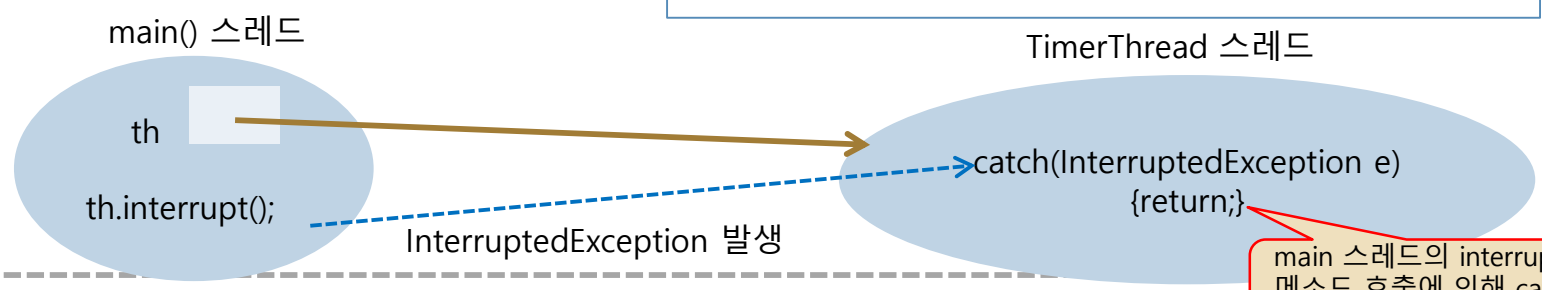


- 스스로 종료
  - run() 메소드 리턴
- 타 스레드에서 강제 종료 : interrupt() 메소드 사용

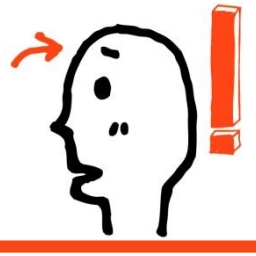
```
class TimerThread extends Thread {
    int n = 0;
    public void run() {
        while(true) {
            System.out.println(n); // 화면에 카운트 값 출력
            n++;
            try {
                sleep(1000);
            }
            catch(InterruptedException e){
                return; // 예외를 받고 스스로 리턴하여 종료
            }
        }
    }
}
```

만일 return 하지 않으면 스레드는 종료하지 않음

```
public static void main(String [] args) {
    TimerThread th = new  TimerThread();
    th.start();

    th.interrupt(); // TimerThread 강제 종료
}
```



스레드 종료하기

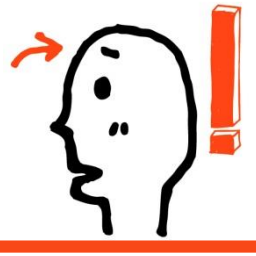


인터럽트가 발생했을 경우

코드 9-17 package com.gilbut.chapter9;

```
1   public class ThreadSample13 implements Runnable
2   {
3       public static void main(String[] args) throws InterruptedException
4       {
5           Thread th = new Thread(new ThreadSample13());
6           th.start();
7
8           Thread.sleep(500); // 작업 지연
9
10          th.interrupt();
11      }
12
```

스레드 종료하기

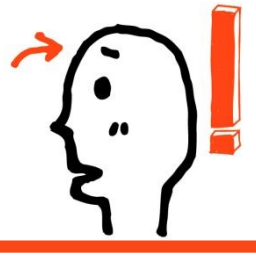


인터럽트가 발생했을 경우

```
13      @Override
14      public void run()
15      {
16          while(true)
17          {
18              System.out.println("is running... ");
19              // 작업 지연
20              try
21              {
22                  Thread.sleep(100);
23              }

24              catch(InterruptedException e)
25              {
26                  e.printStackTrace();
27                  break;
28              }
29          }
30
31          System.out.println("terminated…");
32      }
33  }
```

이렇게 Main 스레드에서 인터럽트를 걸면 예외 처리가 됩니다. 그러면 예외 처리 구문의 break;에 의해서 스레드가 종료됩니다.
코드 9-13의 TerminateSample1과 비교하면 전반적으로 비슷해 보이지만 th.interrupt( ); 구문이 있고 없고의 차이가 있습니다.

스레드 종료하기



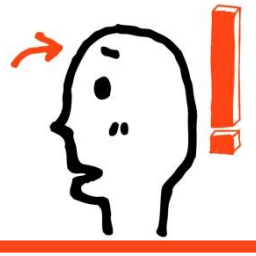
인터럽트가 발생했을 경우

실행결과

```
is running...
is running...
is running...
is running...
is running...
java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at com.gilbut.chapter9.ThreadSample13.run(ThreadSample13.java:24)
    at java.lang.Thread.run(Thread.java:722)
terminated…
```

- InterruptedException이 발생했을 경우 스레드가 정상 종료하는 예제
- 스레드 상태가 sleep이 아닐 경우 인터럽트가 발생해도 인터럽트 상태를 확인하지 않으면 스레드 상태가 바뀌지 않으므로 다음 sleep 상태에서 InterruptedException이 발생

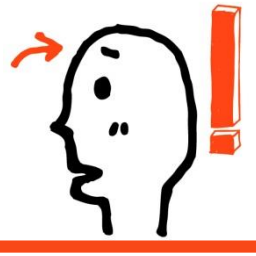
스레드 종료하기



데몬 스레드

- **데몬(Daemon)** : 리눅스 서버에서 주로 많이 사용
  백그라운드 상태에서 대기하고 있다가 처리할 요청이 발생하거나
  조건 상황이 맞으면 작업을 실행하는 프로그램을 의미
- 자바에서는 Thread 클래스에서 제공하는 setDaemon( ) 메소드를 사용하면 이와 비슷한 동작
- setDaemon( ) 메소드를 이용하면 해당 스레드를 데몬으로 설정할 수 있고
  이렇게 데몬으로 설정된 스레드는 모든 스레드가 종료해야 JVM이 종료된다는 조건에서 제외
- 데몬 스레드는 주로 프로그램이 실행되는 동안 백그라운드에서 동작하는 서비스 제공 위해 사용

스레드 종료하기

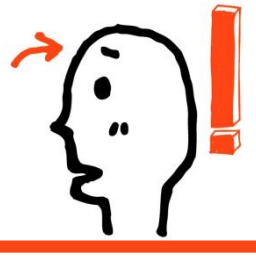


데몬 스레드

코드 9-18 package com.gilbut.chapter9;

```
1   public class DaemonThread implements Runnable
2   {
3      public static void main(String[] args)
4      {
5         Thread th = new Thread(new DaemonThread());
6         th.setDaemon(true);
7         th.start();
8
9         try
10        {
11           Thread.sleep(1000);
12        }
13        catch (InterruptedException e)
14        {
15           e.printStackTrace();
16        }
17
18        System.out.println("Main thread terminated...");
19     }
```

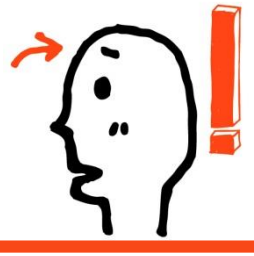
스레드 종료하기



데몬 스레드

```
20
21      @Override
22      public void run()
23      {
24          while (true)
25          {
26              System.out.println("is running...");
27
28              try
29              {
30                  Thread.sleep(500);
31              }
32              catch (InterruptedException e)
33              {
34                  e.printStackTrace();
35                  break;
36              }
37          }
38      }
39  }
```

