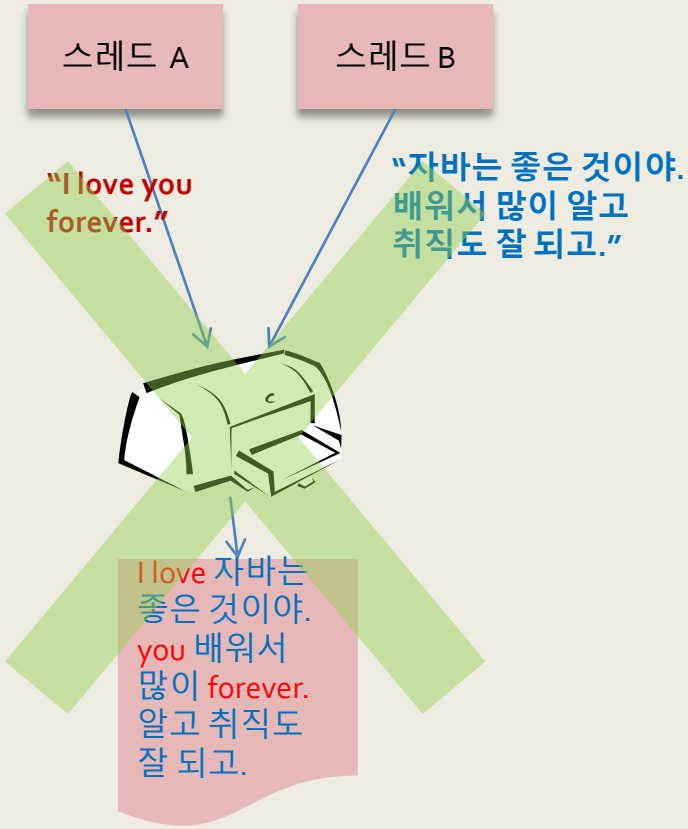
제 14 장

스레드와 멀티태스킹
Part-2 : 스레드 동기화

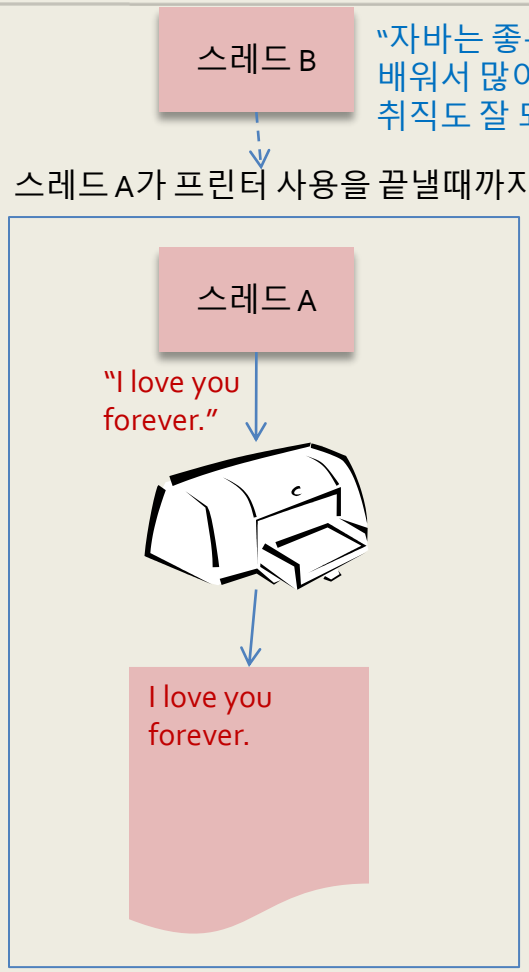
스레드 동기화 (Thread Synchronization)

- 멀티스레드 프로그램 작성시 주의점
  - 다수의 스레드가 **공유 데이터**(shared resource)에 동시에 접근하는 경우
    - 공유 데이터의 값에 예상치 못한 결과 발생 가능
- 스레드 동기화
  - 멀티스레드의 공유 데이터의 동시 접근 문제 해결책
    - 공유데이타를 접근하고하 하는 모든 스레드의 한 줄 세우기
    - 한 스레드가 공유 데이터에 대한 작업을 끝낼 때까지 다른 스레드가 공유 데이터에 접근하지 못하도록 함
  - Mutual exclusion

