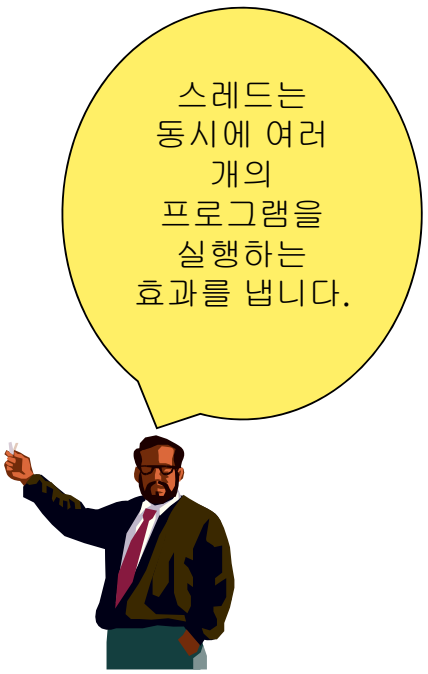
*Power Java*

# 제16장 스레드



## 이번 장에서 학습할 내용

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



# 병렬 처리


그림 16.1 병렬 처리의 예



# 스레드란?

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

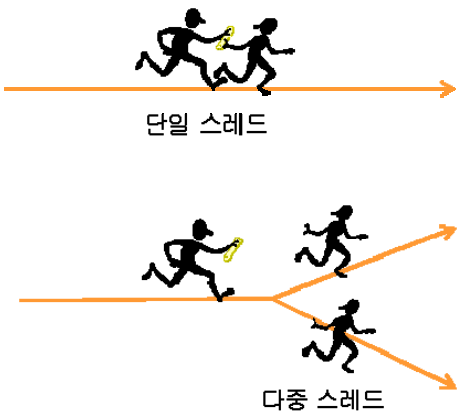



그림 16.2 다중 스레드의 개념

# 프로세스와 스레드

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

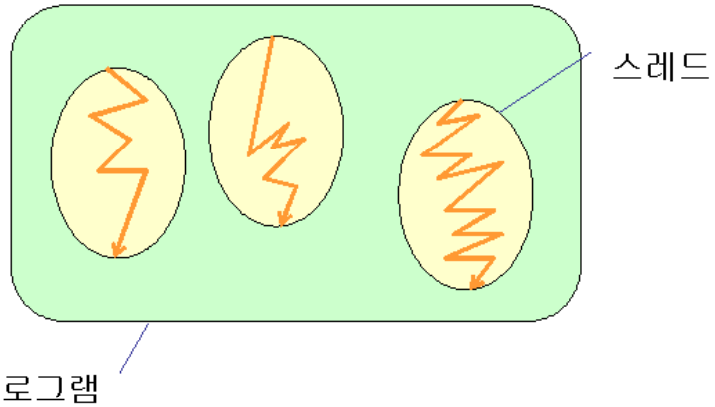



그림 16.3 스레드는 하나의 프로세스 안에 존재한다.



# 스레드를 사용하는 이유

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

# 중간 점검 문제

1. 스레드와 프로세스의 결정적인 차이점은 무엇인가?

2. 스레드를 사용해야만 하는 프로그램을 생각하여 보자.

3. 다중 스레딩에서 발생할 수 있는 문제에는 어떤 것들이 있을까? 추측하여 보라.