스레드 종료하기



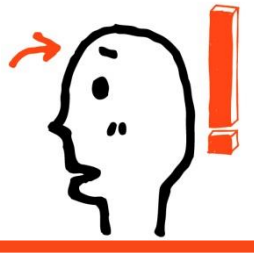
데몬 스레드

실행결과

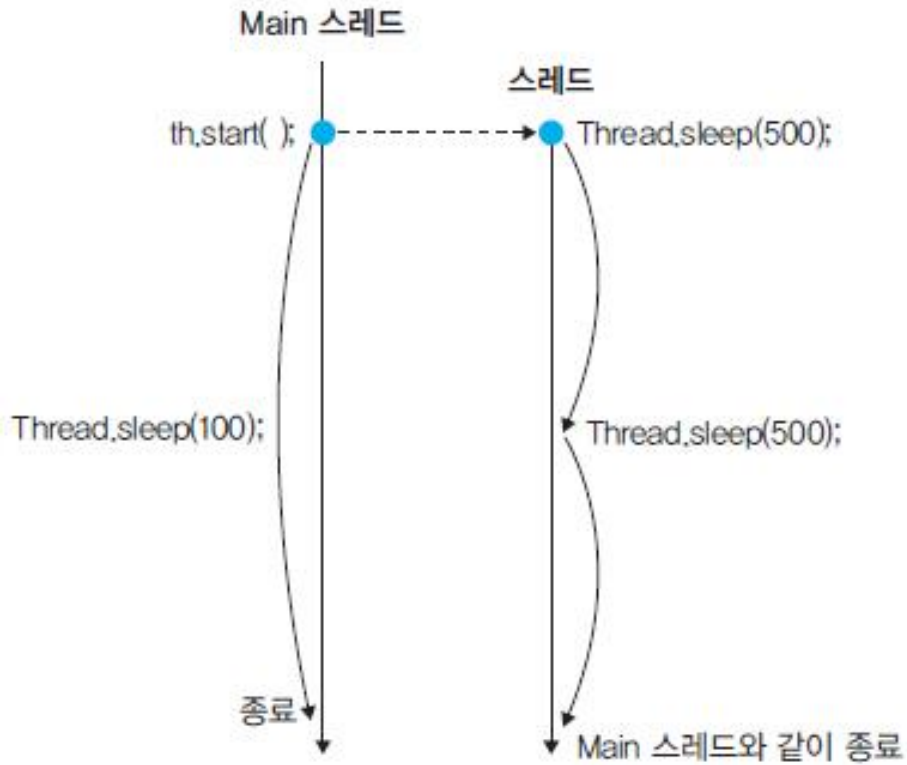
```
is running...
is running...
Main thread terminated...
```