두 스레드가 프린터에 동시 쓰기 수행

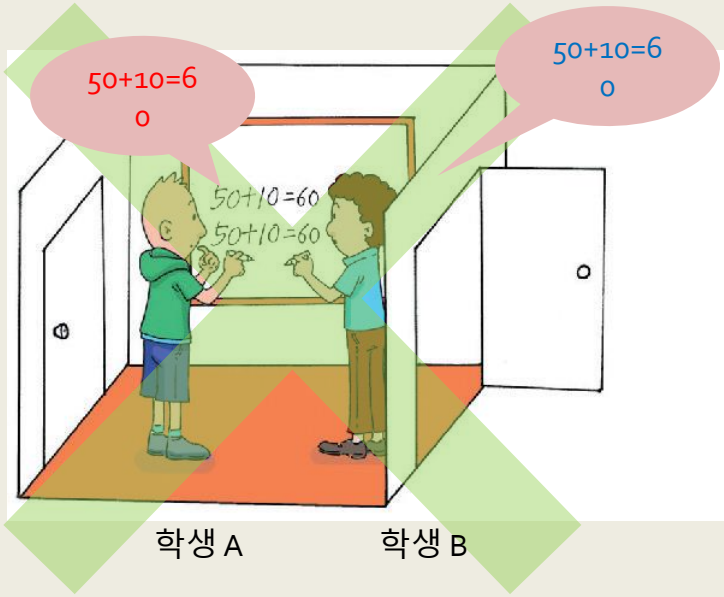


두 개의 스레드가 동시에 프린터에 쓰는 경우 문제 발생



두 개의 스레드가 순서를 지켜 프린터에 쓰는 경우 정상 출력

공유 집계판을 동시 접근하는 경우



두 학생이 동시에 방에 들어와서
집계판을 수정하는 경우
집계판의 결과가 잘못됨

학생 B는 학생 A가
계수를 끝낸 후 계수를 시작함

학생 A는 60을
기록하고 나감

방에 먼저 들어간 학생이
집계를 끝내기를 기다리는 경우
정상 처리

Race condition

- **스레드 간섭(thread interference)**이란 서로 다른 2개 이상의 스레드가 공유 자원을 처리하면 오류를 유발하는 상황이며, 이를 race condition이 존재한다고 한다

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

예제

```
class Counter {
        private int value = 0;
        public void increment() { value++;      }
        public void decrement() { value--;      }
        public void printCounter() {
        System.out.println(value);              }
}
class MyThread extends Thread {
        Counter sharedCounter;
        public MyThread(Counter c) {
                this.sharedCounter = c;
        }
        public void run() {
                int i = 0;
                while (i < 20000) {
                        sharedCounter.increment();
                        sharedCounter.decrement();
                        if (i % 40 == 0)
                                sharedCounter.printCounter();
                        try {
                                sleep((int) (Math.random() * 2));
                        } catch (InterruptedException e) { return; }
                        i++;
                }
        }
}
```

```
public class CounterTest {
        public static void main(String[] args) {
                Counter c = new Counter();    // 하나만 생성(공유객체)
                MyThread th1 = new MyThread(c);
                MyThread th2 = new MyThread(c);
                MyThread th3 = new MyThread(c);
                MyThread th4 = new MyThread(c);
                th1.start();
                th2.start();
                th3.start();
                th4.start();
        }
}
```

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

synchronized 키워드

- synchronized 키워드
  - 한 스레드만이 배타적으로 실행되어야 하는 부분(동기화 코드)을 표시하는 키워드
    - 임계 영역(critical section) 표기 키워드
- synchronized 키워드 사용 가능한 부분
  - 메소드 전체 혹은 코드 블럭
- synchronized 부분이 실행될 때,
  - 실행 스레드는 모니터 소유
    - 모니터란 해당 객체를 독점적으로 사용할 수 있는 권한
  - 모니터를 소유한 스레드가 모니터를 내놓을 때까지 다른 스레드는 대기

```
synchronized void add() {
    int n = getCurrentSum();
    n+=10;
    setCurrentSum(n);
}
```

synchoronized 메소드

```
void execute() {
    // 다른 코드들
    //
    synchronized(this) {
        int n = getCurrentSum();
        n+=10;
        setCurrentSum(n);
    }
    //
    // 다른 코드들
}
```

synchoronized 코드블럭

동기화된 메소드(synchronized methods)