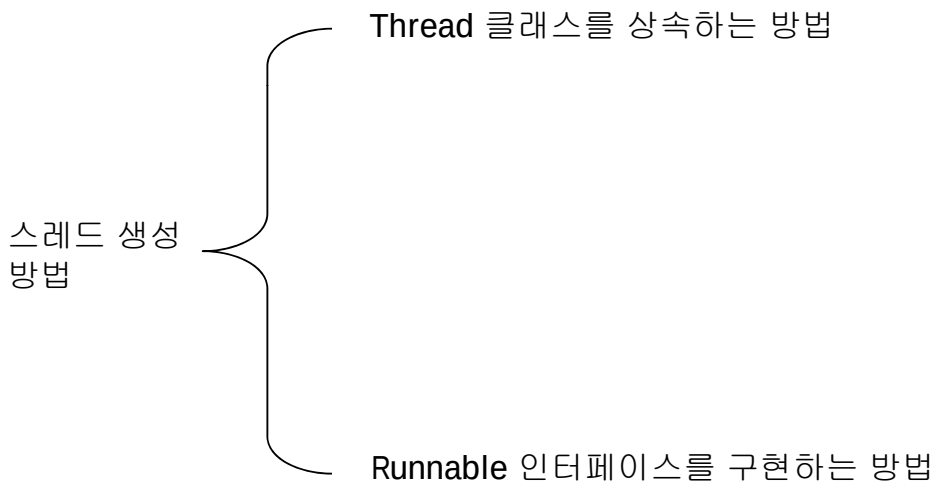


# 스레드 생성과 실행

- 스레드는 Thread 클래스가 담당한다.

| 메소드 | 설명 |
|---|---|
| Thread() | 매개 변수가 없는 기본 생성자 |
| Thread(String *name*) | 이름이 name인 Thread 객체를 생성한다 |
| Thread(Runnable *target*, String *name*) | Runnable을 구현하는 객체로부터 스레드를 생성한다. |
| static int activeCount() | 현재 활동중인 스레드의 개수를 반환한다. |
| String getName() | 스레드의 이름을 반환 |
| int getPriority() | 스레드의 우선 순위를 반환 |
| void interrupt() | 현재의 스레드를 중단한다. |
| boolean isInterrupted() | 현재의 스레드가 중단될 수 있는지를 검사 |
| void setPriority(int *priority*) | 스레드의 우선 순위를 지정한다. |
| void setName(String *name*) | 스레드의 이름을 지정한다. |
| static void sleep(int *milliseconds*) | 현재의 스레드를 지정된 시간만큼 재운다. |
| void run() | 스레드가 하여야하는 작업을 메소드 안에 위치시킨다. |
| void start() | 스레드를 시작한다. |
| static void yield() | 현재 스레드를 다른 스레드에 양보하게 만든다. |

# 스레드를 생성하는 방법






# Thread 클래스를 상속하는 방법

```
class Counting extends Thread {
        public void run() {
                for (int i = 0; i < 10; i++)
                        System.out.println(i);
        }
}
```

```
public class Test {
        public static void main(String args[]) {
                Thread t = new Counting();
                t.start();
        }
}
```

*실행결과*

```
0
1
2
...
9
```

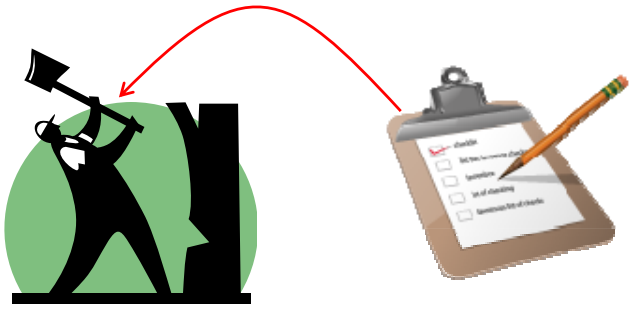
# Runnable 인터페이스를 구현하는 방법

① Runnable을 구현하는 클래스를 생성한다.

② Runnable 클래스에 run() 메소드를 작성한다.

③ Thread 클래스의 인스턴스를 생성하고, Runnable 객체를 Thread 생성자의 매개 변수로 넘긴다.

④ Thread 객체의 start() 메소드를 호출하여야 한다.