- 데몬 스레드로 생성한 경우를 보여주는 예제
- 마지막 스레드인 main 스레드가 종료될 때 함께 종료되는 것

스레드 종료하기

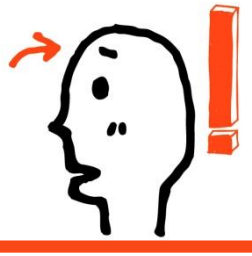


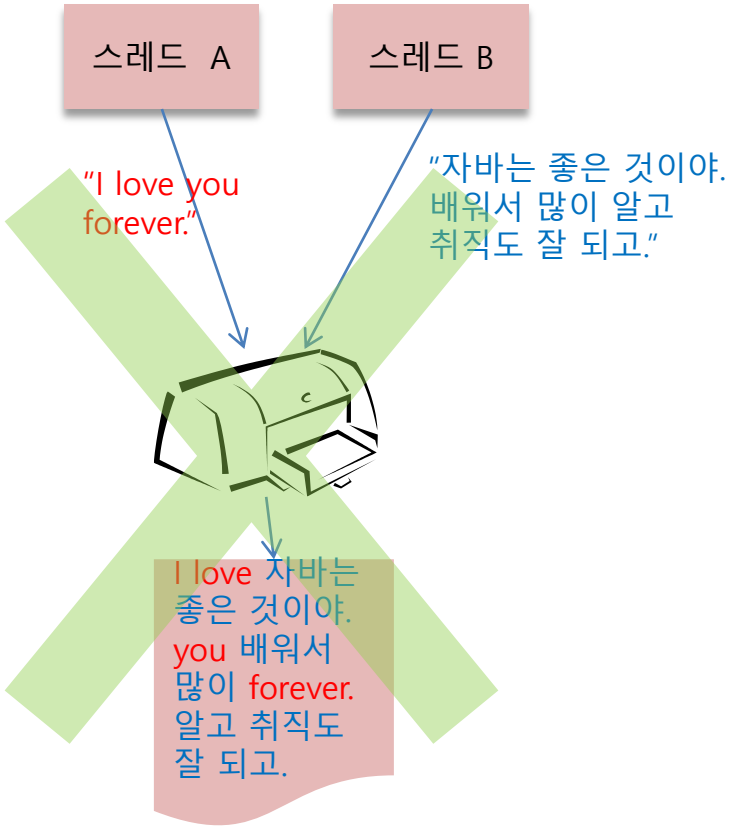
데몬 스레드



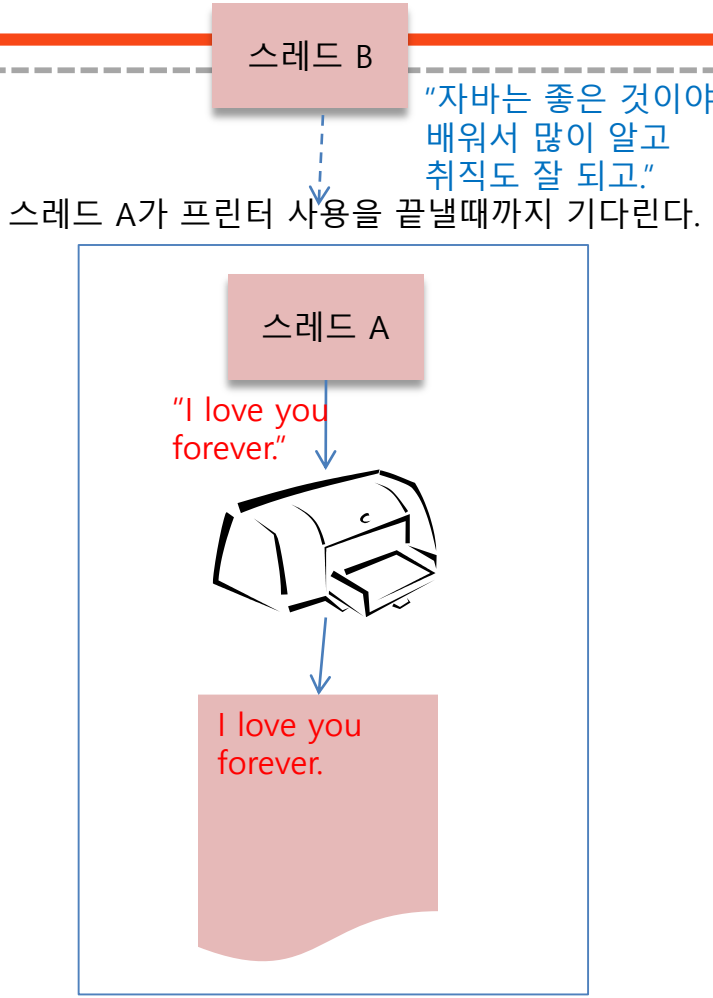
△ 그림 9-7 DaemonThread 클래스의 실행 흐름

두 스레드의 프린터 동시 쓰기로 충돌하는 경우



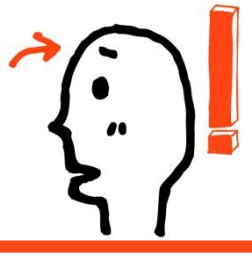


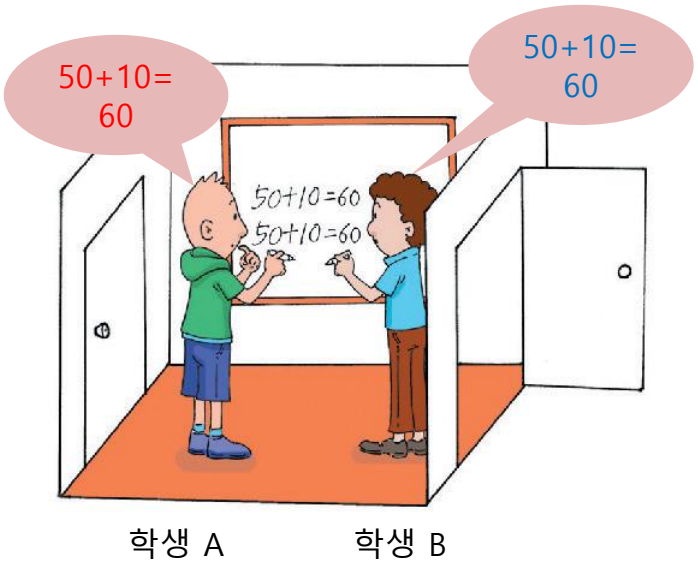
두 스레드가 동시에 프린터에 쓰는 경우
문제 발생



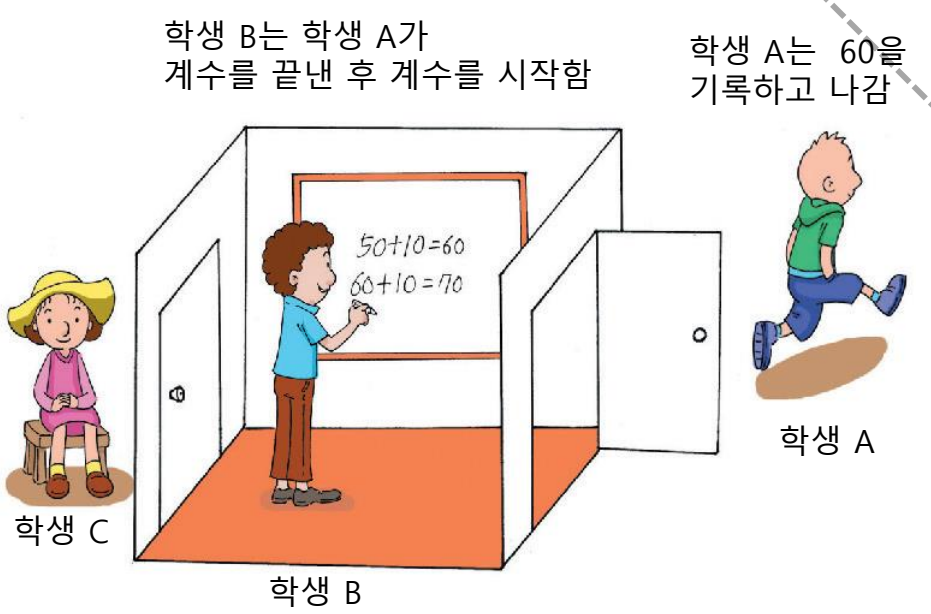
한 스레드의 출력이 끝날 때까지 대기함으로써 **정상 출력**

공유 집계판에 동시 접근하는 경우



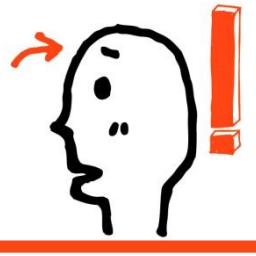


두 학생이 동시에 방에 들어와서
집계판을 수정하는 경우
집계판의 **결과가 잘못됨**



방에 먼저 들어간 학생이
집계를 끝내기를 기다리면
정상 처리

쓰레드간 메모리 영역의 공유 예제



ThreadHeapMultiAccess.java

```
class Sum
{
    int num;
    public Sum() { num=0; }
    public void addNum(int n) { num+=n; }
    public int getNum() { return num; }
}

class AdderThread extends Thread
{
    Sum sumInst;
    int start, end;

    public AdderThread(Sum sum, int s, int e)
    {
        sumInst=sum;
        start=s;
        end=e;
    }
    public void run()
    {
        for(int i=start; i<=end; i++)
            sumInst.addNum(i);
    }
}
```

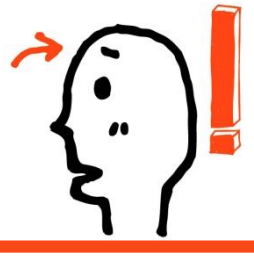
실행 결과

```
1~100까지의 합 : 5050
```

```
public static void main(String[] args)
{
    Sum s=new Sum();
    AdderThread at1=new AdderThread(s, 1, 50);
    AdderThread at2=new AdderThread(s, 51, 100);
    at1.start();
    at2.start();
    try
    {
        at1.join();
        at2.join();
    }
    catch(InterruptedException e)
    {
        e.printStackTrace();
    }
    System.out.println("1~100까지의 합 : "+s.getNum());
}
```

위의 예제는 둘 이상의 쓰레드가 메모리 공간에 동시 접근하는 문제를 가지고 있다. 따라서 정상적이지 못한 실행의 결과가 나올 수도 있다.

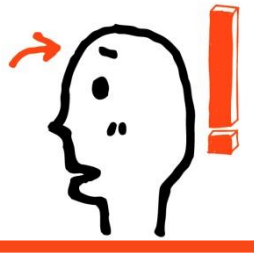
멀티 스레딩의 핵심, 스레드 동기화하기



공유 자원을 동시에 사용할 경우 발생하는 문제

- **스레드 동기화(Thread Synchronization)**
- 정확한 데이터를 처리하기 위해서 매우 중요한 개념
- 멀티 스레딩과 바로 직결되는 문제
- 멀티 스레딩의 가장 큰 문제점 : 공유 자원을 보호하기 힘들다는 것
- 스레드가 동시에 공유 자원에 접근하는 경우, 의도했던 것과 다른 결과가 발생할 수 있음

멀티 스레딩의 핵심, 스레드 동기화하기



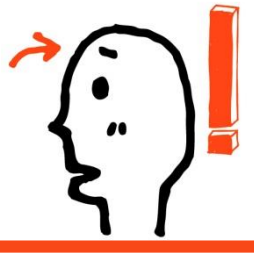
공유 자원을 동시에 사용할 경우 발생하는 문제

코드 9-22 package com.gilbut.chapter9;

```
1   public class ConcurrentSample1 implements Runnable
2   {
3       private int res = 0;
4
5       public static void main(String[] args)
6       {
7           ConcurrentSample1 concurrent = new ConcurrentSample1();
8           Thread th1 = new Thread(concurrent);
9           Thread th2 = new Thread(concurrent);
10
11          th1.start();
```

하나의 concurrent 객체를 사용하여 두 개의 스레드를 실행하는 구문

멀티 스레딩의 핵심, 스레드 동기화하기

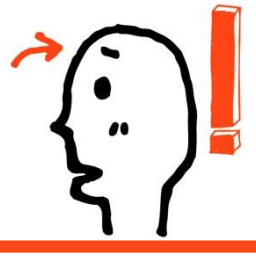


공유 자원을 동시에 사용할 경우 발생하는 문제

```
12          th2.start();
13
14          try
15          {
16              th1.join();
17              th2.join();
18          }
19          catch (InterruptedException e)
20          {
21              e.printStackTrace();
22          }
23
24          System.out.println(concurrent.res);
25      }
26
```

두 개의 스레드가 실행되므로 res의 값은 20,000이 되지 않을까? 하지만 여기서 공유 자원에 대한 문제가 발생합니다.

