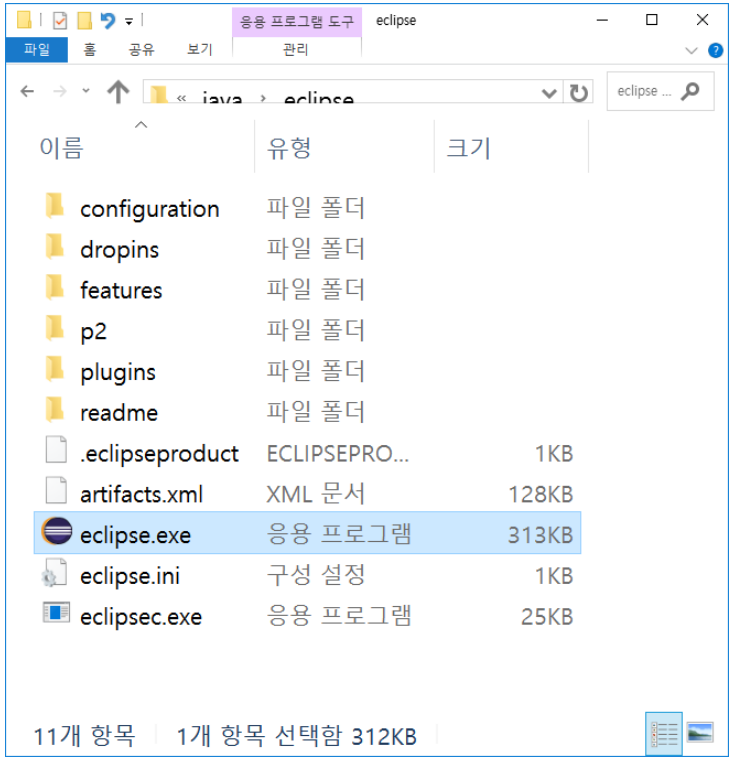
Java의 정석

제 13 장

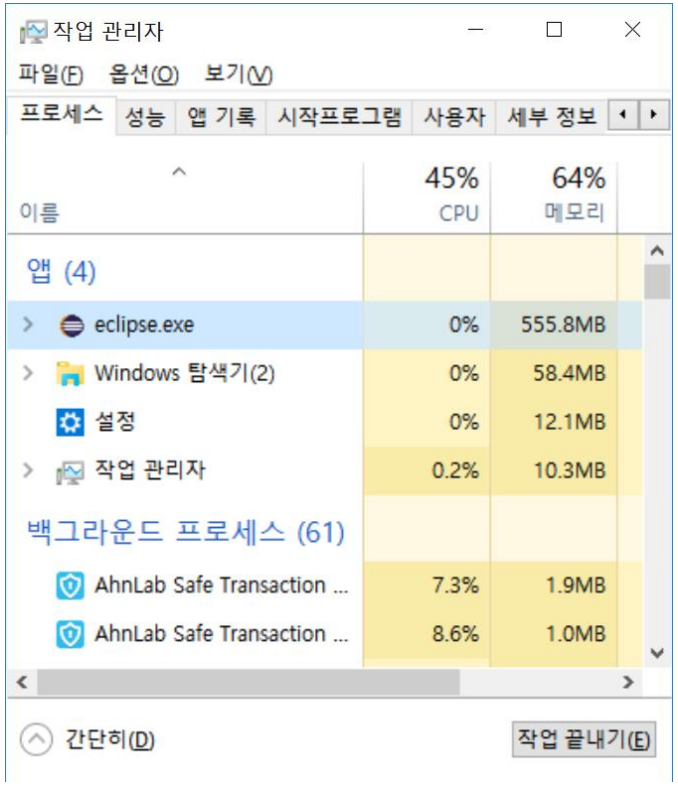
쓰레드 (thread)

1.1 프로세스와 쓰레드(process & thread) (1/2)

▶ 프로그램 : 실행 가능한 파일(HDD, SSD)



▶ 프로세스 : 실행 중인 프로그램(메모리)

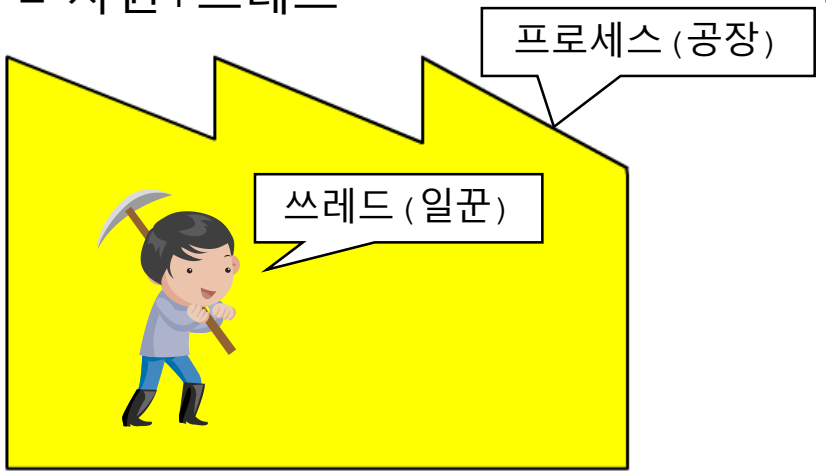


1.1 프로세스와 쓰레드(process & thread) (2/2)

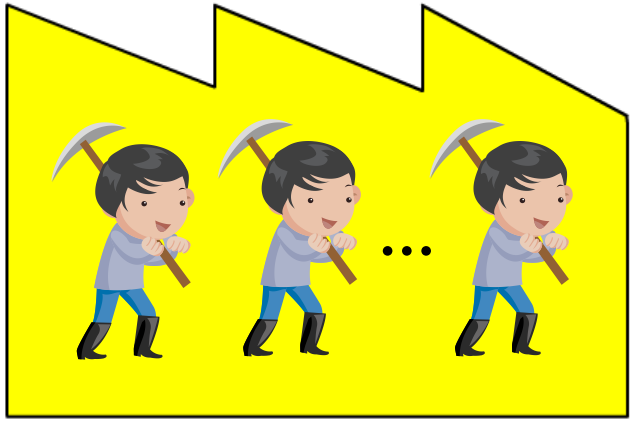
- 프로세스 : 실행 중인 프로그램, 자원(resources)과 쓰레드로 구성
- 쓰레드 : 프로세스 내에서 실제 작업을 수행.
  모든 프로세스는 최소한 하나의 쓰레드를 가지고 있다.