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

```
0
0
0
0
0
1
0
0
...
```

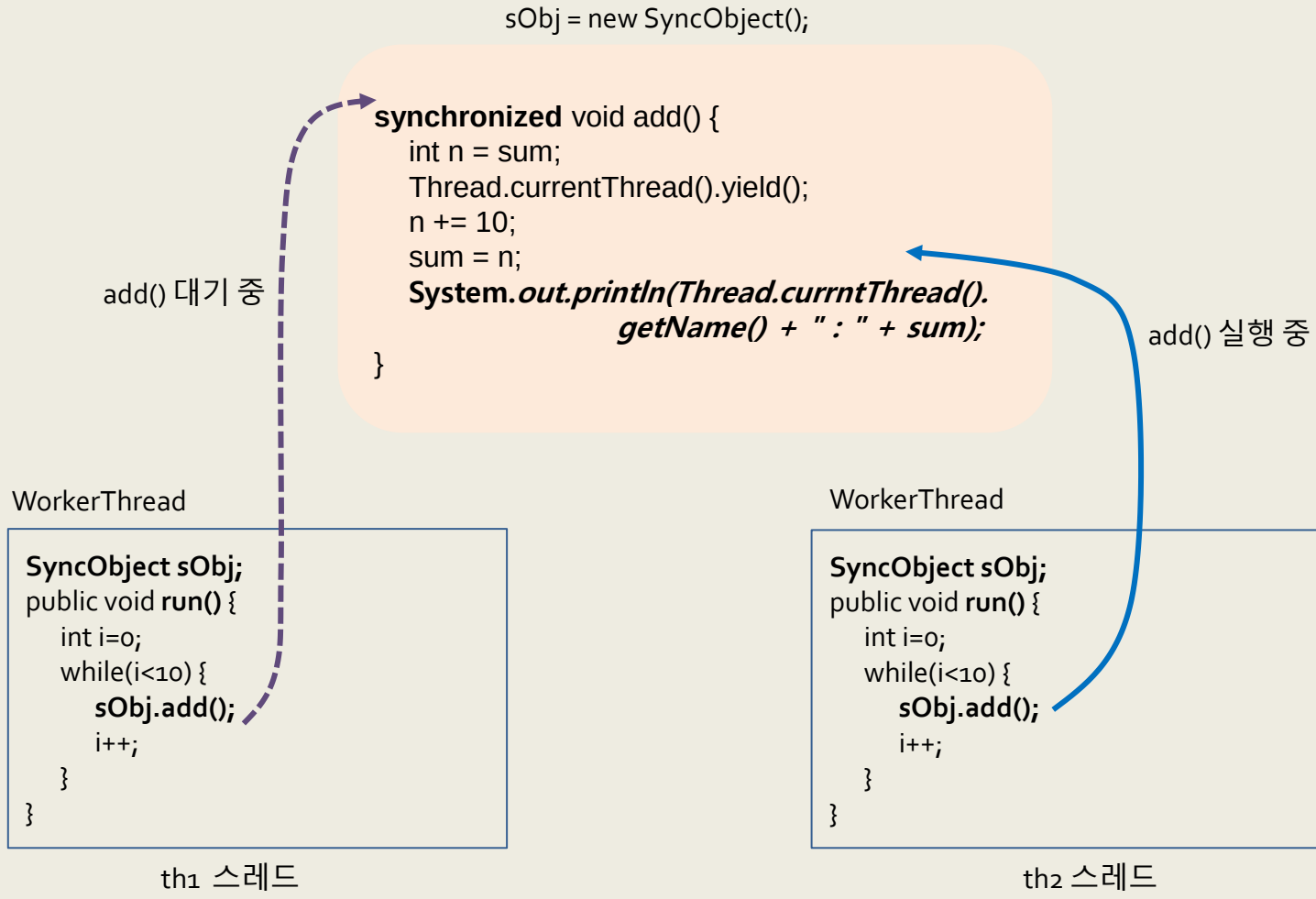
synchronized 사용 예 : 집계판 사례를 코딩

```
public class SynchronizedEx {
    public static void main(String [] args) {
        SyncObject obj = new SyncObject();
        Thread th1 = new WorkerThread("wonho", obj);
        Thread th2 = new WorkerThread("hyosoo", obj);
        th1.start();
        th2.start();
    }
}
class SyncObject {
    int sum = 0;
     synchronized void add() {
        int n = sum;
        Thread.currentThread().yield();
        n += 10;
        sum = n;
        System.out.println(Thread.currntThread().getName() + " : " + sum);
    }
    int getSum() {return sum;}
}
class WorkerThread extends Thread {
    SyncObject  sObj;
    WorkerThread(String name, SyncObject  sObj) {
        super(name);
        this.sObj = sObj;
    }
    public void run() {
        int i=0;
        while(i<10) {
            sObj.add();
            i++;
        }
    }
}
```