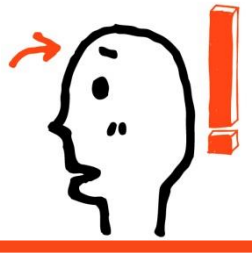
멀티 스레딩의 핵심, 스레드 동기화하기



공유 자원을 동시에 사용할 경우 발생하는 문제

```
27      @Override
28      public void run()
29      {
30          sum();
31      }
32
33      private void sum()
34      {
35          for (int i = 0; i < 10000; i++)
36          {
37              res++;
38          }
39      }
40  }
```

멀티 스레딩의 핵심, 스레드 동기화하기



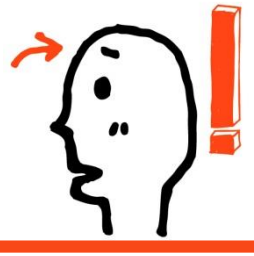
공유 자원을 동시에 사용할 경우 발생하는 문제

실행결과

```
12320
```

- 여러 스레드가 공유 자원을 동시에 사용할 경우 나타날 수 있는 문제점을 보여줌
- 스레드 th1과 th2가 동시에 res 클래스 변수에 접근함으로써 발생 하는 현상

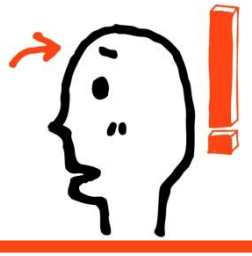
멀티 스레딩의 핵심, 스레드 동기화하기



해결책 : 동기화

- **상호 배타적 처리** : 스레드가 공유 자원에 대한 사용이 끝날 때까지 다른 스레드가 접근하지 못하도록 제한
- **동기화** : 자원을 상호 베타적으로 처리하는 방법
  - 동기화는 오직 하나의 스레드만이 공유 자원에 대한 접근을 허락
  - 동기화된 블록이나 메소드는 먼저 접근한 스레드가 작업을 완료할 때까지 다른 스레드가 진입할 수 없음

멀티 스레딩의 핵심, 스레드 동기화하기



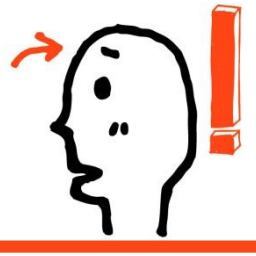
해결책 : 동기화

▷ 자바 동기화에 필요한 것은 **synchronized 구문**과 **스레드 락(Lock)**

▷ **자바에서 동기화하는 방법**

```
사용법 : //메소드에 키워드를 선언하는 방법 : 동기화 메소드
        [제어자] synchronized [반환값] [ 메소드 이름 ]( ) { //공유 자원 변경 }

        //메소드 내부에 키워드를 선언하는 방법 : 동기화 블록
        [메소드 선언부]
        {
          synchronized ( [락 객체] ) {
              //공유 자원 변경
          }
        }
```

멀티 스레딩의 핵심, 스레드 동기화하기



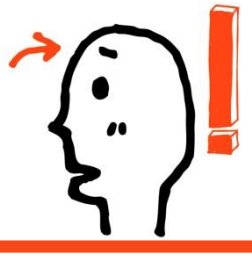
해결책 : 동기화

사용예 :

```
//동기화 메소드의 예제
public synchronized void reportCurrentStats() { //구문 }
//동기화 블록의 예제
public void reportCurrentStatus()
{
    synchronized(this) {
        //구문
    }
}
```

- 메소드에 synchronized 키워드를 선언한 경우에는 메소드 전체가 동기화되는 것
- 블록에 synchronized 키워드를 사용한 경우에는 블록 내부 내용만 동기화

멀티 스레딩의 핵심, 스레드 동기화하기



해결책 : 동기화

- 락 객체는 문지기 역할을 해서 오직 하나의 스레드만이 동기화 블록에 접근 가능

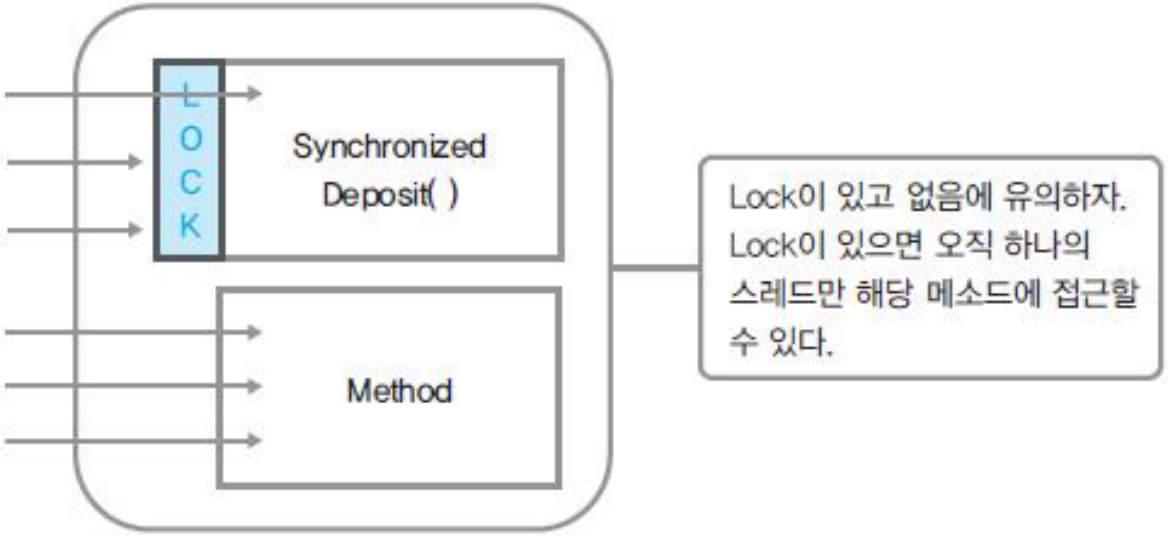
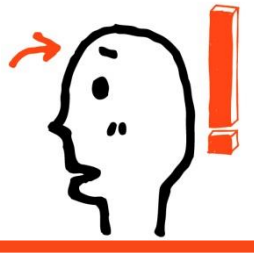
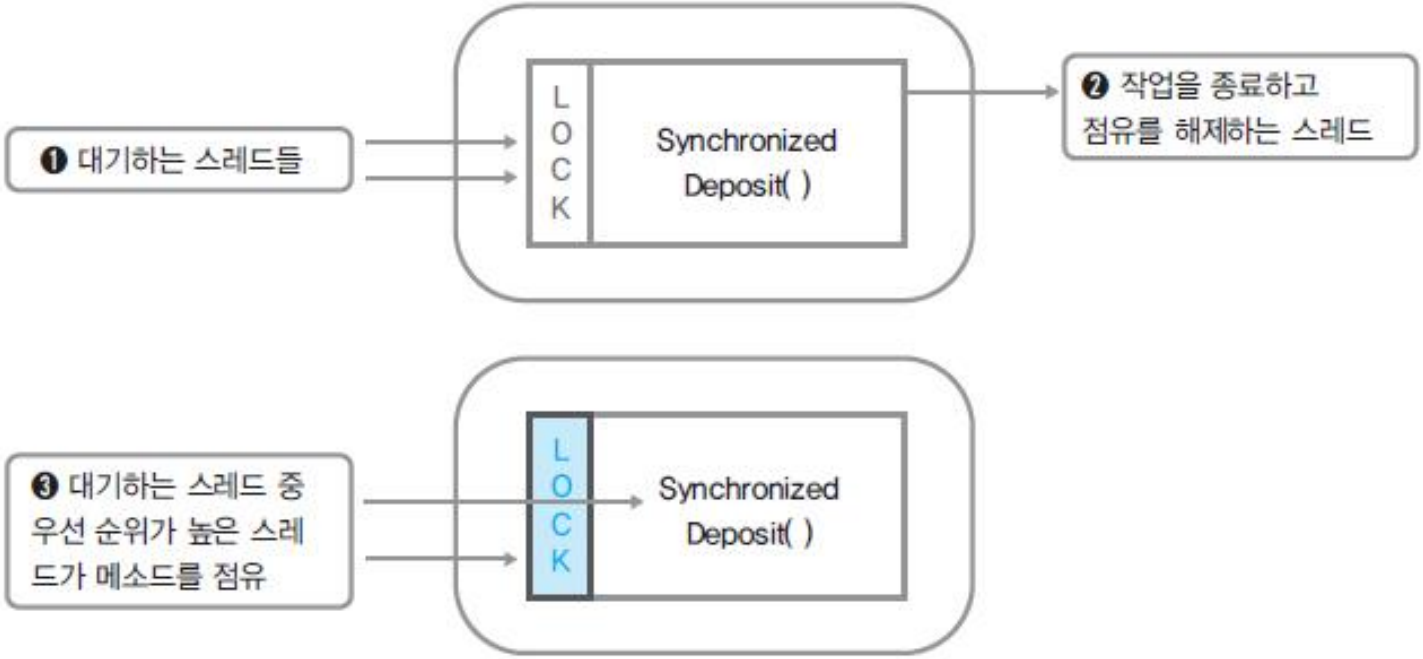


