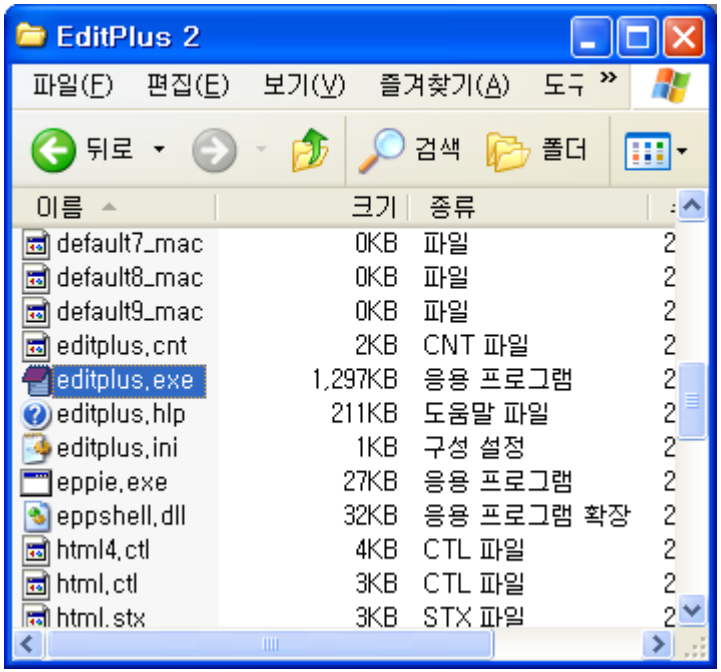
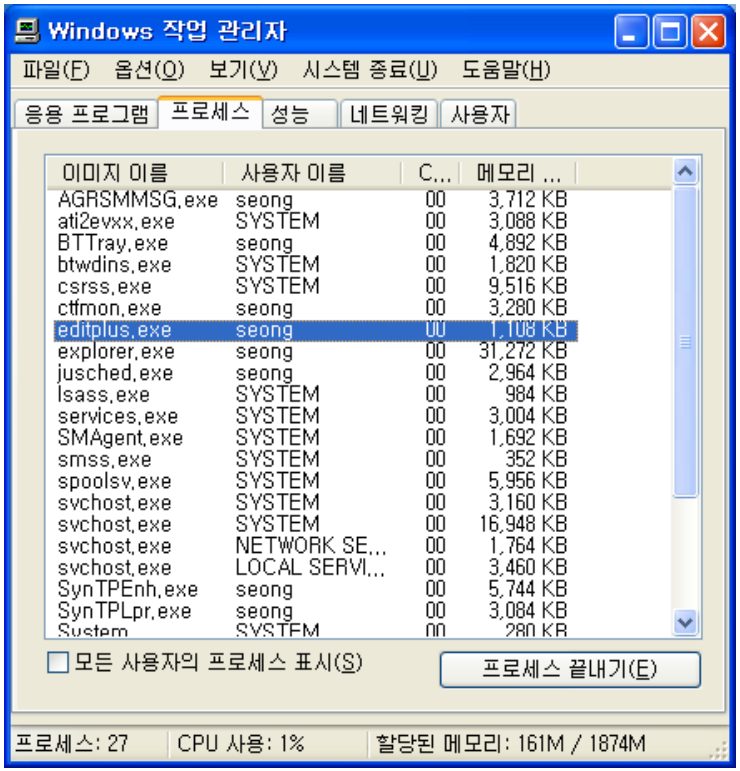
스레드(Thread) I

프로세스와 스레드(process & thread)

▶ 프로그램 : 실행 가능한 파일(HDD)



▶ 프로세스 : 실행 중인 프로그램(메모리)