프로세스 : 쓰레드 = 공장 : 일꾼

- 싱글 쓰레드 프로세스
  = 자원+쓰레드



- 멀티 쓰레드 프로세스
  = 자원+쓰레드+쓰레드+...+쓰레드



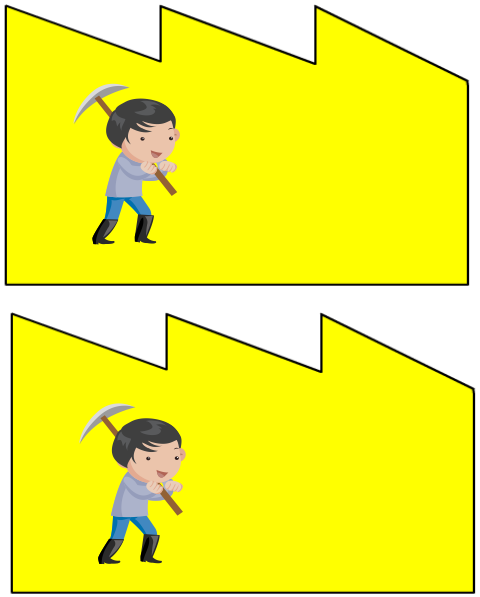
1.2 멀티프로세스 vs. 멀티쓰레드

▶ 멀티 태스킹(멀티 프로세싱) : 동시에 여러 프로세스를 실행시키는 것

▶ 멀티 쓰레딩 : 하나의 프로세스 내에 동시에 여러 쓰레드를 실행시키는 것

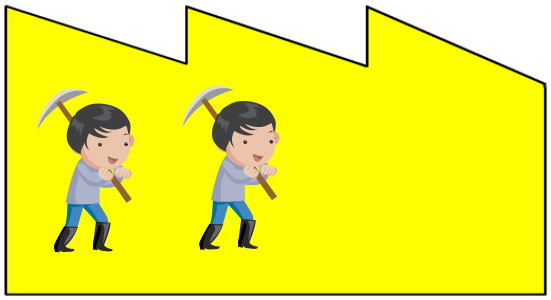
- 프로세스를 생성하는 것보다 쓰레드를 생성하는 비용이 적다.
- 같은 프로세스 내의 쓰레드들은 서로 자원을 공유한다.

2 프로세스 1 쓰레드



VS.

1 프로세스 2 쓰레드



1.3 멀티쓰레드의 장단점

대부분의 프로그램이 멀티쓰레드로 작성되어 있다.
그러나, 멀티쓰레드 프로그래밍이 장점만 있는 것은 아니다.

장점	- 시스템 자원을 보다 **효율적**으로 사용할 수 있다. - 사용자에 대한 **응답성(responseness)이 향상**된다. - 작업이 분리되어 **코드가 간결**해 진다. **"여러 모로 좋다."**
단점	- 동기화(synchronization)에 주의해야 한다. - 교착상태(dead-lock)가 발생하지 않도록 주의해야 한다. - 각 쓰레드가 효율적으로 고르게 실행될 수 있게 해야 한다. **"프로그래밍할 때 고려해야 할 사항들이 많다."**

1.4 쓰레드의 구현과 실행

① Thread클래스를 상속

```
class MyThread extends Thread {
        public void run() { // Thread클래스의 run()을 오버라이딩
            /* 작업내용 */
        }
}
```

② Runnable인터페이스를 구현

```
class MyThread2 implements Runnable {
      public void run() { // Runnable인터페이스의 추상메서드 run()을 구현
            /* 작업내용 */
      }
}
```

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

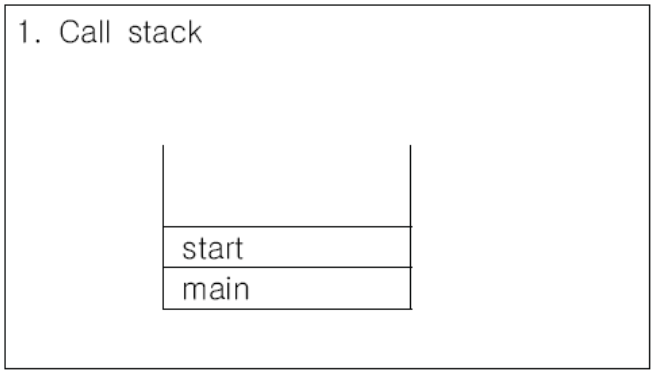
```
MyThread t1 = new MyThread();  // 쓰레드의 생성
t1.start(); // 쓰레드의 실행
```

```
Runnable r = new MyThread2();
Thread  t2 = new Thread(r); // Thread(Runnable r)
//  Thread  t2 = new Thread(new MyThread2());
t2.start();
```

1.5 start()와 run()

```
class ThreadTest {
    public static void main(String args[]){
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

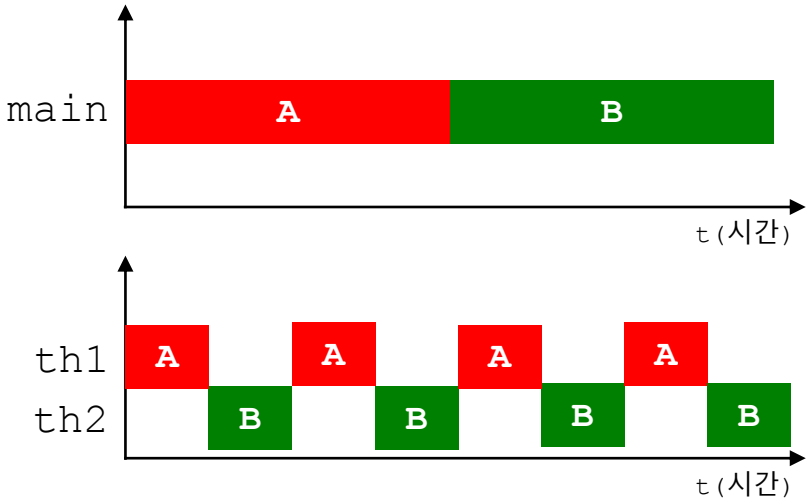
```
class MyThread extends Thread {
    public void run() {
        //...
    }
}
```



2.1 싱글쓰레드 vs. 멀티쓰레드(1/3)

▶ 싱글쓰레드

```
class ThreadTest {
    public static void main(String args[]){
        for(int i=0;i<300;i++) {
            System.out.println("-");
        }

        for(int i=0;i<300;i++) {
            System.out.println("|");
        }
    } // main
}
```



▶ 멀티쓰레드

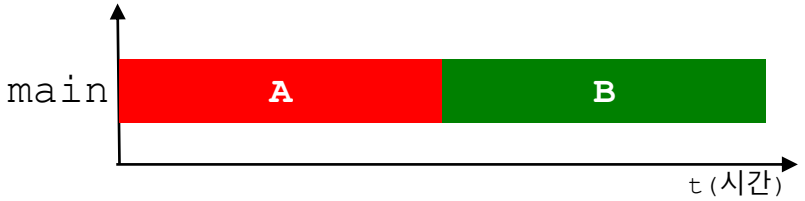
```
class ThreadTest {
    public static void main(String args[]){
        MyThread1 th1 = new MyThread1();
        MyThread2 th2 = new MyThread2();
        th1.start();
        th2.start();
    }
}

class MyThread1 extends Thread {
    public void run() {
        for(int i=0;i<300;i++) {
            System.out.println("-");
        }
    } // run()
}

class MyThread2 extends Thread {
    public void run() {
        for(int i=0;i<300;i++) {
            System.out.println("|");
        }
    } // run()
}
```

2.1 싱글쓰레드 vs. 멀티쓰레드(2/3) – 병행과 병렬

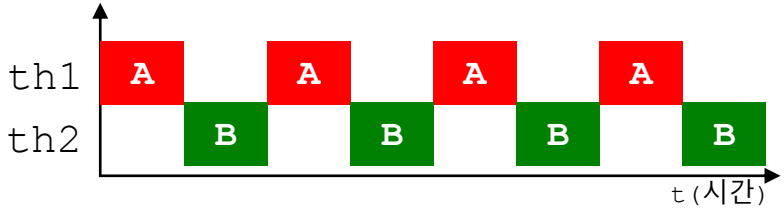
▶ 멀티쓰레드