- 집계판 : class SyncObject
- 각 학생 : class WorkerThread

```
wonho : 10
hyosoo : 20
wonho : 30
hyosoo : 40
wonho : 50
hyosoo : 60
wonho : 70
hyosoo : 80
hyosoo : 90
hyosoo : 100
hyosoo : 110
hyosoo : 120
hyosoo : 130
hyosoo : 140
wonho : 150
wonho : 160
wonho : 170
wonho : 180
wonho : 190
wonho : 200
```

wonho 와 hyosoo가 각각 10번씩 add()를 호출하였으며 동기화가 잘 이루어져서 최종 누적 점수 sum이 200이 됨


sObj = new SyncObject();
synchronized void add() {
int n = sum;
Thread.currentThread().yield();
n += 10;
sum = n;
System.out.println(Thread.currntThread().
getName() + " : " + sum);
}
add() 대기 중
add() 실행 중
WorkerThread
SyncObject sObj;
public void run() {
int i=0;
while(i<10) {
sObj.add();
i++;
}
}
th1 스레드
WorkerThread
SyncObject sObj;
public void run() {
int i=0;
while(i<10) {
sObj.add();
i++;
}
}
th2 스레드

집계판 예에서 synchronized 사용하지 않을 경우

```
public class SynchronizedEx {
    public static void main(String [] args) {
        SyncObject obj = new SyncObject();
        Thread th1 = new WorkerThread("wonho", obj);
        Thread th2 = new WorkerThread("hyosoo", obj);
        th1.start();
        th2.start();
    }
}
class SyncObject {
    int sum = 0;
     synchronized void add() {
        int n = sum;
        Thread.currentThread().yield();
        n += 10;
        sum = n;
        System.out.println(Thread.currntThread().getName() + " : " + sum);
    }
    int getSum() {return sum;}
}
class WorkerThread extends Thread {
    SyncObject sObj;
    WorkerThread(String name, SyncObject sObj) {
        super(name);
        this.sObj = sObj;
    }
    public void run() {
        int i=0;
        while(i<10) {
            sObj.add();
            i++;
        }
    }
}
```

```
w : 10
hyosoo : 20
wonho : 30
hyosoo : 30
wonho : 40
hyosoo : 50
wonho : 50
wonho : 60
hyosoo : 60
wonho : 70
hyosoo : 70
wonho : 80
hyosoo : 90
wonho : 100
hyosoo : 100
wonho : 110
hyosoo : 120
wonho : 130
hyosoo : 140
hyosoo : 150
```