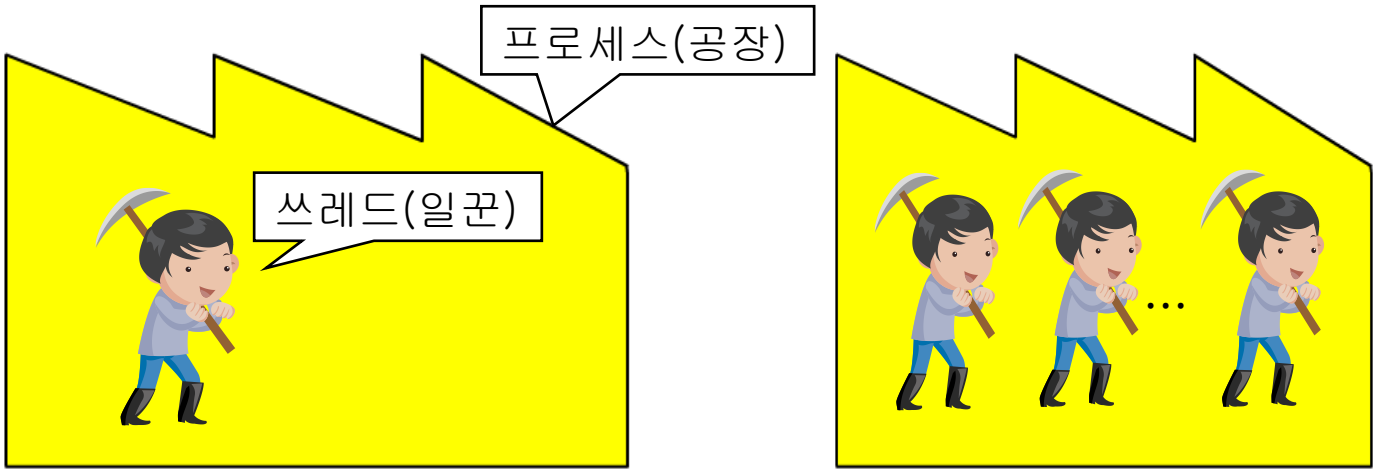
프로세스와 스레드(process & thread)

- **프로세스** : 실행 중인 프로그램. 자원(resources)과 쓰레드로 구성
- **쓰레드**: 프로세스 내에서 실제 작업을 수행하는 것
- 모든 프로세스는 하나 이상의 쓰레드를 가지고 있다.