```
class ThreadTest {
    public static void main(String args[]){
        MyThread1 th1 = new MyThread1();
        MyThread2 th2 = new MyThread2();
        th1.start();
        th2.start();
    }
}

class MyThread1 extends Thread {
    public void run() {
        for(int i=0;i<300;i++) {
            System.out.println("-");
        }
    } // run()
}

class MyThread2 extends Thread {
    public void run() {
        for(int i=0;i<300;i++) {
            System.out.println("|");
        }
    } // run()
}
```

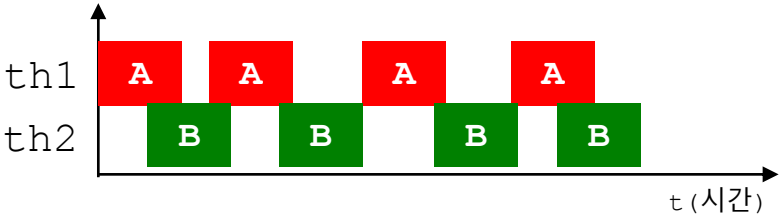
▶ 싱글 코어 – 순차 실행



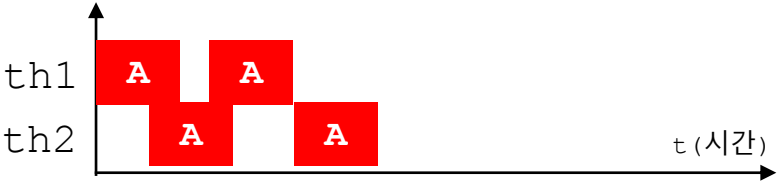
▶ 싱글 코어 – 병행(concurrent)



▶ 멀티 코어 – 병행(concurrent)



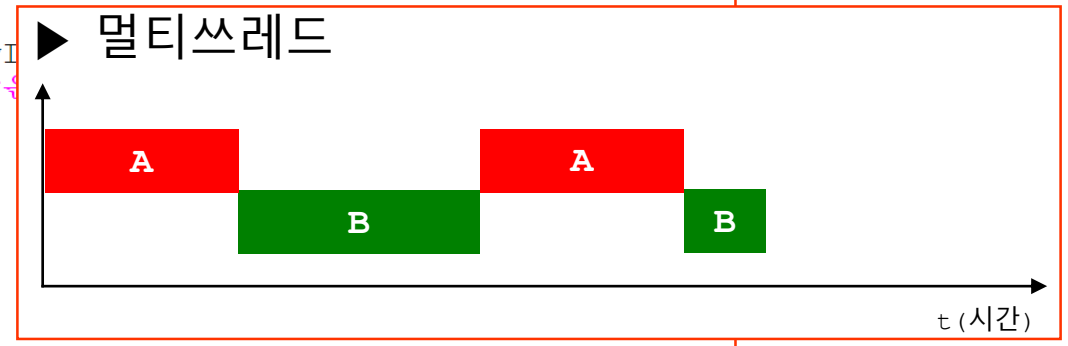
▶ 멀티 코어 – 병렬(parallel)



2.1 싱글쓰레드 vs. 멀티쓰레드(3/3) - blocking

```
class ThreadEx6 {
    public static void main(String[] args){
        String input = JOptionPane.showInputDialog("아무 값이나 입력하세요.");
        System.out.println("입력하신 값은 " + input + "입니다.");

        for(int i=10; i > 0; i--) {
            System.out.println(i);
            try { Thread.sleep(1000); } ca
        }
    } // main
}
```

```
class ThreadEx7 {
    public static void main(String[] arg
        ThreadEx7_1 th1 = new ThreadEx7_
        th1.start();

        String input = JOptionPane.showI
        System.out.println("입력하신 값
    }
}

class ThreadEx7_1 extends Thread {
    public void run() {
        for(int i=10; i > 0; i--) {
            System.out.println(i);
            try { sleep(1000); } catch(Exception e ) {}
        }
    } // run()
}
```



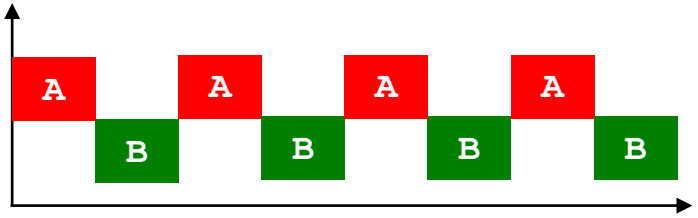
2.2 쓰레드의 우선순위(priority of thread)

- 작업의 중요도에 따라 쓰레드의 우선순위를 다르게 하여 특정 쓰레드가 더 많은 작업시간을 갖게 할 수 있다.

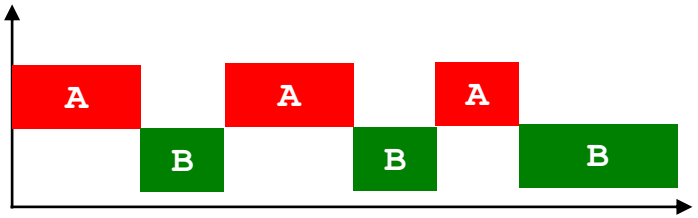
void setPriority(int newPriority)	쓰레드의 우선순위를 지정한 값으로 변경한다.
int getPriority()	쓰레드의 우선순위를 반환한다.

```
public static final int MAX_PRIORITY  = 10   // 최대우선순위
public static final int MIN_PRIORITY  = 1    // 최소우선순위
public static final int NORM_PRIORITY = 5    // 보통우선순위
```

▶ 우선순위가 같은 경우



▶ A의 우선순위가 높은 경우



2.3 쓰레드 그룹(ThreadGroup)

- 서로 관련된 쓰레드를 그룹으로 묶어서 다루기 위한 것(보안상의 이유)
- 모든 쓰레드는 반드시 하나의 쓰레드 그룹에 포함되어 있어야 한다.
- 쓰레드 그룹을 지정하지 않고 생성한 쓰레드는 'main쓰레드 그룹'에 속한다.
- 자신을 생성한 쓰레드(부모 쓰레드)의 그룹과 우선순위를 상속받는다.