add() 충돌

add() 충돌

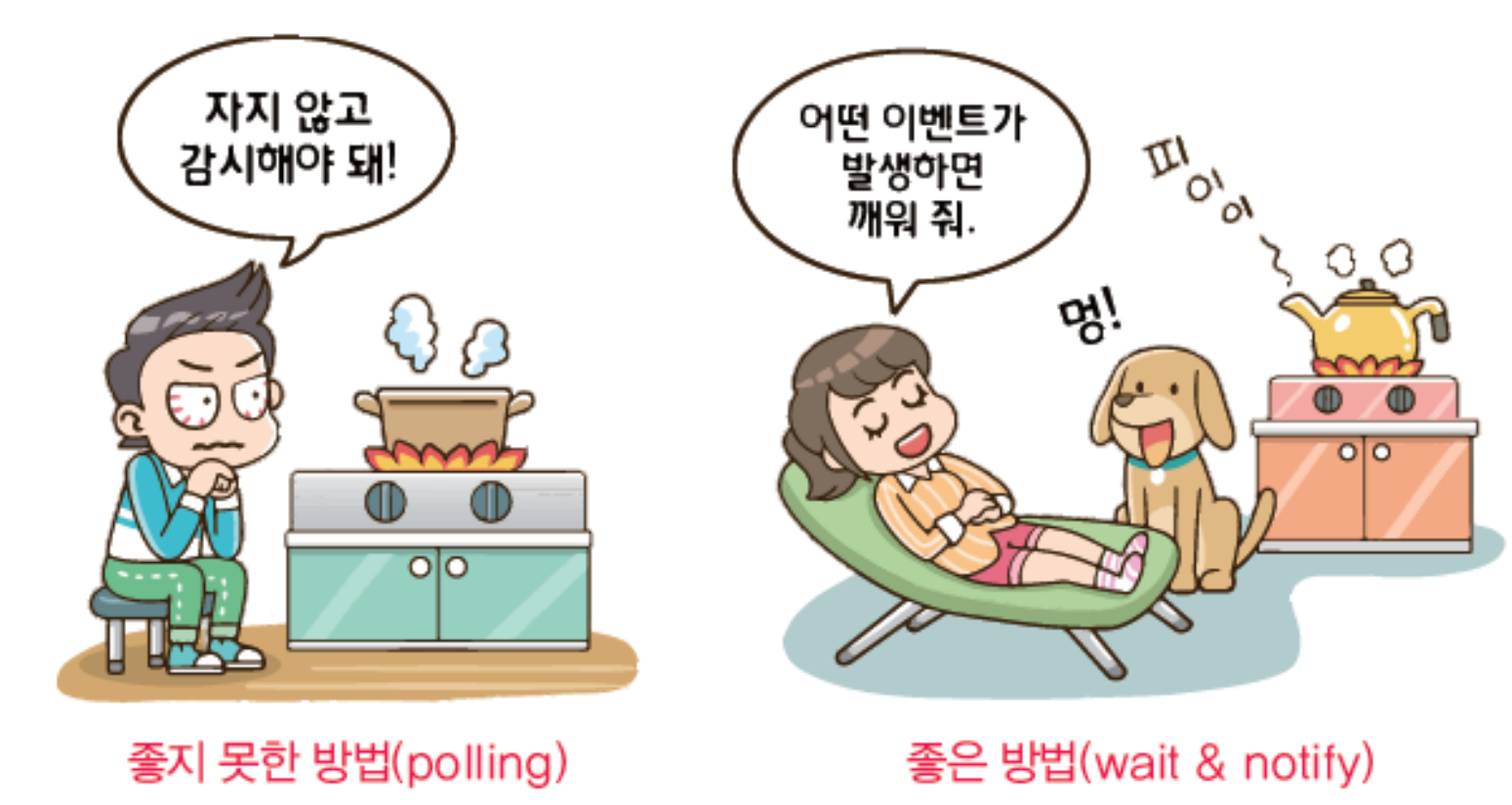
add() 충돌

add() 충돌

wonho 와 hyosoo가 각각 10번씩 add()를 호출하였지만 동기화가 이루어지지 않아 공유 변수 sum에 대한 접근에 충돌이 있었고, 수를 많이 잃어버리게 되어 누적 점수가 150 밖에 되지 못함

Solutions

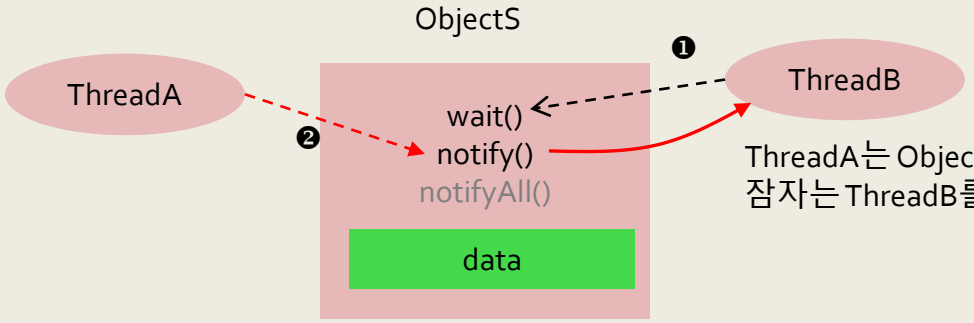
- Polling
- Event driven



wait(), notify(), notifyAll()