Thread 객체 = 일꾼

Runnable 객체 = 작업의 내용



# Runnable 인터페이스를 구현하는 방법

```
class Counting implements Runnable {
    public void run() {
            for (int i = 0; i < 10; i++)
                System.out.println(i);
    }
}
```

```
public class Test {
    public static void main(String args[]) {
                Counting c = new Counting();
                Thread t = new Thread(c);
                t.start();
            }
}
```

# 어떤 방법이 좋은가?

- 자바에서 다중 상속이 불가능한 것을 감안한다면 Runnable 인터페이스를 사용하는 것이 좋다.
- Runnable 인터페이스를 사용하면 고수준의 스레드 관리 API도 사용할 수 있다.




# 예제

TestThread.java

```
class MyRunnable implements Runnable {
    String myName;
    public MyRunnable(String name) {
            myName = name;
    }
    public void run() {
            for (int i = 0; i < 10; i++) {
                    System.out.println(myName + i);
            }
    }
}
public class TestThread {
    public static void main(String[] args) {
            Thread t1 = new Thread(new MyRunnable("First Thread"));
            Thread t2 = new Thread(new MyRunnable("Second Thread"));
            t1.start();
            t2.start();
    }
}
```

실행결과

```
First Thread0
First Thread1
First Thread2
...
Second Thread8
Second Thread9
```

```
import java.util.Random;
class Horse implements Runnable {
        String name;
        private int sleepTime;
        private final static Random generator = new Random();
        public Horse(String name) {
                this.name = name;
                sleepTime = generator.nextInt(3000);
        }
        public void run() {
                try {
                        Thread.sleep(sleepTime);
                } catch (InterruptedException e) {
                }
                System.out.println(name + "말이 경주를 완료하였습니다");
        }
}
public class TestThread1 {
        public static void main(String[] args) {
                Thread t1 = new Thread(new Horse("질풍"));
                Thread t2 = new Thread(new Horse("번개"));
                Thread t3 = new Thread(new Horse("적토마"));
                t1.start();
                t2.start();
                t3.start();
        }
}
```

실행결과

```
질풍말이 경주를 완료하였습니다
적토마말이 경주를 완료하였습니다
번개말이 경주를 완료하였습니다
```

# 중간 점검 문제

1. 스레딩을 담당하는 클래스의 이름은?

2. Thread를 상속받는 방법의 문제점은 무엇인가?

# 스레드 상태

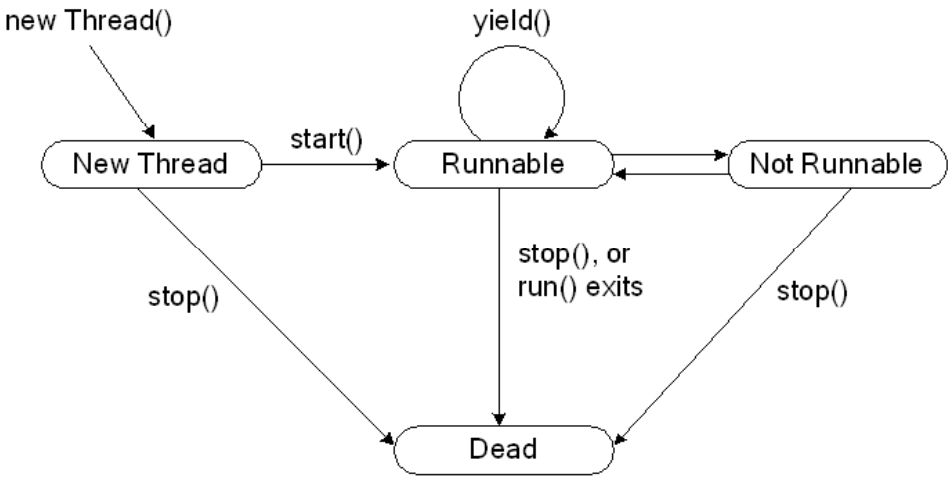
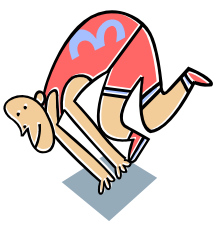