```
Thread(ThreadGroup group, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

void destroy( )	쓰레드 그룹과 하위 쓰레드 그룹까지 모두 삭제한다.
int enumerate(Thread[ ] list) int enumerate(Thread[ ] list, boolean recurse) int enumerate(ThreadGroup[ ] list) int enumerate(ThreadGroup[ ] list, boolean recurse)	쓰레드 그룹에 속한 쓰레드 또는 하위 쓰레드 그룹의 목록을 지정된 배열에 담고 그 개수를 반환. 두 번째 매개변수인 recurse의 값을 true로 하면 쓰레드 그룹에 속한 하위 쓰레드 그룹에 쓰레드 또는 쓰레드 그룹까지 배열에 담는다.
int getMaxPriority( )	쓰레드 그룹의 최대우선순위를 반환
String getName( )	쓰레드 그룹의 이름을 반환
ThreadGroup getParent( )	쓰레드 그룹의 상위 쓰레드그룹을 반환
void interrupt( )	쓰레드 그룹에 속한 모든 쓰레드를 interrupt
boolean isDaemon( )	쓰레드 그룹이 데몬 쓰레드그룹인지 확인
boolean isDestroyed( )	쓰레드 그룹이 삭제되었는지 확인
void list( )	쓰레드 그룹에 속한 쓰레드와 하위 쓰레드그룹에 대한 정보를 출력
boolean parentOf(ThreadGroup g)	지정된 쓰레드 그룹의 상위 쓰레드그룹인지 확인
void setDaemon(boolean daemon)	쓰레드 그룹을 데몬 쓰레드그룹으로 설정/해제
void setMaxPriority(int pri)	쓰레드 그룹의 최대우선순위를 설정

2.4 데몬 쓰레드(daemon thread)

- 일반 쓰레드(non-daemon thread)의 작업을 돕는 보조적인 역할을 수행.
- 일반 쓰레드가 모두 종료되면 자동적으로 종료된다.
- 가비지 컬렉터, 자동저장, 화면자동갱신 등에 사용된다.
- 무한루프와 조건문을 이용해서 실행 후 대기하다가 특정조건이 만족되면 작업을 수행하고 다시 대기하도록 작성한다.

```
boolean isDaemon() –
void setDaemon(boole
```

* setDaemon(boolean on)은
그렇지 않으면 IllegalThrea

```
public void run() {
    while(true) {
        try {
            Thread.sleep(3 * 1000); // 3초마다
        } catch(InterruptedException e) {}

        // autoSave의 값이 true이면 autoSave()를 호출한다.
        if(autoSave) {
            autoSave();
        }
    }
}
```

3.1 쓰레드의 실행제어

- 쓰레드의 실행을 제어(스케줄링)할 수 있는 메서드가 제공된다.
  이 들을 활용해서 보다 효율적인 프로그램의 작성할 수 있다.

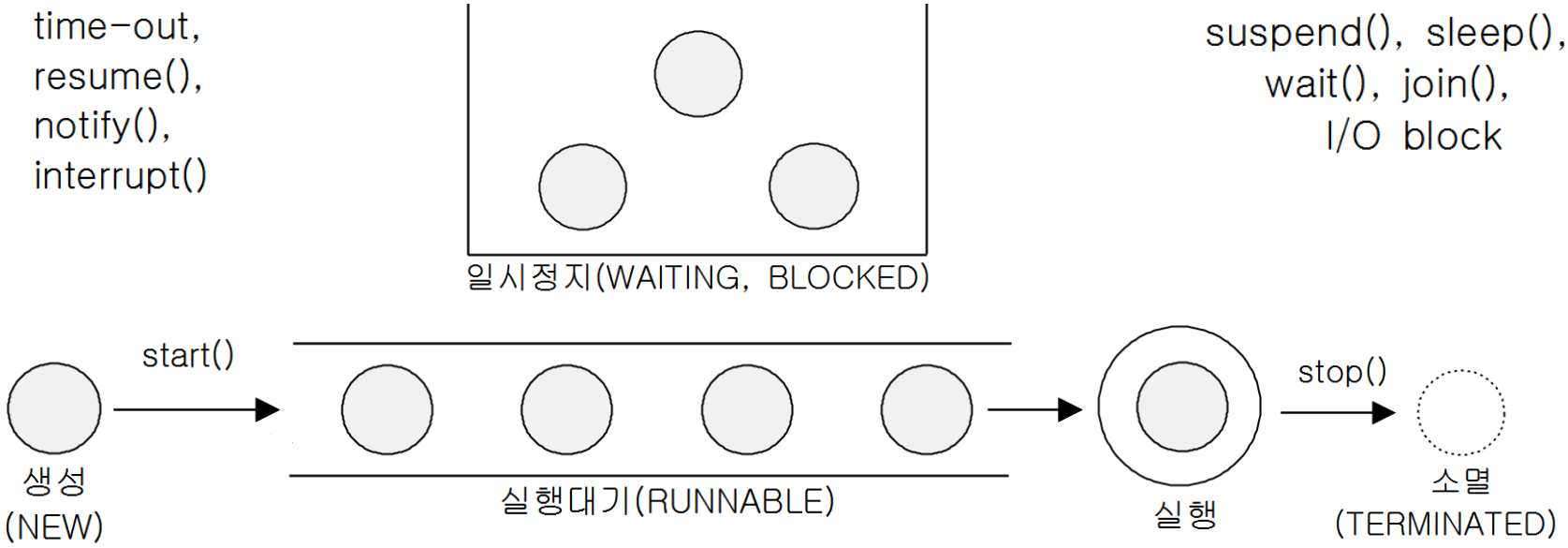
메서드	설 명
static void sleep(long millis) static void sleep(long millis, int nanos)	지정된 시간(천분의 일초 단위)동안 쓰레드를 일시정지시킨다. 지정한 시간이 지나고 나면, 자동적으로 다시 실행대기상태가 된다.
void join( ) void join(long millis) void join(long millis, int nanos)	지정된 시간동안 쓰레드가 실행되도록 한다. 지정된 시간이 지나거나 작업이 종료되면 join( )을 호출한 쓰레드로 다시 돌아와 실행을 계속한다.
void interrupt( )	sleep( )이나 join( )에 의해 일시정지상태인 쓰레드를 깨워서 실행대기상태로 만든다. 해당 쓰레드에서는 InterruptedException이 발생함으로써 일시정지상태를 벗어나게 된다.
void stop( )	쓰레드를 즉시 종료시킨다.
void suspend( )	쓰레드를 일시정지시킨다. resume( )을 호출하면 다시 실행대기상태가 된다.
void resume( )	suspend( )에 의해 일시정지상태에 있는 쓰레드를 실행대기상태로 만든다.
static void yield( )	실행 중에 자신에게 주어진 실행시간을 다른 쓰레드에게 양보(yield)하고 자신은 실행대기상태가 된다.

▲ 표13-2 쓰레드의 스케줄링과 관련된 메서드

* resume(), stop(), suspend()는 쓰레드를 교착상태로 만들기 쉽기 때문에 deprecated되었다.

3.2 쓰레드의 상태(state of thread)

상태	설명
NEW	쓰레드가 생성되고 아직 start( )가 호출되지 않은 상태
RUNNABLE	실행 중 또는 실행 가능한 상태
BLOCKED	동기화블럭에 의해서 일시정지된 상태(lock이 풀릴 때까지 기다리는 상태)
WAITING, TIMED_WAITING	쓰레드의 작업이 종료되지는 않았지만 실행가능하지 않은(unrunnable) 일시정지 상태. TIMED_WAITING은 일시정지시간이 지정된 경우를 의미한다.
TERMINATED	쓰레드의 작업이 종료된 상태



3.3 쓰레드의 실행제어 메서드(1/5) – sleep()

- 현재 쓰레드를 지정된 시간동안 멈추게 한다.

```
static void sleep(long millis)              // 천분의 일초 단위
static void sleep(long millis, int nanos) // 천분의 일초 + 나노초
```

- 예외처리를 해야 한다.(InterruptedException이 발생하면 깨어남)

```
try {
    Thread.sleep(1, 500000);     // 쓰레드를 0.0015초 동안 멈추게 한다.
} catch(InterruptedException e) {}
```

```
void delay(long millis) {
        try {
            Thread.sleep(millis);
        } catch(InterruptedException e) {}
}
```

- 특정 쓰레드를 지정해서 멈추게 하는 것은 불가능하다.

```
try {
    th1.sleep(2000);
} catch(InterruptedException e) {}
```

```
try {
    Thread.sleep(2000);
} catch(InterruptedException e) {}
```

3.3 쓰레드의 실행제어 메서드(2/5) – interrupt()

- 대기상태(WAITING)인 쓰레드를 실행대기 상태(RUNNABLE)로 만든다.