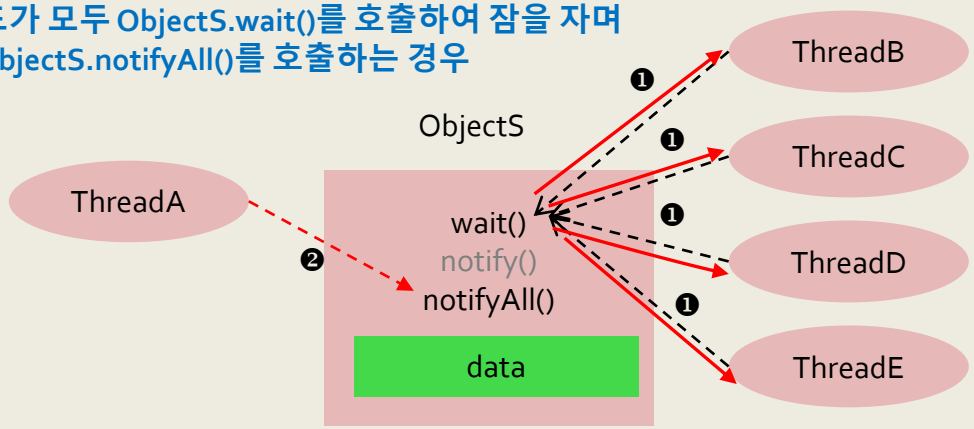
- 동기화 객체
  - 두 개 이상의 스레드 사이에 동기화 작업에 사용되는 객체
- 동기화 메소드
  - synchronized 블럭 내에서만 사용되어야 함
  - wait() : 다른 스레드가 notify()를 불러줄 때까지 기다린다.
  - notify()
    - wait() 로 대기중인 스레드를 깨우고 RUNNABLE 상태로 변경
    - 2개 이상의 스레드가 대기중이라도 오직 한 개의 스레드만 깨워 RUNNABLE 상태로 한다.
  - notifyAll()
    - wait()로 대기중인 모든 스레드를 깨우고 이들을 모두 RUNNABLE 상태로 변경
- 동기화 메소드는 Object의 메소드이다.
  - 모든 객체가 동기화 객체가 될 수 있다.
  - Thread 객체도 동기화 객체로 사용될 수 있다.

하나의 Thread가 ObjectS.wait()를 호출하여 잠을 자는 경우



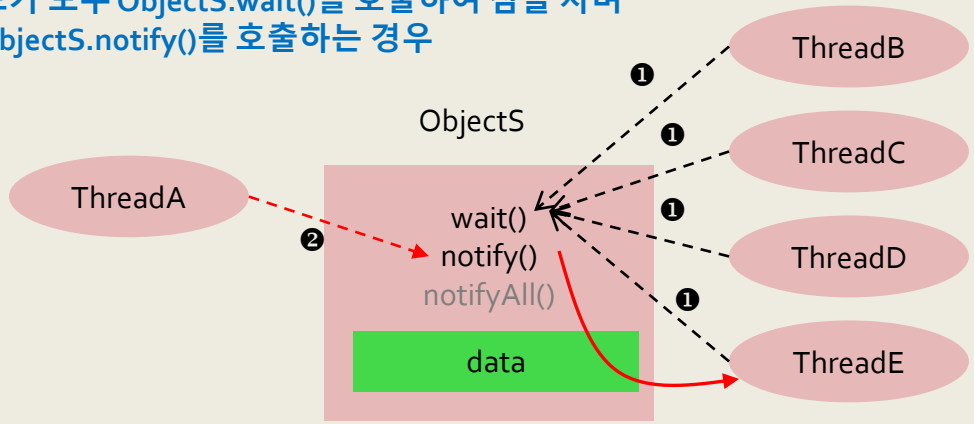
ThreadA는 ObjectS.notify()를 호출하여
잠자는 ThreadB를 깨운다.

4 개의 스레드가 모두 ObjectS.wait()를 호출하여 잠을 자며 ThreadA는 ObjectS.notifyAll()를 호출하는 경우



잠자는 4 개의 스레드를
모두 깨운다.