그림 16-4 스레드의 상태



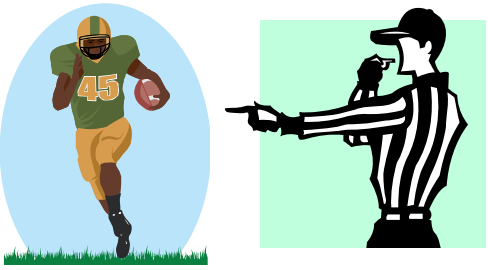
# 생성 상태와 실행 가능 상태

- 생성 상태
  - Thread 클래스를 이용하여 새로운 스레드를 생성
  - start()는 생성된 스레드를 시작
  - stop()은 생성된 스레드를 멈추게 한다.
- 실행 가능 상태
  - 스레드가 스케줄링 큐에 넣어지고 스케줄러에 의해 우선순위에 따라 실행



# 실행 중지 상태

- 실행 가능한 상태에서 다음의 이벤트가 발생하면 실행 중지 상태로 된다.
  - 스레드나 다른 스레드가 suspend()를 호출하는 경우
  - 스레드가 wait()를 호출하는 경우
  - 스레드가 sleep()을 호출하는 경우
  - 스레드가 입출력 작업을 하기 위해 대기하는 경우

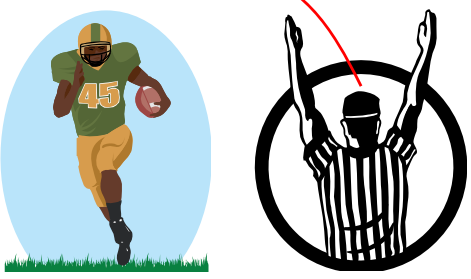




# 강제적인 종료

```
Thread my = new Thread() ;

my.start() ;
try {
        Thread.currentThread().sleep(1000) ;
} catch(InterruptedException e) {}
my.stop() ;
```



# 중간 점검 문제

1. Thread의 run() 메소드의 역할은?

2. Thread의 start(), stop() 메소드의 역할은?

3. 어떤 일이 발생하면 스레드가 실행 중지 상태로 가는가?



# 스레드 스케줄링

*thread1.c*

```
public class CountDown {
    public static void main(String args[]) throws InterruptedException {

        for (int i = 10; i >= 0; i--) {
            //1초 동안 중단
            Thread.sleep(1000);
            //카운트를 재개 한다.
            System.out.println(i);
        }
    }
}
```

*실행결과*

```
10
9
...
0
```

# 동기화

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

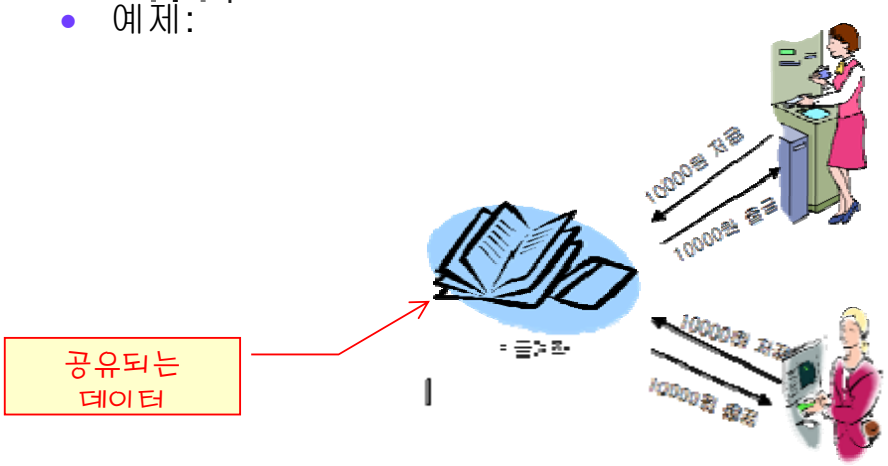



그림 16.5 동기화 문제는 동시에 실행되는 스레드들 사이에 공유되는 데이터가 있는 경우에 발생한다.