`void interrupt()`	쓰레드의 interrupted상태를 false에서 true로 변경.
`boolean isInterrupted()`	쓰레드의 interrupted상태를 반환.
`static boolean interrupted()`	현재 쓰레드의 interrupted상태를 알려주고, false로 초기화

```
public static void main(String[] args){
    ThreadEx13_2 th1 = new ThreadEx13_2();
    th1.start();
        ...
    th1.interrupt();   // interrupt()를 호출하면, interrupted상태가 true가 된다.
        ...
    System.out.println("isInterrupted():"+ th1.isInterrupted()); // true
```

```
class Thread { // 알기 쉽게 변경한 코드
        ...
    boolean interrupted = false;
        ...
    boolean isInterrupted() {
        return interrupted;
    }

    boolean interrupt() {
        interrupted = true;
    }
}
```

```
class ThreadEx13_2 extends Thread {
    public void run() {
            ...
        while( downloaded && !isInterrupted()) {
            // download를 수행한다.
            ...
        }

        System.out.println("다운로드가 끝났습니다.");
    } // main
}
```

3.3 쓰레드의 실행제어 메서드(3/5) - suspend(), resume(), stop()

- 쓰레드의 실행을 일시정지, 재개, 완전정지 시킨다. 교착상태에 빠지기 쉽다.

void suspend()	쓰레드를 일시정지 시킨다.
void resume()	suspend()에 의해 일시정지된 쓰레드를 실행대기상태로 만든다.
void stop()	쓰레드를 즉시 종료시킨다.

- suspend(), resume(), stop()은 deprecated되었으므로, 직접 구현해야 한다.

```
class ThreadEx17_1 implements Runnable {
    boolean suspended = false;
    boolean stopped   = false;

    public void run() {
        while(!stopped) {
            if(!suspended) {
                /* 쓰레드가 수행할 코드를 작성 */
            }
        }
    }
    public void suspend() { suspended = true;  }
    public void resume()  { suspended = false; }
    public void stop()    { stopped = true;    }
}
```

3.3 쓰레드의 실행제어 메서드(4/5) – yield()

- 남은 시간을 다음 쓰레드에게 양보하고, 자신(현재 쓰레드)은 실행대기한다.
- yield()와 interrupt()를 적절히 사용하면, 응답성과 효율을 높일 수 있다.

```
class MyThreadEx18 implements Runnable {
    boolean suspended = false;
    boolean stopped = false;

    Thread th;

    MyThreadEx18(String name) {
        th = new Thread(this, name);
    }

    public void run() {
        while(!stopped) {
            if(!suspended) {
                /*
                    작업수행
                */
                try {
                    Thread.sleep(1000);
                } catch(InterruptedException e) {}
            } else {
                Thread.yield();
            } // if
        } // while
    }
```

```
    public void start() {
        th.start();
    }

    public void resume() {
        suspended = false;
    }

    public void suspend() {
        suspended = true;
        th.interrupt();
    }

    public void stop() {
        stopped = true;
        th.interrupt();
    }
```

3.3 쓰레드의 실행제어 메서드(5/5) – join()

- 지정된 시간동안 특정 쓰레드가 작업하는 것을 기다린다.

```
void join()                          // 작업이 모두 끝날 때까지
void join(long millis)               // 천분의 일초 동안
void join(long millis, int nanos)    // 천분의 일초 + 나노초 동안
```

- 예외처리를 해야 한다.(InterruptedException이 발생하면 작업 재개)

```
public static void main(String args[]) {
    ThreadEx19_1 th1 = new ThreadEx19_1();
    ThreadEx19_2 th2 = new ThreadEx19_2();
    th1.start();
    th2.start();
    startTime = System.currentTimeMillis();

    try {
        th1.join(); // main쓰레드가 th1의 작업이 끝날 때까지 기다린다.
        th2.join(); // main쓰레드가 th2의 작업이 끝날 때까지 기다린다.
    } catch(InterruptedException e) {}

    System.out.print("소요시간:" + (System.currentTimeMillis()
                                    - ThreadEx19.startTime));
} // main
```

3.4 쓰레드의 실행제어 예제 – join() & interrupt()

```
public void run() {
    while(true) {
        try {
            Thread.sleep(10 * 1000); // 10초를 기다린다.
        } catch(InterruptedException e) {
            System.out.println("Awaken by interrupt().");
        }

        gc(); // garbage collection을 수행한다.
        System.out.println("Garbage Collected. Free Memory :"+ freeMemory());
    }
}
```

```
for(int i=0; i < 20; i++) {
    requiredMemory = (int)(Math.random() * 10) * 20;
    // 필요한 메모리가 사용할 수 있는 양보다 적거나 전체 메모리의 60%이상 사용했을 경우 gc를 깨운다.
    if(gc.freeMemory() < requiredMemory ||
       gc.freeMemory() < gc.totalMemory() * 0.4)
    {
        gc.interrupt(); // 잠자고 있는 쓰레드 gc를 깨운다.



    }
    gc.usedMemory += requiredMemory;
    System.out.println("usedMemory:"+gc.usedMemory);
}
```

1.13 쓰레드의 동기화 - synchronized

- 한 번에 하나의 쓰레드만 객체에 접근할 수 있도록 객체에 락(lock)을 걸어서 데이터의 일관성을 유지하는 것.

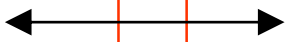
1. 특정한 객체에 lock을 걸고자 할 때

```
synchronized(객체의 참조변수) {
    //...
}
```

2. 메서드에 lock을 걸고자할 때

```
public synchronized void calcSum() {
    //...
}
```

```
public synchronized void withdraw(int money) {
    if(balance >= money) {
        try {
            Thread.sleep(1000);
        } catch(Exception e) {}

        balance -= money;
    }
}
```

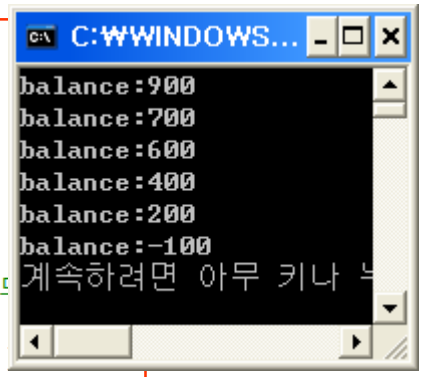


```
public void withdraw(int money) {
    synchronized(this) {
        if(balance >= money) {
            try {
                Thread.sleep(1000);
            } catch(Exception e) {}

            balance -= money;
        }
    } // synchronized(this)
}
```