4 개의 스레드가 모두 ObjectS.wait()를 호출하여 잠을 자며 ThreadA는 ObjectS.notify()를 호출하는 경우



잠자는 4개의 스레드 중 하나만 깨운다.
선택받지 못한 3 개의 스레드는
계속 잠을 잔다.

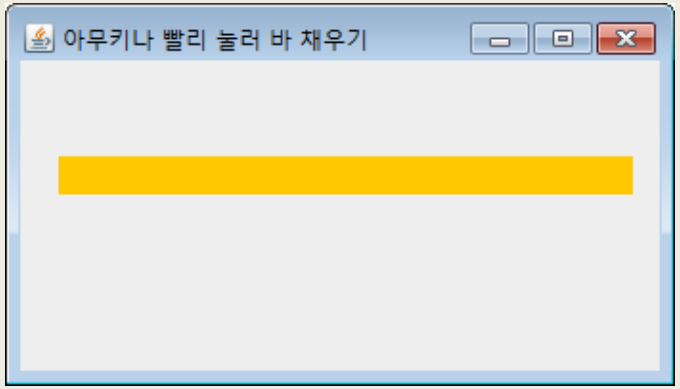
예제 : wait(), notify()를 이용한 바 채우기

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class MyLabel extends JLabel {
    int barSize = 0; // 바의 크기
    int maxBarSize;

    MyLabel(int maxBarSize) {
        this.maxBarSize = maxBarSize;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.MAGENTA);
        int width = (int)((double)(this.getWidth()))
                /maxBarSize*barSize);
        if(width==0) return;
        g.fillRect(0, 0, width, this.getHeight());
    }

    synchronized void fill() {
        if(barSize == maxBarSize) {
            try {
                wait();
            } catch (InterruptedException e)
{ return; }
        }
        barSize++;
        repaint(); // 바 다시 그리기
        notify();
    }
```

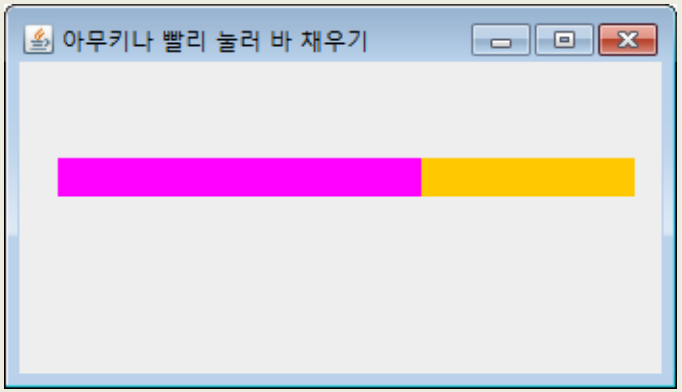
```
    synchronized void consume() {
        if(barSize == 0) {
            try {
                wait();
            } catch (InterruptedException e)
                { return; }
        }
        barSize--;
        repaint(); // 바 다시 그리기
        notify();
    }
}

class ConsumerThread extends Thread {
    MyLabel bar;

    ConsumerThread(MyLabel bar) {
        this.bar = bar;
    }
    public void run() {
        while(true) {
            try {
                sleep(200);
                bar.consume();
            } catch (InterruptedException e)
             { return; }
        }
    }
}
```

```
public class TabAndThreadEx  extends
JFrame {
    MyLabel bar = new MyLabel(100);
    TabAndThreadEx(String title) {
        super(title);
        this.setDefaultCloseOperation
                (JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(null);
        bar.setBackground(Color.ORANGE);
        bar.setOpaque(true);
        bar.setLocation(20,  50);
        bar.setSize(300, 20);
        c.add(bar);

        c.addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent
e)
            {
                bar.fill();
            }
        });
        setSize(350,200);
        setVisible(true);

        c.requestFocus();
        ConsumerThread th = new
            ConsumerThread(bar);
        th.start(); // 스레드 시작
    }

    public static void main(String[] args) {
        new TabAndThreadEx(
            "아무키나 빨리 눌러 바 채우기");
    }
}
```

실행 결과



초기 화면



키를 반복하여 빨리 누른 화면

Buffer 클래스

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

Producer

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

Consumer

```
class Consumer implements Runnable {
        private Buffer buffer;
        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) {
                        }
                }
        }
}
```

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