그림 9-9 일반 메소드와 동기화 메소드의 차이점

멀티 스레딩의 핵심, 스레드 동기화하기

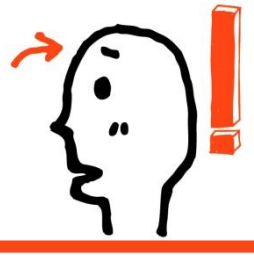


해결책 : 동기화



△ 그림 9-10 동기화의 상호 배타적 처리 과정

멀티 스레딩의 핵심, 스레드 동기화하기



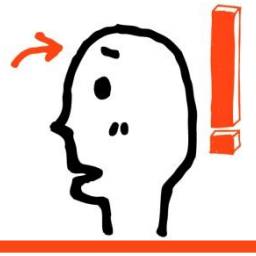
해결책 : 동기화

- 보통 동기화 블록에서 락을 걸 때는 락을 걸기 위한 전용 객체를 따로 생성하며, 메모리의 부담을 최소화하기 위해서 Object 클래스 객체를 사용

- 일반적으로 락 객체는 private 키워드로 선언해 다른 객체에 의해 변경되지 않도록 함

```
public class ThreadLock(){
    private Object lockObj = new Object();
    public void testLock(String name) {
        synchronized(lockObj){
            //처리해야 할 일
        }
    }
```

멀티 스레딩의 핵심, 스레드 동기화하기

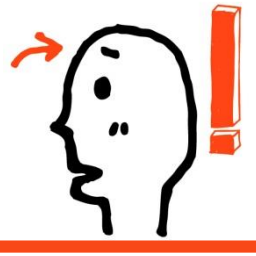


해결책 : 동기화

코드 9-23 package com.gilbut.chapter9;

```
1   public class ConcurrentSample2 implements Runnable
2   {
3       private int res = 0;
4
5       public static void main(String[] args)
6       {
7           ConcurrentSample2 concurrent = new ConcurrentSample2();
8           Thread th1 = new Thread(concurrent);
9           Thread th2 = new Thread(concurrent);
10
11          th1.start();
12          th2.start();
13
```

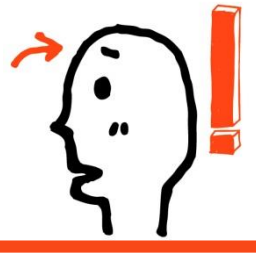
멀티 스레딩의 핵심, 스레드 동기화하기



해결책 : 동기화

```
14          try
15          {
16              th1.join();
17              th2.join();
18          }
19          catch (InterruptedException e)
20          {
21              e.printStackTrace();
22          }
23
24          System.out.println(concurrent.res);
25      }
26
```

멀티 스레딩의 핵심, 스레드 동기화하기

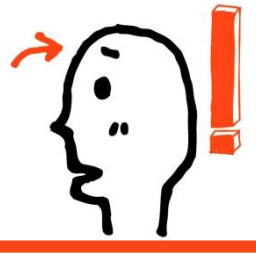


해결책 : 동기화

```
27      @Override
28      public void run()
29      {
30          sum();
31      }
32
33      private synchronized void sum()
34      {
35          for (int i = 0; i < 10000; i++)
36          {
37              res++;
38          }
39      }
40  }
```

코드 9-22와 차이점입니다. 이번 예제에서 가장 중요한 역할을 하는 키워드임을 기억하세요!

멀티 스레딩의 핵심, 스레드 동기화하기



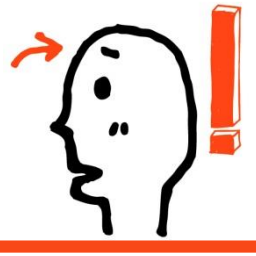
해결책 : 동기화

실행결과

```
20000
```

- 메소드를 동기화해서 여러 스레드가 동시에 sum( ) 메소드에 접근할 수 없도록 함

멀티 스레딩의 핵심, 스레드 동기화하기



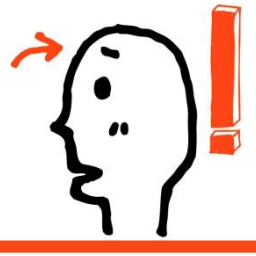
해결책 : 동기화

코드 9-24 package com.gilbut.chapter9;

```
1   private void sum()
2   {
3       synchronized (this)
4       {
5           for (int i = 0; i < 10000; i++)
6           {
7               res++;
8           }
9       }
10  }
```

- sum( ) 메소 드에 먼저 접근한 스레드가 res++ 연산을 10,000번 실행한 후 종료되면 대기 중인 다음 스레드가 sum( ) 메소드에 순차적으로 접근하는 방식

쓰레드의 동기화 기법1
: synchronized 기반 동기화 메소드



```
class Increment
{
    int num=0;
    public synchronized void increment(){ num++; }
    public int getNum() { return num; }
}

class IncThread extends Thread
{
    Increment inc;

    public IncThread(Increment inc)
    {
        this.inc=inc;
    }
    public void run()
    {
        for(int i=0; i<10000; i++)
            for(int j=0; j<10000; j++)
                inc.increment();
    }
}
```

```
public synchronized void increment( )
{
    num++;
}
```

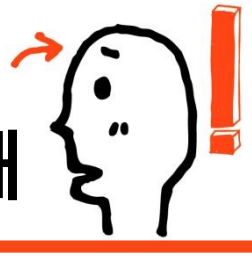
동기화 메소드의 선언!

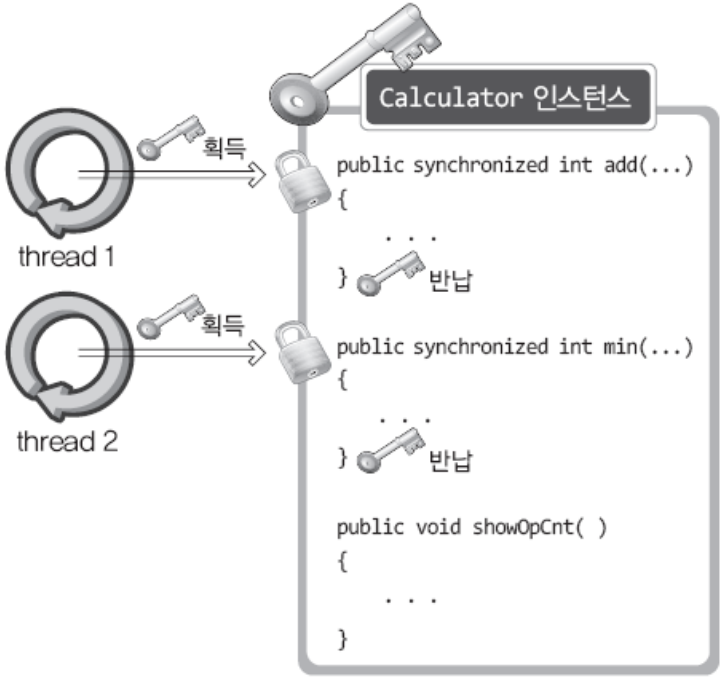
synchronized 선언으로 인해서 increment 메소드는 쓰레드에 안전한 함수가 된다.

synchronized 선언으로 인해서 increment 메소드는 정상적으로 동작한다.

그러나 엄청난 성능의 감소를 동반한다! 특히 위 예제와 같이 빈번함 메소드의 호출은 문제가 될 수 있다.

synchronized 기반 동기화 메소드의 정확한 이해

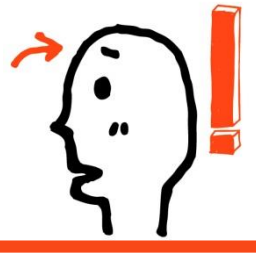




동기화에 사용되는 인스턴스는 하나이며, 이 인스턴스에는 **하나의 열쇠만이 존재**한다.

동기화의 대상은 인스턴스이며, 인스턴스의 열쇠를 획득하는 순간 **모든 동기화 메소드에는 타 쓰레드의 접근이 불가능**하다. 따라서 메소드 내에서 동기화가 필요한 영역이 매우 제한적이라면 메소드 전부를 synchronized로 선언하는 것은 적절치 않다.