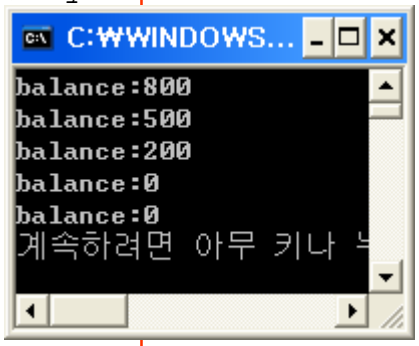
1.13 쓰레드의 동기화 - Example

```
class Account2 {
    private int balance = 1000; // private으로 해야 동기화가 의미가 있다.

    public int getBalance() {
        return balance;
    }

    public synchronized void withdraw(int money){ // synchronized로 메서
        if(balance >= money) {
            try { Thread.sleep(1000);} catch(InterruptedException e) {
            balance -= money;
        }
    } // withdraw
}

class RunnableEx22 implements Runnable {
    Account2 acc = new Account2();

    public void run() {
        while(acc.getBalance() > 0) {
            // 100, 200, 300중의 한 값을 임으로 선택해서 출금(withdraw)
            int money = (int)(Math.random() * 3 + 1) * 100;
            acc.withdraw(money);
            System.out.println("balance:"+acc.getBalance());
        }
    } // run()
}
```

▶synchronized없을 때



▶synchronized있을 때




```
class ThreadEx22 {
    public static void main(String args[]) {
        Runnable r = new RunnableEx22();
        new Thread(r).start();
        new Thread(r).start();
    }
}
```

1.14 쓰레드의 동기화 – wait(), notify(), notifyAll()

- 동기화의 효율을 높이기 위해 wait(), notify()를 사용.
- Object클래스에 정의되어 있으며, 동기화 블록 내에서만 사용할 수 있다.

- wait() – 객체의 lock을 풀고 쓰레드를 해당 객체의 waiting pool에 넣는다.
- notify() – waiting pool에서 대기중인 쓰레드 중의 하나를 깨운다.
- notifyAll() – waiting pool에서 대기중인 모든 쓰레드를 깨운다.

```
class Account {
    int balance = 1000;

    public synchronized void withdraw(int money){
        while(balance < money) {
            try {
                wait(); // 대기 - 락을 풀고 기다린다. 통지를 받으면 락을 재획득(ReEntrance)
            } catch(InterruptedException e) {}
        }

        balance -= money;
    } // withdraw

    public synchronized void deposit(int money) {
        balance += money;
        notify(); // 통지 - 대기중인 쓰레드 중 하나에게 알림.
    }
}
```

1.14 쓰레드의 동기화(Ex1) – 생산자와 소비자 문제

- 요리사는 Table에 음식을 추가. 손님은 Table의 음식을 소비
- 요리사와 손님이 같은 객체(Table)을 공유하므로 동기화가 필요

Table

```
private ArrayList dishes
                    = new ArrayList();

public void add(String dish) {
    // 테이블이 가득찼으면, 음식을 추가안함
    if(dishes.size() >= MAX_FOOD)
        return;
    dishes.add(dish);
    System.out.println("Dishes:"
        + dishes.toString());
}

public boolean remove(String dishName)
{
  // 지정된 요리와 일치하는 요리를 테이블에서 제거한다.
  for(int i=0; i<dishes.size();i++)
    if(dishName.equals(dishes.get(i)))
    {
        dishes.remove(i);
        return true;
    }
    return false;
}
```

Cook

```
public void run() {
    while(true) {
        // 임의의 요리를 하나 선택해서 table에 추가한다.
        int idx = (int)(Math.random()*table.dishNum());
        table.add(table.dishNames[idx]);
        try { Thread.sleep(1);} catch(InterruptedException e) {}
    } // while
}
```

Customer

```
public void run() {
    while(true) {
        try { Thread.sleep(10);} catch(InterruptedException e) {}
        String name = Thread.currentThread().getName();

        if(eatFood())
            System.out.println(name + " ate a " + food);
        else
            System.out.println(name + " failed to eat. :(");
    } // while
}

boolean eatFood() { return table.remove(food); }
```

main( )

```
Table table = new Table(); // 여러 쓰레드가 공유하는 객체

new Thread(new Cook(table), "COOK1").start();
new Thread(new Customer(table, "donut"), "CUST1").start();
new Thread(new Customer(table, "burger"), "CUST2").start();
```

1.14 쓰레드의 동기화(Ex1) – 실행결과

[예외1] 요리사가 Table에 요리를 추가하는 과정에 손님이 요리를 먹음
[예외2] 하나 남은 요리를 손님2가 먹으려하는데, 손님1이 먹음.

[실행결과]

```
Dishes:[donut]
Dishes:[donut, burger]
Dishes:[donut, burger, donut]
Dishes:[donut, burger, donut, donut]
CUST1 ate a donut
CUST2 ate a burger
Dishes:[burger, donut, donut]
Dishes:[burger, donut, donut, burger]
Dishes:[burger, donut, donut, burger, donut]
Dishes:[burger, donut, donut, burger, donut, donut]
CUST2 ate a burger
CUST1 ate a donut
Exception in thread "COOK1" java.util.ConcurrentModificationException
      at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
      at java.util.ArrayList$Itr.next(ArrayList.java:851)
      at java.util.AbstractCollection.toString(AbstractCollection.java:461)
      at Table.add(ThreadWaitEx1.java:49)
      at Cook.run(ThreadWaitEx1.java:35)
      at java.lang.Thread.run(Thread.java:745)
CUST1 ate a donut
CUST2 ate a burger
CUST1 ate a donut
CUST2 ate a burger
CUST1 ate a donut
Exception in thread "CUST2" java.lang.IndexOutOfBoundsException: Index: 0,
  Size: 0
      at java.util.ArrayList.rangeCheck(ArrayList.java:653)
      at java.util.ArrayList.get(ArrayList.java:429)
      at Table.remove(ThreadWaitEx1.java:54)
      at Customer.eatFood(ThreadWaitEx1.java:24)
      at Customer.run(ThreadWaitEx1.java:17)
      at java.lang.Thread.run(Thread.java:745)
CUST1 failed to eat. :(
CUST1 failed to eat. :(
```

1.14 쓰레드의 동기화(Ex2) – 생산자와 소비자 문제

[문제점] Table을 여러 쓰레드가 공유하기 때문에 작업 중에 끼어들기 발생

[해결책] Table의 add()와 remove()를 synchronized로 동기화

Table

```
private ArrayList dishes
                    = new ArrayList();

public void add(String dish) {
    // 테이블이 가득찼으면, 음식을 추가안함
    if(dishes.size() >= MAX_FOOD)
        return;
    dishes.add(dish);
    System.out.println("Dishes:"
        + dishes.toString());
}

public boolean remove(String dishName)
{
  // 지정된 요리와 일치하는 요리를 테이블에서 제거한다.
  for(int i=0; i<dishes.size();i++)
    if(dishName.equals(dishes.get(i)))
    {
        dishes.remove(i);
        return true;
    }
    return false;
}
```