프로세스 : 쓰레드 = 공장 : 일꾼

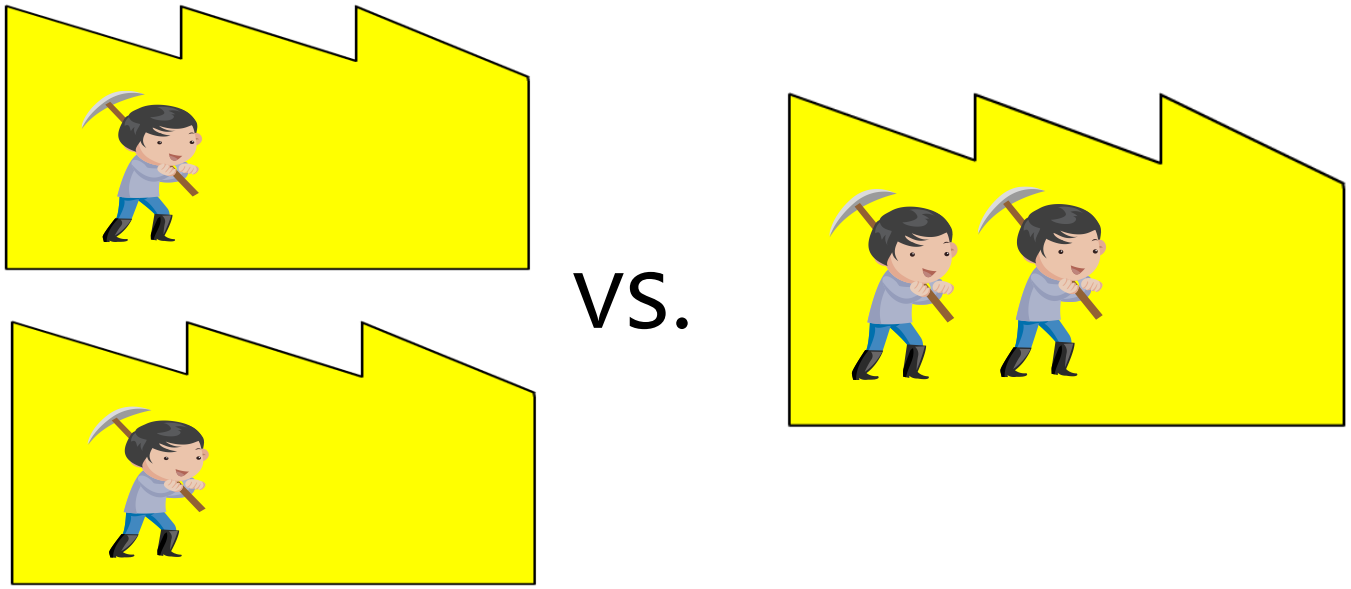
▶ 싱글 쓰레드 프로세스
= 자원+쓰레드

▶ 멀티 쓰레드 프로세스
= 자원+쓰레드+쓰레드+…+쓰레드



멀티 프로세스 vs. 멀티 스레드

- “하나의 새로운 프로세스를 생성하는 것보다
  하나의 새로운 쓰레드를 생성하는 것이 더 적은 비용이 든다.”
- 2개의 프로세스 1 쓰레드 vs. 1개의 프로세스 2 쓰레드



멀티 스레드의 장단점

- 장점: “여러 모로 좋다.”
  - 자원을 보다 효율적으로 사용할 수 있다.
  - 사용자에 대한 응답성이 향상된다.
  - 작업이 분리되어 코드가 간결해 진다.

- 단점: “프로그래밍할 때 고려해야 할 사항들이 많다.”
  - 동기화(synchronization)에 주의해야 한다.
  - 교착상태(dead-lock)가 발생하지 않도록 주의해야 한다.
  - 각 쓰레드가 효율적으로 고르게 실행될 수 있게 해야 한다.

스레드의 구현과 실행1

```
public class MyThread01 extends Thread{
   @Override
   public void run() {
      for(int i=0; i<100; i++)
           System.out.println("MyThread01:"+i);
      System.out.println("==========>> MyThread01 업무종료");
   }
}

public class MyThread02 implements Runnable{
   @Override
   public void run() {
      for(int i=0; i<100; i++)
           System.out.println("MyThread02:"+i);
      System.out.println("==========>> MyThread02 업무종료");
   }
}
```

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

스레드의 구현과 실행2

```
public class MyThreadTest {
   public static void main(String[] args) {
      Thread t1 = new MyThread01();
      Thread t2 = new Thread(new MyThread02());
      Thread t3 = new Thread(){
            public void run() {
                for(int i=0; i<100; i++)
                    System.out.println("MyThread03:"+i);
                System.out.println("==========>> MyThread03 업무종료
                ");
            };
      };
      Thread t4 = new Thread(new Runnable(){
            public void run() {
                for(int i=0; i<100; i++)
                    System.out.println("MyThread04:"+i);
                System.out.println("==========>> MyThread04 업무종료
                ");
            };
      });
      t1.start();t2.start();t3.start();t4.start();
   }
}
```

스레드 이름

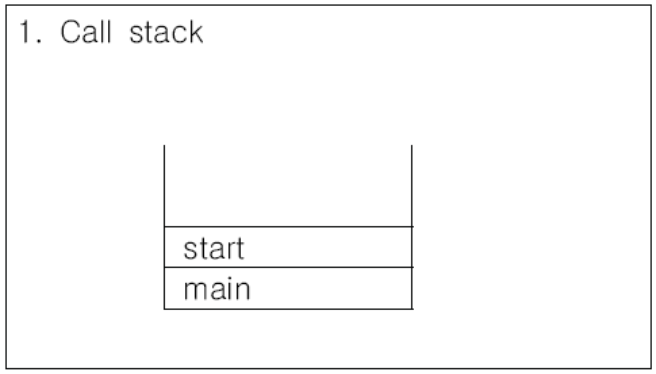
- setName("스레드 이름"): 스레드 이름 지정
- getName( ): 스레드 이름 반환
- Thread.currentThread( ): 현재 스레드의 주소값 반환