# 은행 계좌

```
class BankAccount {
        int balance;
        public void deposit(int amount) {
                balance += amount;
        }
        public void withdraw(int amount) {
                balance -= amount;
        }
        public int getBalance() {
                return balance;
        }
}
```

# 사용자

```
class User implements Runnable {
        BankAccount account;
        public User(BankAccount account) {
                this.account = account;
        }
        public void run() {
                for (int i = 0; i < 100; i++) {
                        account.deposit(10000);
                        try {
                                Thread.sleep(99);
                        } catch (InterruptedException e) {
                        }
                        account.withdraw(10000);
                        if (account.getBalance() < 0) {
                                System.out.println("오류 발생!!!");
                        }
                }
        }
}
```



# BankTest 클래스

```
public class BankTest {
        static BankAccount account = new BankAccount();

        public static void main(String[] args) {

                Thread one = new Thread(new User(account));
                Thread two = new Thread(new User(account));

                one.start();
                two.start();
         }
}
```

*실행결과*

```
오류 발생!!!
...
```

# 오류가 발생하는 원인

- 오류가 발생하는 여러 가지의 경우가 있지만 그 중의 하나는 다음과 같다.

①스레드 one: balance의 현재값을 CPU의 레지스터로 가져온다.
②스레드 two: balance의 현재값을 CPU의 레지스터로 가져온다.
③스레드 one: 레지스터로 가져온 값을 10000만큼 증가시킨다. (+10000)
④스레드 two: 레지스터로 가져온 값을 10000만큼 증가시킨다. (+10000)
⑤스레드 one: 결과를 balance에 저장한다. balance는 10000이 된다.
⑥스레드 two: 결과를 balance에 저장한다. balance는 10000이 된다.

스레드 one의 저금은 없어진다.