동기화된 Table

```
public synchronized void add(String dish) {
    if(dishes.size() >= MAX_FOOD)
        return;

    dishes.add(dish);
    System.out.println("Dishes:" + dishes.toString());
}

public boolean remove(String dishName) {
    synchronized(this) {
        while(dishes.size()==0) {
            String name = Thread.currentThread().getName();
            System.out.println(name+" is waiting.");
            try {
                Thread.sleep(500);
            } catch(InterruptedException e) {}
        }
        for(int i=0; i<dishes.size();i++)
            if(dishName.equals(dishes.get(i))) {
                dishes.remove(i);
                return true;
            }
    } // synchronized
    return false;
}
```

1.14 쓰레드의 동기화(Ex2) – 실행결과

[문제] 예외는 발생하지 않지만, 손님(CUST2)이 Table에 lock건 상태를 지속
요리사가 Table의 lock을 얻을 수 없어서 음식을 추가하지 못함

[실행결과]

```
Dishes:[burger]
CUST2 ate a burger
CUST1 failed to eat. :(   ← donut이 없어서 먹지 못했다.
CUST2 is waiting.   ← 음식이 없어서 테이블에 lock을 건 채로 계속 기다리고 있다.
CUST2 is waiting.
CUST2 is waiting.
CUST2 is waiting.
CUST2 is waiting.
CUST2 is waiting.
CUST2 is waiting.
CUST2 is waiting.
CUST2 is waiting.
CUST2 is waiting.
CUST2 is waiting.
```

1.14 쓰레드의 동기화(Ex3) – 생산자와 소비자 문제

[문제점] 음식이 없을 때, 손님이 Table의 lock을 쥐고 안놓는다.
요리사가 lock을 얻지못해서 Table에 음식을 추가할 수 없다.

[해결책] 음식이 없을 때, wait()으로 손님이 lock을 풀고 기다리게하자.
요리사가 음식을 추가하면, notify()로 손님에게 알리자.(손님이 lock을 재획득)

```
public synchronized void add(String dish) {
    while(dishes.size() >= MAX_FOOD) {
        String name = Thread.currentThread().getName();
        System.out.println(name+" is waiting.");
        try {
            wait(); // COOK쓰레드를 기다리게 한다.
            Thread.sleep(500);
        } catch(InterruptedException e) {}
    }
    dishes.add(dish);
    notify(); // 기다리고 있는 CUST를 깨우기 위함.
    System.out.println("Dishes:" + dishes.toString());
}
```

```
public void remove(String dishName) {
    synchronized(this) {
        String name = Thread.currentThread().getName();
        while(dishes.size()==0) {
            System.out.println(name+" is waiting.");
            try {
                wait(); // CUST쓰레드를 기다리게 한다.
                Thread.sleep(500);
            } catch(InterruptedException e) {}
        }

        while(true) {
            for(int i=0; i<dishes.size();i++) {
                if(dishName.equals(dishes.get(i))) {
                    dishes.remove(i);
                    notify(); // 잠자고 있는 COOK을 깨우기 위함
                    return;
                }
            } // for문의 끝

            try {
                System.out.println(name+" is waiting.");
                wait(); // 원하는 음식이 없는 CUST쓰레드를 기다리게 한다.
                Thread.sleep(500);
            } catch(InterruptedException e) {}
        } // while(true)
    } // synchronized
}
```

1.14 쓰레드의 동기화(Ex3) – 실행결과

- 전과 달리 한 쓰레드가 lock을 오래 쥐는 일이 없어짐. 효율적이 됨!!!

[실행결과]

```
Dishes:[donut]
Dishes:[donut, burger]
... 중간 생략...
Dishes:[donut, donut, donut, donut, donut, donut]
COOK1 is waiting.
CUST2 is waiting.
CUST1 ate a donut
Dishes:[donut, donut, donut, donut, donut, donut]
CUST2 is waiting.  ← 원하는 음식이 없어서 손님이 기다리고 있다.
COOK1 is waiting.  ← 테이블이 가득차서 요리사가 기다리고 있다.
CUST1 ate a donut  ← 테이블의 음식이 소비되어 notify()가 호출된다.
CUST2 is waiting.  ← 요리사가 아닌 손님이 통지를 받고, 원하는 음식이 없어서 다시 기다린다.
CUST1 ate a donut ← 테이블의 음식이 소비되어 notify()가 호출된다.
Dishes:[donut, donut, donut, donut, donut]  ← 이번엔 요리사가 통지받고 음식추가
CUST2 is waiting. ← 음식추가 통지를 받았으나 원하는 음식이 없어서 다시 기다린다.
Dishes:[donut, donut, donut, donut, donut, burger] ← 요리사가 음식추가(활동 중)
CUST1 ate a donut
CUST2 ate a burger ← 음식추가 통지를 받고, 원하는 음식을 소비(활동 중)
Dishes:[donut, donut, donut, donut, donut]
Dishes:[donut, donut, donut, donut, donut, burger]
COOK1 is waiting.
CUST1 ate a donut
```

1.15 Lock과 Condition을 이용한 동기화(1)

- java.util.concurrent.locks패키지를 이용한 동기화(JDK1.5)

ReentrantLock	재진입이 가능한 lock. 가장 일반적인 배타 lock
ReentrantReadWriteLock	읽기에는 공유적이고, 쓰기에는 배타적인 lock
StampedLock	ReentrantReadWriteLock에 낙관적인 lock의 가능을 추가

[참고] StampedLock은 JDK1.8부터 추가되었으며, 다른 lock과 달리 Lock인터페이스를 구현하지 않았다.

- 낙관적인 잠금(Optimistic Lock) : 일단 무조건 저지르고 나중에 확인

```
int getBalance() {
    long stamp = lock.tryOptimisticRead(); // 낙관적 읽기 lock을 건다.

    int curBalance = this.balance;         // 공유 데이터인 balance를 읽어온다.

    if(lock.validate(stamp)) {    // 쓰기 lock에 의해 낙관적 읽기 lock이 풀렸는지 확인
        stamp = lock.readLock();  // lock이 풀렸으면, 읽기 lock을 얻으려고 기다린다.

        try {
            curBalance = this.balance; // 공유 데이터를 다시 읽어온다.
        } finally {
            lock.unlockRead(stamp);    // 읽기 lock을 푼다.
        }
    }

    return curBalance; // 낙관적 읽기 lock이 풀리지 않았으면 곧바로 읽어온 값을 반환
}
```

1.15 Lock과 Condition을 이용한 동기화(2)

- ReentrantLock을 이용한 동기화

```
ReentrantLock()
ReentrantLock(boolean fair)
```

- synchronized대신 lock()과 unlock()을 사용

```
void lock()          lock을 잠근다.
void unlock()        lock을 해지한다.
boolean isLocked()   lock이 잠겼는지 확인한다.
```

```
synchronized(lock) {
   // 임계 영역
}
```

```
lock.lock();
// 임계 영역
lock.unlock();
```

```
lock.lock();  // ReentrantLock lock = new ReentrantLock();
try {
    // 임계 영역
} finally {
    lock.unlock();
}
```

1.15 Lock과 Condition을 이용한 동기화(3)

- ReentrantLock과 Condition으로 쓰레드를 구분해서 wait() & notify()