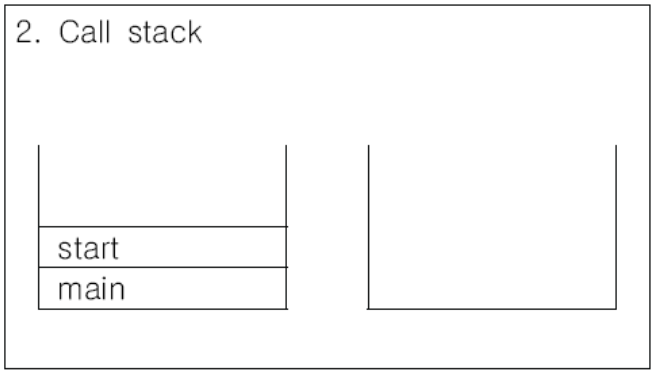
```
public class ThreadNameEx {
    public static void main(String[] args) {
        Thread t1 = Thread.currentThread();
        System.out.println("현재 스레드 이름: "+ t1.getName());

        Thread t2 = new Thread() {
            @Override
            public void run() {
                setName("MyThread");
            }
        };
        t2.start();

        try{Thread.sleep(1000);}catch(InterruptedException e){}
        System.out.println("새로 만든 스레드 이름: "+t2.getName());
    }
}
```

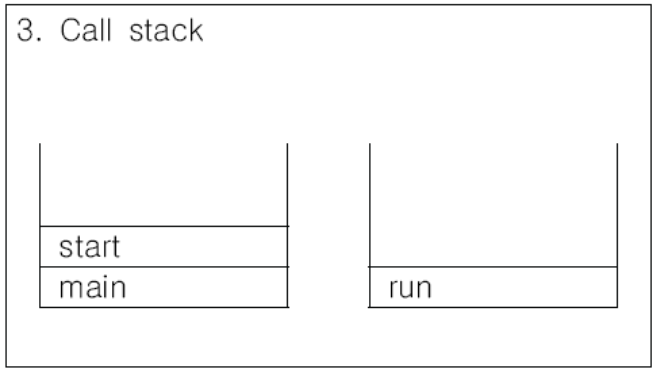
start() & run()

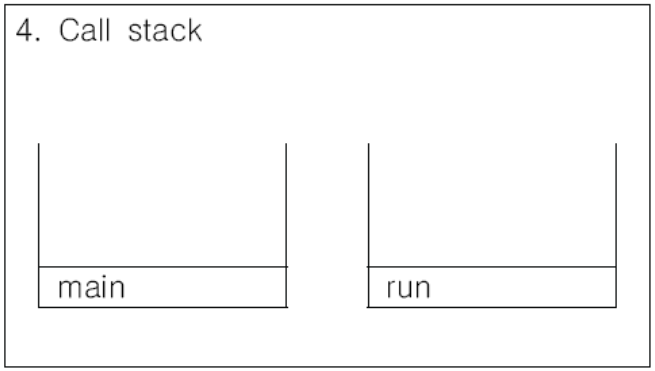
```
class ThreadTest {
    public static void main(String args[]){
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

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





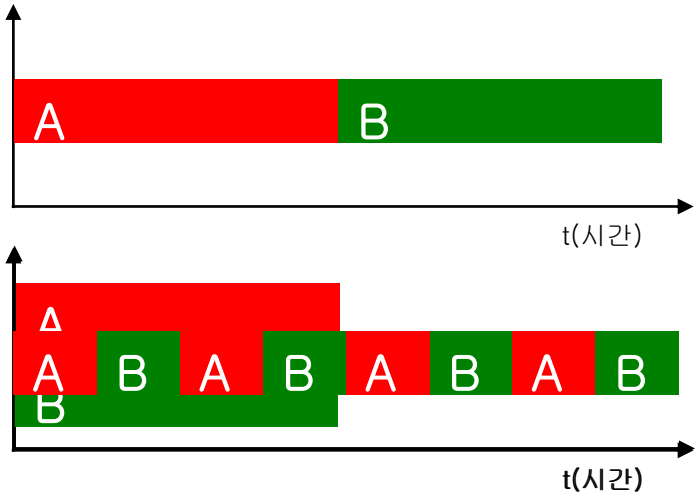




싱글 스레드 vs. 멀티 스레드

▶ 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
}
```