쓰레드의 동기화 기법2
: synchronized 기반 동기화 블록



동기화 메소드 기반

```
public synchronized int add(int n1, int n2)
{
    opCnt++;     // 동기화가 필요한 문장
    return n1+n2;
}

public synchronized int min(int n1, int n2)
{
    opCnt++;     // 동기화가 필요한 문장
    return n1-n2;
}
```

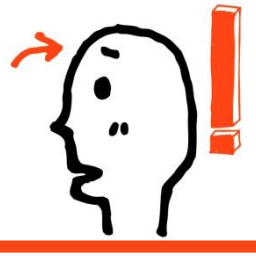
동기화 블록 기반

```
public int add(int n1, int n2)
{
    synchronized(this)
    {
        opCnt++;     // 동기화 된 문장
    }
    return n1+n2;
}

public int min(int n1, int n2)
{
    synchronized(this)
    {
        opCnt++;     // 동기화 된 문장
    }
    return n1-n2;
}
```

동기화 블록을 이용하면 동기화의 대상이 되는 영역을 세밀하게 제한할 수 있다.

synchronized(this)에서 this는 동기화의 대상을 알리는 용도로 사용이 되었다. 즉, 메소드가 호출된 인스턴스 자신의 열쇠를 대상으로 동기화를 진행하는 문장이다.

동기화 블록의 예



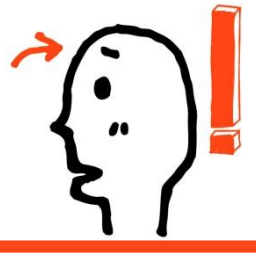
```
public synchronized void addOneNum1()
{
    synchronized(key1)
    {
        num1+=1;
    }
}
public synchronized void addTwoNum1()
{
    synchronized(key1)
    {
        num1+=2;
    }
}
public synchronized void addOneNum2()
{
    synchronized(key2)
    {
        num2+=1;
    }
}
public synchronized void addTwoNum2()
{
    synchronized(key2)
    {
        num2+=2;
    }
}
. . . . .
Object key1=new Object();
Object key2=new Object();
```

```
class IHaveTwoNum
{
    . . . .
    public void addOneNum1()
    {
        synchronized(this) { num1+=1; }
    }
    public void addTwoNum1()
    {
        synchronized(this) { num1+=2; }
    }
    public void addOneNum2()
    {
        synchronized(key) { num2+=1; }
    }
    public void addTwoNum2()
    {
        synchronized(key) { num2+=2; }
    }
    . . . .
    Object key=new Object();
}
```

보다 일반적인 형태, 두 개의 동기화 인스턴스 중 하나는 this로 지정!

왼쪽의 코드에서 보이듯이 동기화 블록을 이용하면 동기화의 기준을 다양화할 수 있다.

쓰레드 접근순서의 동기화 필요성



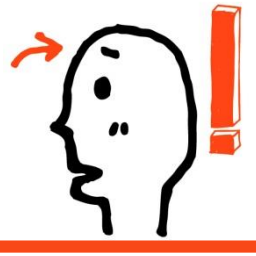
```
class NewsWriter extends Thread
{
    NewsPaper paper;
    public NewsWriter(NewsPaper paper)
    {
        this.paper=paper;
    }
    public void run()
    {
        paper.setTodayNews("자바의 열기가 뜨겁습니다.");
    }
}

class NewsReader extends Thread
{
    NewsPaper paper;
    public NewsReader(NewsPaper paper)
    {
        this.paper=paper;
    }
    public void run()
    {
        System.out.println("오늘의 뉴스 : "+paper.getTodayNews());
    }
}
```

```
class NewsPaper
{
    String todayNews;
    public void setTodayNews(String news)
    {
        todayNews=news;
    }
    public String getTodayNews()
    {
        return todayNews;
    }
}
```

```
public static void main(String[] args)
{
    NewsPaper paper=new NewsPaper();
    NewsReader reader=new NewsReader(paper);
    NewsWriter writer=new NewsWriter(paper);
    reader.start();
    writer.start();
    try
    {
        reader.join();
        writer.join();
    }
    catch(InterruptedException e)
    {
        e.printStackTrace();
    }
}
```

본 예제가 논리적으로 실행되려면 NewsWriter 쓰레드가 먼저 실행되고, 이어서 NewReader 쓰레드가 실행되어야 한다. 하지만 이를 보장하지 못하는 구조로 구현이 되어 있다.

wait, notify, notifyall에 의한 실행순서 동기화



· public final void wait() throws InterruptedException

위의 함수를 호출한 쓰레드는 notify 또는 notifyAll 메소드가 호출될 때까지 블로킹 상태에 놓이게 된다.

· public final void notify()

wait 함수의 호출을 통해서 블로킹 상태에 놓여있는 쓰레드 하나를 깨운다.

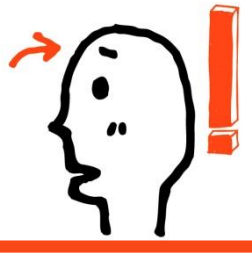
· public final void notifyAll()

wait 함수의 호출을 통해서 블로킹 상태에 놓여있는 모든 쓰레드를 깨운다.

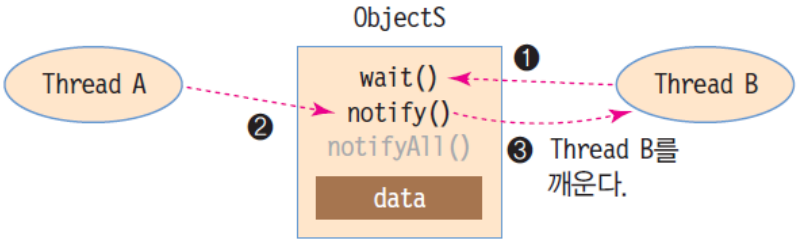
```
synchronized(this)
{
    wait();
}
```

위의 함수들은 왼쪽에서 보이는 바와 같이 한 순간에 하나의 쓰레드만 호출할 수 있도록 동기화 처리를 해야 한다.

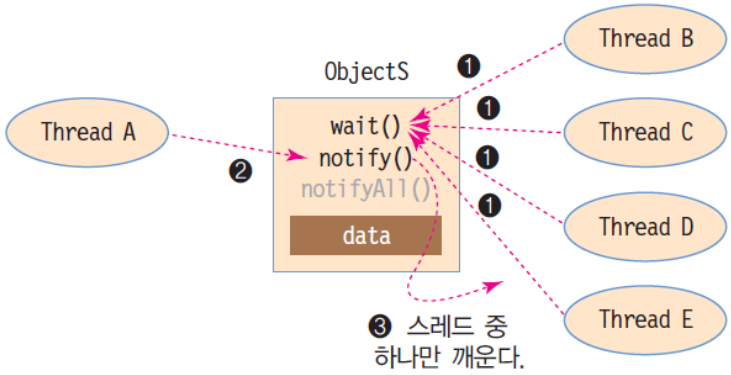
wait, notify, notifyall에 의한 실행순서 동기화



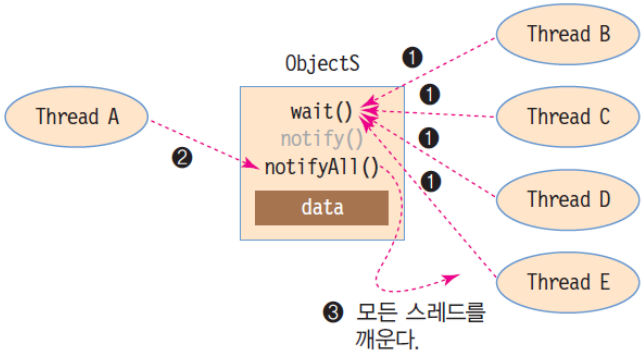
Thread B가 ObjectS.wait()를 호출하여 무한 대기하고, Thread A가 ObjectS.notify()를 호출하여 ObjectS에 대기하고 있는 Thread B를 깨운다.