# 동기화 문제

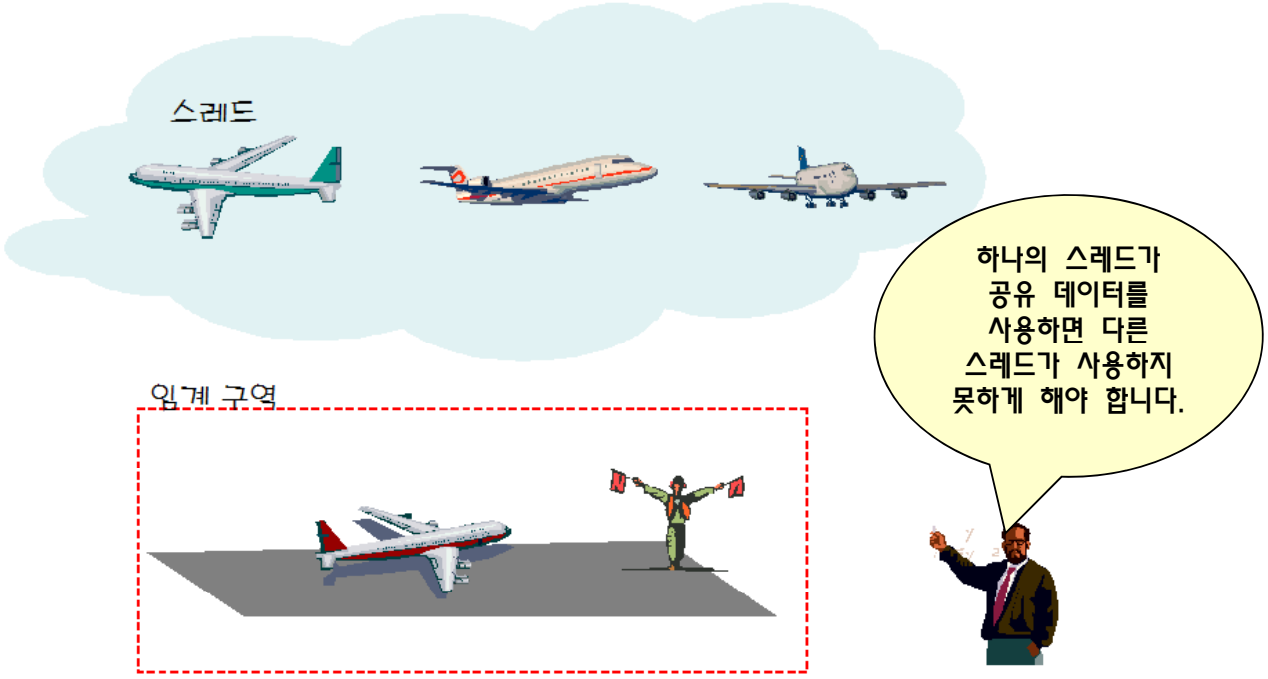



그림 16.6 동기화의 개념

# 동기화 메소드

- 자바가 제공하는 동기화 방법

```
public  synchronized void deposit(int amount) {
        balance += amount;
    }

    public synchronized void widthdraw(int amount) {
        balance -= amount;
    }
    public synchronized int getBalance() {
            return balance;
}
```

한순간에 하나의 스레드만을 허용



# 스레드간의 조정

- 만약 두개의 스레드가 데이터를 주고 받는 경우에 발생

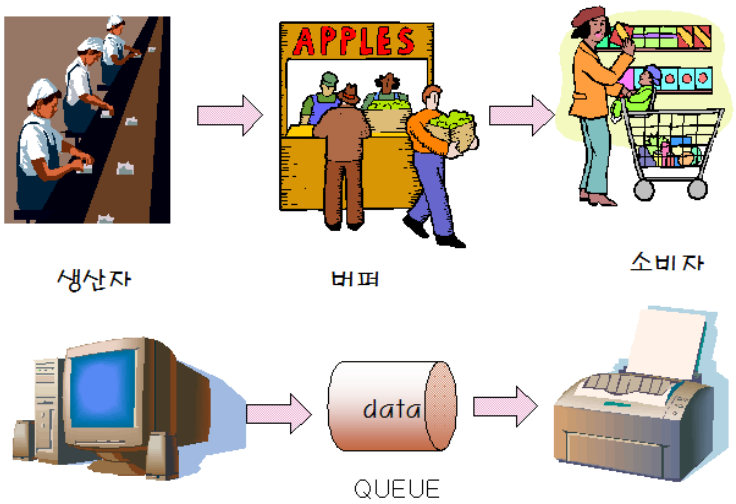



그림 16.7 생산자와 소비자 문제

# Buffer 클래스

- 생산자와 소비자가 공유

```
class Buffer {
        private int data;

        public synchronized int get() {
                return data;
        }

        public synchronized void put(int d) {
                data = d;
        }
}
```



# 생산자 클래스

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

# 소비자 클래스

```
class Consumer extends Thread {
        private Buffer buffer;
        private int data;

        public Consumer(Buffer b) {
                buffer = b;
        }

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



# 테스트 클래스

```
public class CoordinationTest {
        public static void main(String[] args) {
                Buffer b = new Buffer();
                Producer t1 = new Producer(b);
                Consumer t2 = new Consumer(b);
                t1.start();
                t2.start();
        }
}
```

*실행결과*