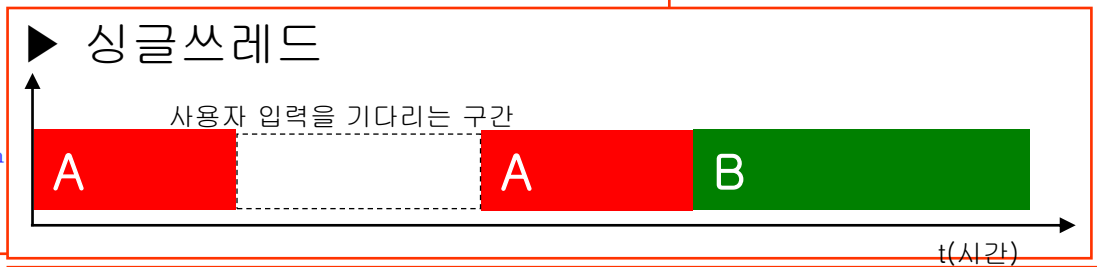
▶ 멀티쓰레드

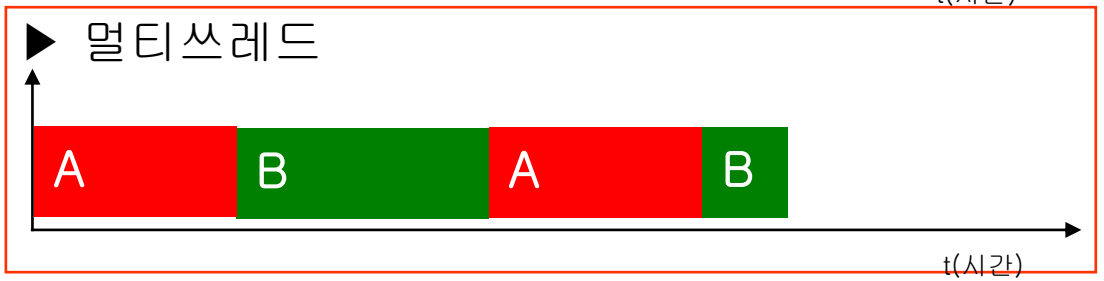
```
class ThreadTest {
    public static void main(String args[]){
        MyThread1 th1 = new MyThread1();
        MyThread2 th2 = new MyThread2();
        th1.start();
        th2.start();
    }
}

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()
}
```

싱글 스레드 vs. 멀티 스레드

```
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); } catch(Exception e ) {}
        }
    } // main
}
```