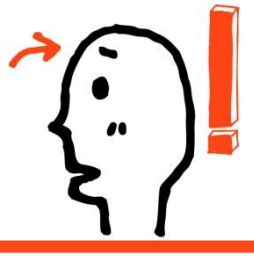
4 개의 스레드가 모두 ObjectS.wait()를 호출하여 대기하고, ThreadA는 ObjectS.notify()를 호출하여 대기 중인 스레드 중 하나만 깨우는 경우



4 개의 스레드가 모두 ObjectS.wait()를 호출하여 대기하고, ThreadA는 ObjectS.notifyAll()를 호출하여 대기중인 4개의 스레드를 모두 깨우는 경우



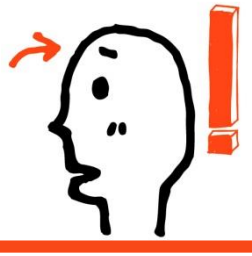
실행순서 동기화 예제



```
class NewsPaper
{
    String todayNews;
    boolean isTodayNews=false;
    public void setTodayNews(String news)
    {
        todayNews=news;
        isTodayNews=true;
        synchronized(this)
        {
            notifyAll();    // 모두 일어나세요!
        }
    }
    public String getTodayNews()
    {
        if(isTodayNews==false)
        {
            try
            {
                synchronized(this)
                {
                    wait();     // 한숨 자면서 기다리겠습니다.
                }
            }
            catch(InterruptedException e)
            {
                e.printStackTrace();
            }
        }

        return todayNews;
    }
}
```

wait과 notifyAll 메소드에 의한 동기화가 진행될 때, 이전 예제에서 달라지는 부분은 쓰레드 클래스가 아닌 쓰레드에 의해 접근이 이뤄지는 NewsPaper 클래스라는 사실에 주목하기 바란다.

synchronized 키워드의 대체



```
class MyClass
{
    private final ReentrantLock criticObj=new ReentrantLock();
    . . . .
    void myMethod(int arg)
    {
        criticObj.lock();    // 다른 쓰레드가 진입하지 못하게 문을 잠근다.
        . . . .
        . . . .
        criticObj.unlock(); // 다른 쓰레드의 진입이 가능하게 문을 연다.
    }
}
```

보다 안정적인 구현모델,

반드시 unlock 메소드가 호출되는 모델

```
void myMethod(int arg)
{
    criticObj.lock();    // 다른 쓰레드가 진입하지 못하게 문을 잠근다.
    try
    {
        . . . .
        . . . .
    }
    finally
    {
        criticObj.unlock(); // 다른 쓰레드의 진입이 가능하게 문을 연다.
    }
}
```

ReentrantLock 인스턴스를 이용한 동기화 기법

Java Ver 5.0 이후로 제공된 동기화 방식이다. lock 메소드와 unlock 메소드의 호출을 통해서 동기화 블록을 구성한다.

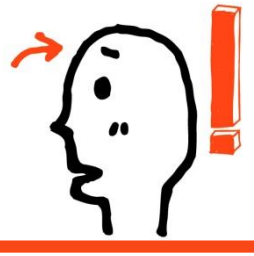
await, signal, signalAll에 의한 실행순서의 동기화

• await	낮잠을 취한다(wait 메소드에 대응)
• signal	낮잠 자는 쓰레드 하나를 깨운다(notify 메소드에 대응).
• signalAll	낮잠 자는 모든 쓰레드를 깨운다(notifyAll 메소드에 대응).

ReentrantLock 인스턴스 대상으로 newCondition 메소드 호출 시, Condition 인터페이스를 구현하는 인스턴스의 참조 값 반환!
이 인스턴스를 대상으로 위의 메소드를 호출하여, 쓰레드의 실행순서를 동기화 한다.

위의 메소드의 사용방법을 보이는 예제 ConditionSyncStringReadWrite.java는 코드 양이 많은 관계로 해당 파일을 열어서 참고하기 바란다. 참고로, 앞서 보인 wait, notify 메소드의 호출을 통한 동기화 방법과 크게 다르지 않다.

람다식을 이용한 쓰레드 생성 방법



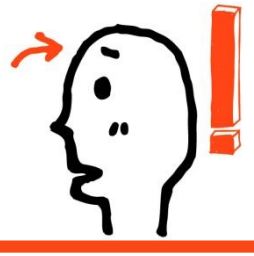
```
public static void main(String[] args) {
    Runnable task = () -> { // 쓰레드가 실행하게 할 내용
        int n1 = 10;
        int n2 = 20;
        String name = Thread.currentThread().getName();
        System.out.println(name + ": " + (n1 + n2));
    };

    Thread t = new Thread(task);
    t.start(); // 쓰레드 생성 및 실행
    System.out.println("End " + Thread.currentThread().getName());
}
// Runnable          void run()
```

```
명령 프롬프트
C:₩JavaStudy>java MakeThreadDemo
End main
Thread-0: 30

C:₩JavaStudy>
```

모든 쓰레드가 일을 마쳐야 프로그램 종료

- 1단계 Runnable을 구현한 인스턴스 생성
- 2단계 Thread 인스턴스 생성
- 3단계 start 메소드 호출

람다식을 이용한 쓰레드 생성의 예



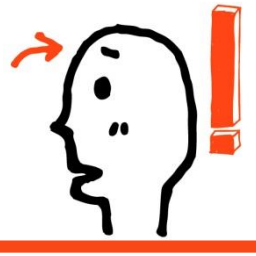
```
public static void main(String[] args) {
    Runnable task = () -> { // 쓰레드가 실행하게 할 내용
        int n1 = 10;
        int n2 = 20;
        String name = Thread.currentThread().getName();
        System.out.println(name + ": " + (n1 + n2));
    };

    Thread t = new Thread(task);
    t.start(); // 쓰레드 생성 및 실행
    System.out.println("End " + Thread.currentThread().getName());
}
// Runnable          void run()
```

```
명령 프롬프트
C:₩JavaStudy>java MakeThreadDemo
End main
Thread-0: 30

C:₩JavaStudy>
```

모든 쓰레드가 일을 마쳐야 프로그램 종료

- 1단계 Runnable을 구현한 인스턴스 생성
- 2단계 Thread 인스턴스 생성
- 3단계 start 메소드 호출

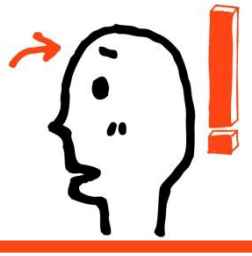
람다식을 이용한 쓰레드 생성의 예

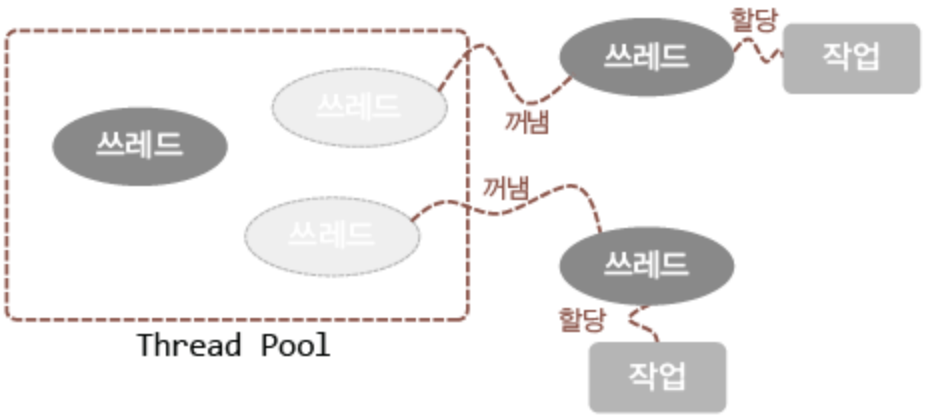


```
public static void main(String[] args) {
   Runnable task1 = () -> { // 20 미만 짝수 출력
      try {
         for(int i = 0; i < 20; i++) {
            if(i % 2 == 0)
               System.out.print(i + " ");
            Thread.sleep(100); // 0.1초간 잠을 잔다.
         }
      }catch(InterruptedException e) {
         e.printStackTrace();
      }
   };
   Runnable task2 = () -> { // 20 미만 홀수 출력
      try {
         for(int i = 0; i < 20; i++) {
            if(i % 2 == 1)
               System.out.print(i + " ");
            Thread.sleep(100); // 0.1초간 잠을 잔다.
         }
      }catch(InterruptedException e) {
         e.printStackTrace();
      }
   };
   Thread t1 = new Thread(task1);
      Thread t2 = new Thread(task2);
      t1.start();
      t2.start();
   }
```

```
명령 프롬프트
C:₩JavaStudy>java MakeThreadMultiDemo
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
C:₩JavaStudy>
```

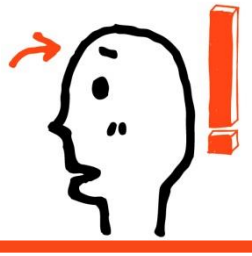
쓰레드 풀 모델





쓰레드의 생성과 소멸은 리소스 소모가 많은 작업이다.
그런데 쓰레드 풀은 쓰레드의 재활용을 위한 모델이다.