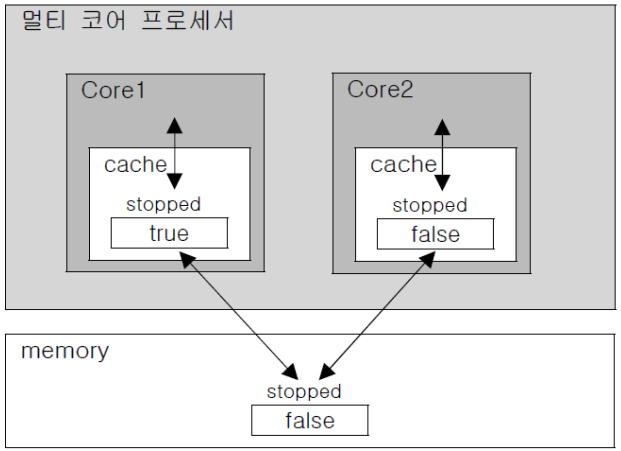
```
public void add(String dish) {
    lock.lock();

    try {
        while(dishes.size() >= MAX_FOOD) {
            String name = Thread.currentThread().getName();
            System.out.println(name+" is waiting.");
            try {
                forCook.await(); // wait(); COOK쓰레드를 기다리게 한다.
                Thread.sleep(500);
            } catch(InterruptedException e) {}
        }
        dishes.add(dish);
        forCust.signal(); // notify(); 기다리고 있는 CUST를 깨우기 위함.
        System.out.println("Dishes:" + dishes.toString());
    } finally {
        lock.unlock();
    }
}
```

- ReentrantLock과 Condition의 생성방법

```
private ReentrantLock lock  = new ReentrantLock(); // lock을 생성

private Condition forCook = lock.newCondition();  // lock으로 condition을 생성
private Condition forCust = lock.newCondition();
```

1.16 volatile – cache와 메모리간의 불일치 해소

- 성능 향상을 위해 변수의 값을 core의 cache에 저장해 놓고 작업
- 여러 쓰레드가 공유하는 변수에는 volatile을 붙여야 항상 메모리에서 읽어옴



[그림13-11] 멀티 코어 프로세서의 캐시(cache)와 메모리간의 통신

```
boolean suspended = false;
boolean stopped   = false;
```

→

```
volatile boolean suspended = false;
volatile boolean stopped   = false;
```

```
public void stop() {
   stopped = true;
}
```

→

```
public synchronized void stop() {
   stopped = true;
}
```

6.1 fork & join 프레임웍

- 작업을 여러 쓰레드가 나눠서 처리하는 것을 쉽게 해준다.(JDK1.7)
- RecursiveAction 또는 RecursiveTask를 상속받아서 구현

RecursiveAction	반환값이 없는 작업을 구현할 때 사용
RecursiveTask	반환값이 있는 작업을 구현할 때 사용

```
public abstract class RecursiveAction extends ForkJoinTask<Void> {
        ...
    protected abstract void compute(); // 상속을 통해 이 메서드를 구현해야 한다.
        ...
}
```

```
public abstract class Recurs
        ...
    V result;
    protected abstract V com
        ...
}
```

```
class SumTask extends RecursiveTask<Long> {
    long from, to;

    SumTask(long from, long to) {
        this.from = from;
        this.to   = to;
    }

    public Long compute() {
        // 처리할 작업을 수행하기 위한 문장을 넣는다.
    }
}
```

6.2 compute()의 구현

- 수행할 작업과 작업을 어떻게 나눌 것인지를 정해줘야 한다.
- fork()로 나눈 작업을 큐에 넣고, compute()를 재귀호출한다.

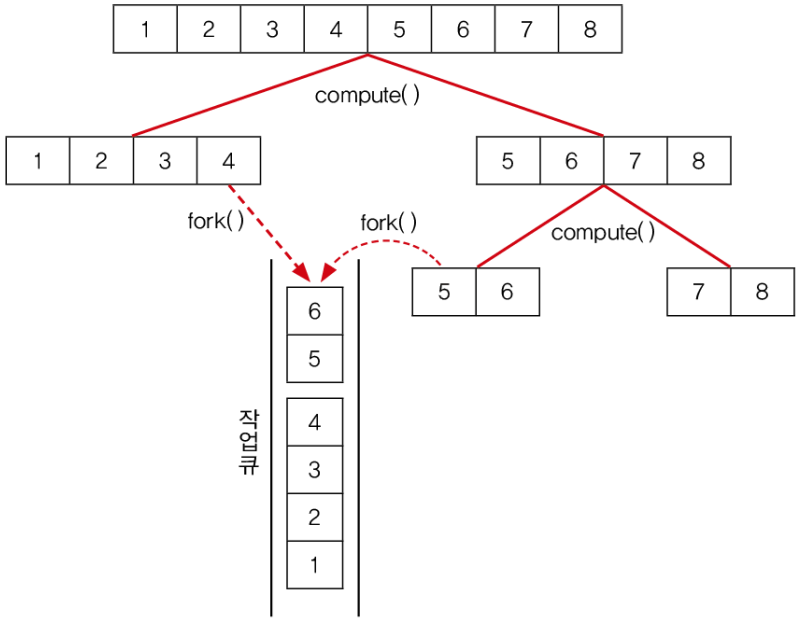
```
public Long compute() {
    long size = to - from + 1;  // from ≤ i

    if(size <= 5)       // 더할 숫자가 5개 이하면
        return sum(); // 숫자의 합을 반환.

    // 범위를 반으로 나눠서 두 개의 작업을 생성
    long half = (from+to)/2;

    SumTask leftSum  = new SumTask(from, half);
    SumTask rightSum = new SumTask(half+1, to);

    leftSum.fork(); // 작업(leftSum)을 작업 큐에 넣는다.

    return rightSum.compute() + leftSum.join();
}
```

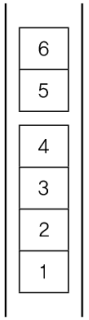
```
long sum() {
    long tmp = 0L;

    for(long i=from;i<=to;i++)
        tmp += i;

    return tmp;
}
```

```
ForkJoinPool pool = new ForkJoinPool(); // 쓰레드풀을 생성
SumTask task = new SumTask(from, to);   // 수행할 작업을 생성

Long result = pool.invoke(task); // invoke()를 호출해서 작업을 시작
```

6.3 작업 훔치기(work stealing)

- 작업을 나눠서 다른 쓰레드의 작업 큐에 넣는 것





6.4 fork()와 join()

- compute()는 작업을 나누고, fork()는 작업을 큐에 넣는다.(반복)
- join()으로 작업의 결과를 합친다.(반복)

fork( )	해당 작업을 쓰레드 풀의 작업 큐에 넣는다. 비동기 메서드
join( )	해당 작업의 수행이 끝날 때까지 기다렸다가, 수행이 끝나면 그 결과를 반환한다. 동기 메서드

```
public Long compute() {
    long size = to - from + 1;    // from ≤ i ≤ to

    if(size <= 5)       // 더할 숫자가 5개 이하면
        return sum(); // 숫자의 합을 반환

    long half = (from+to)/2;

    // 범위를 반으로 나눠서 두 개의 작업을 생성
    SumTask leftSum  = new SumTask(from, half);
    SumTask rightSum = new SumTask(half+1, to);
    leftSum.fork();  // 비동기 메서드. 호출 후 결과를 기다리지 않는다.

    // 동기 메서드. 호출결과를 기다린다.
    return rightSum.compute()+leftSum.join();
}
```