```
소비자: 0번 케익을 소비하였습니다.
생산자: 0번 케익을 생산하였습니다.
생산자: 1번 케익을 생산하였습니다.
생산자: 2번 케익을 생산하였습니다.
소비자: 2번 케익을 소비하였습니다.
...
생산자: 9번 케익을 생산하였습니다.
소비자: 9번 케익을 소비하였습니다.
소비자: 9번 케익을 소비하였습니다.
소비자: 9번 케익을 소비하였습니다.
소비자: 9번 케익을 소비하였습니다.
```

생산이 너무 빠름 !!

소비가 너무 빠름!!

# wait()와 notify()

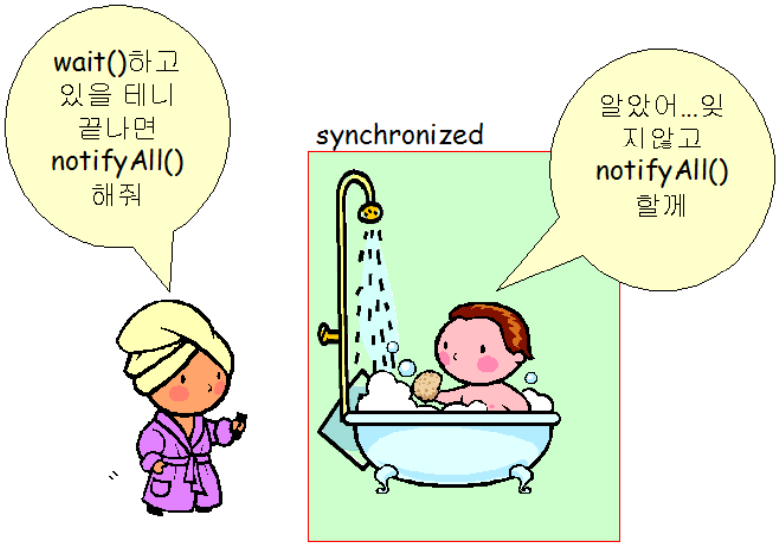



그림 16.8 wait() notify()



# 생산자/소비자 문제에 적용

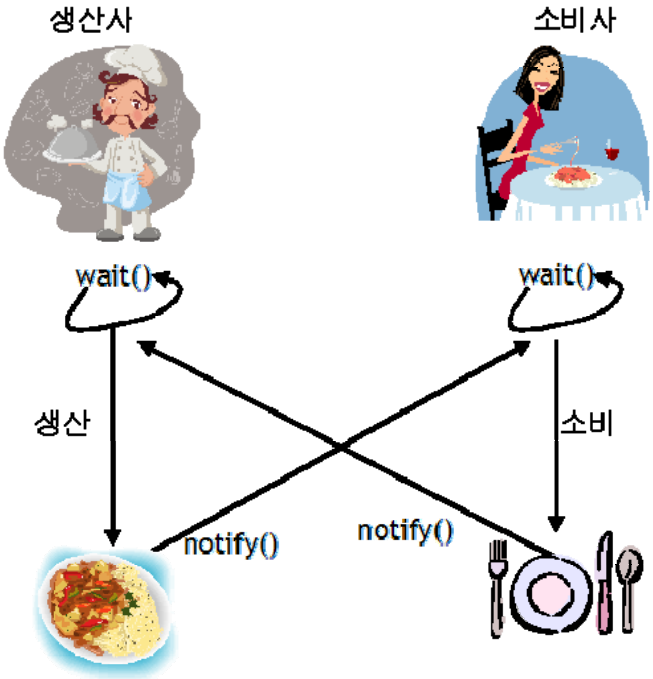



그림 16.9 생산자와 소비자 문제

# 생산자/소비자 문제 해결

```
class Buffer {
        private int data;
        private boolean empty = true;

        public synchronized int get() {
                while (empty) {
                        try {
                                wait();
                        } catch (InterruptedException e) {
                        }
                 }
                // Toggle status.
                empty = true;
                // Notify producer that status has changed.
                notifyAll();
                return data;
        }
```



# 생산자/소비자 문제 해결

```
        public synchronized void put(int data) {
                // Wait until message has been retrieved.
                while (!empty) {
                        try {
                                wait();
                        } catch (InterruptedException e) {
                        }
                }
                // Toggle status.
                empty = false;
                // Store message.
                this.data = data;
                // Notify consumer that status has
                notifyAll();
        }
}
```

*실행결과*

```
생산자: 0번 케익을 생산하였습니다.
소비자: 0번 케익을 소비하였습니다.
생산자: 1번 케익을 생산하였습니다.
소비자: 1번 케익을 소비하였습니다.
생산자: 2번 케익을 생산하였습니다.
소비자: 2번 케익을 소비하였습니다.
...
생산자: 9번 케익을 생산하였습니다.
소비자: 9번 케익을 소비하였습니다.
```

# 중간 점검 문제

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



# Q & A