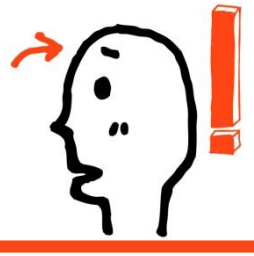
쓰레드 풀 기반의 예제1



```
class ExecutorsDemo {
   public static void main(String[] args) {
      Runnable task = () -> {    // 쓰레드에게 시킬 작업
         int n1 = 10;
         int n2 = 20;
         String name = Thread.currentThread().getName();
         System.out.println(name + ": " + (n1 + n2));
      };

      ExecutorService exr = Executors.newSingleThreadExecutor();
      exr.submit(task);    // 쓰레드 풀에 작업을 전달

      System.out.println("End " + Thread.currentThread().getName());
      exr.shutdown();    // 쓰레드 풀과 그 안에 있는 쓰레드의 소멸
   }
}
```

```
명령 프롬프트
C:₩JavaStudy>java ExecutorsDemo
End main
pool-1-thread-1: 30

C:₩JavaStudy>
```

쓰레드 풀의 유형



- `newSingleThreadExecutor`

  풀 안에 하나의 쓰레드만 생성하고 유지한다.

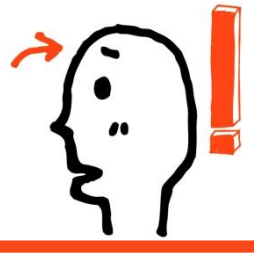
- `newFixedThreadPool`

  풀 안에 인자로 전달된 수의 쓰레드를 생성하고 유지한다.

- `newCachedThreadPool`

  풀 안의 쓰레드의 수를 작업의 수에 맞게 유동적으로 관리한다.

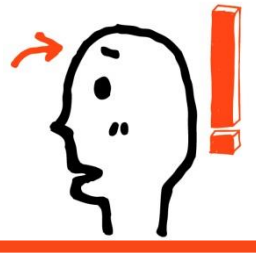
쓰레드 풀 기반의 예제2



```
public static void main(String[] args) {
   Runnable task1 = () -> {
      String name = Thread.currentThread().getName();
      System.out.println(name + ": " + (5 + 7));
   };
   Runnable task2 = () -> {
      String name = Thread.currentThread().getName();
      System.out.println(name + ": " + (7 - 5));
   };

   ExecutorService exr = Executors.newFixedThreadPool(2);
   exr.submit(task1);
   exr.submit(task2);
   exr.submit(() -> {
      String name = Thread.currentThread().getName();
      System.out.println(name + ": " + (5 * 7));
   });

   exr.shutdown();
}
```

```
명령 프롬프트
C:₩JavaStudy>java ExecutorsDemo2
pool-1-thread-1: 12
pool-1-thread-2: 2
pool-1-thread-1: 35

C:₩JavaStudy>
```

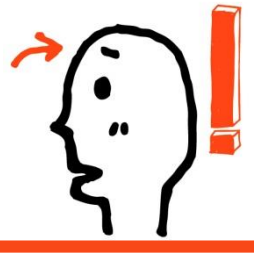
Callable & Future



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

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

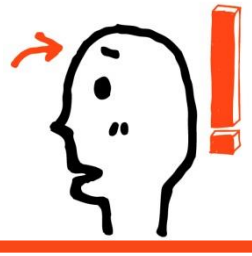
Callable & Future 관련 예



```
public static void main(String[] args) throws InterruptedException, ExecutionException {
   Callable<Integer> task = () -> {
      int sum = 0;
      for(int i = 0; i < 10; i++)
         sum += i;
      return sum;
   };

   ExecutorService exr = Executors.newSingleThreadExecutor();
   Future<Integer> fur = exr.submit(task);

   Integer r = fur.get(); // 쓰레드의 반환 값 획득
   System.out.println("result: " + r);
   exr.shutdown();
}
```

```
명령 프롬프트
C:₩JavaStudy>java CallableDemo
result: 45

C:₩JavaStudy>
```

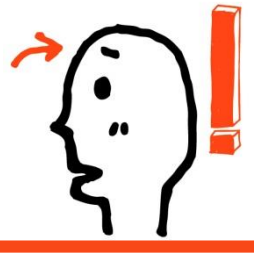
synchronized를 대신하는 ReentrantLock



```
class MyClass {
   ReentrantLock criticObj = new ReentrantLock();
   void myMethod(int arg) {
      criticObj.lock() ;    // 문을 잠근다.
      .... // 한 쓰레드에 의해서만 실행되는 영역
      criticObj.unlock();    // 문을 연다.
   }
}
```

```
class MyClass {
   ReentrantLock criticObj = new ReentrantLock();
   void myMethod(int arg) {
      criticObj.lock() ;    // 문을 잠근다.
      try {
         ....    // 한 쓰레드에 의해서만 실행되는 영역
      } finally {
         criticObj.unlock();    // 문을 연다.
      }
   }
}
```

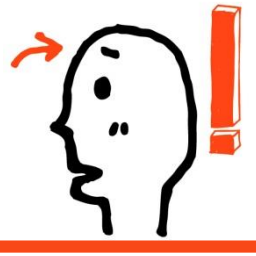
컬렉션 인스턴스 동기화



```
public static <T> Set<T> synchronizedSet(Set<T> s)

public static <T> List<T> synchronizedList(List<T> list)

public static <K, V> Map<K, V> synchronizedMap(Map<K, V> m)

public static <T> Collection<T> synchronizedCollection(Collection<T> c)
```

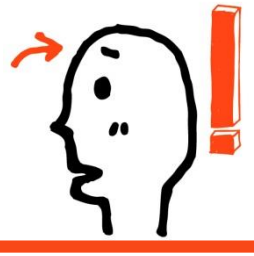
```
List<String> lst = Collections.synchronizedList(new ArrayList<String>());
```

컬렉션 인스턴스 동기화의 예 1



```
Runnable task = () -> {
   synchronized(lst) {
      ListIterator<Integer> itr = lst.listIterator();
      while(itr.hasNext())
         itr.set(itr.next() + 1);
   }
};
```

컬렉션 인스턴스 동기화의 예2



```
class SyncArrayList {
   public static List<Integer> lst =
      Collections.synchronizedList(new ArrayList<Integer>());

   public static void main(String[] args) throws InterruptedException {
      for(int i = 0; i < 16; i++)
         lst.add(i);
      System.out.println(lst);

      Runnable task = () -> {
         ListIterator<Integer> itr = lst.listIterator();
         while(itr.hasNext())
            itr.set(itr.next() + 1);
      };
```

```
명령 프롬프트
C:₩JavaStudy>java SyncArrayList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
[1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]

C:₩JavaStudy>
```

```
ExecutorService exr =
Executors.newFixedThreadPool(3);
      exr.submit(task);
      exr.submit(task);
      exr.submit(task);

      exr.shutdown();
      exr.awaitTermination(100,
TimeUnit.SECONDS);
      System.out.println(lst);
   }
}
```

쓰레드를 생성하는 더 좋은 방법

THANK YOU

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술