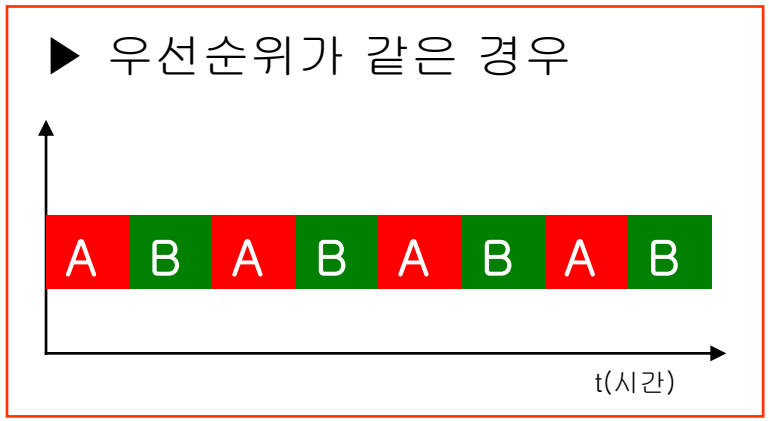
```
class ThreadEx7 {
    public static void main(String[] args) {
        ThreadEx7_1 th1 = new ThreadEx7_1();
        th1.start();

        String input = JOptionPane.showInputDialog("아무 값이나 입력하세요.");
        System.out.println("입력하신 값은 " + input + "입니다.");
    }
}

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

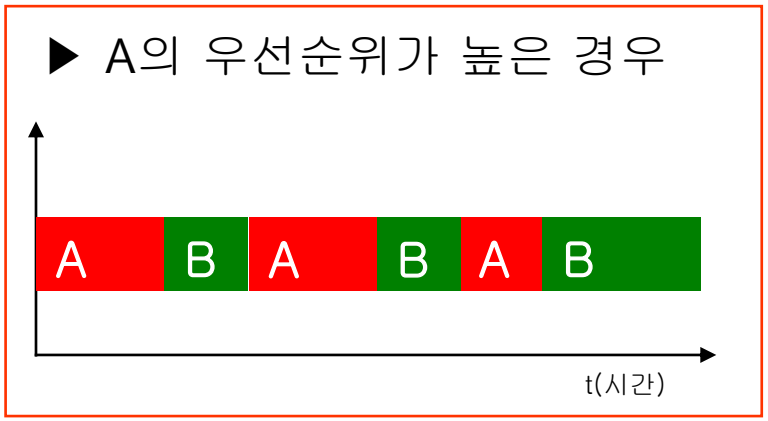




스레드의 우선순위(priority of thread)

- 스레드 스케쥴링: 우선순위(Priority), 순환 할당(Round-Robin)
- 작업의 중요도에 따라 스레드의 우선순위를 다르게 하여 우선순위가 높은 스레드가 더 많은 작업시간을 갖도록 함
- 우선순위 값: 1(MIN_PRIORITY) – 10 (MAX_PRIORITY) *기본값: 5 (NORM_PRIORITY)
- setPriority(우선순위 상수)
- 코어 숫자보다 많은 스레드의 개수에서 의미 있음





스레드의 동기화 - synchronized

- 멀티 스레드 프로그램에서 스레드들이 하나의 객체를 공유해서 작업해야 하는 경우 필요함
- 공유 객체를 스레드들이 차례대로 사용하도록 함
- 임계영역(critical section, 하나의 스레드만 실행할 수 있는 코드 영역)을 지정함
- 임계영역 지정방법: 동기화 메소드, 동기화 블록
- 동기화 메소드: 메소드 선언부의 수정자 부분에 synchronized 수정자 붙임. 메소드 전체가 임계영역이 되며 스레드가 해당 메소드를 실행하는 즉시 공유객체에 잠금(lock) 발생
- 동기화 블록: synchronized (공유객체) { ...임계영역...}. 메소드의 일부 영역만 임계영역으로 만듦
- 동기화 메소드와 블록이 여러 개 있을 경우, 스레드가 이들 중 하나를 실행하면 다른 스레드는 모든 동기화 메소드와 블록을 실행할 수 없음

스레드의 동기화 예제: BankAccount.java

```
public class BankAccount {
    private int balance=1000;

    public synchronized void withdraw(int money) {
        System.out.println(Thread.currentThread().getName()+"출금전 은행잔고: "+balance);
            if(balance >= money) {
                try {Thread.sleep(2000);} catch (InterruptedException e) {}
                balance -= money;
                System.out.println(Thread.currentThread().getName()+" 출금액: "+money);
                System.out.println("출금후 은행잔고: "+balance);
            }else
                System.out.println("은행잔고가 부족합니다");
    }

    public int getBalance() {  return balance;  }
}
```

스레드의 동기화 예제: BankCustomer01.java

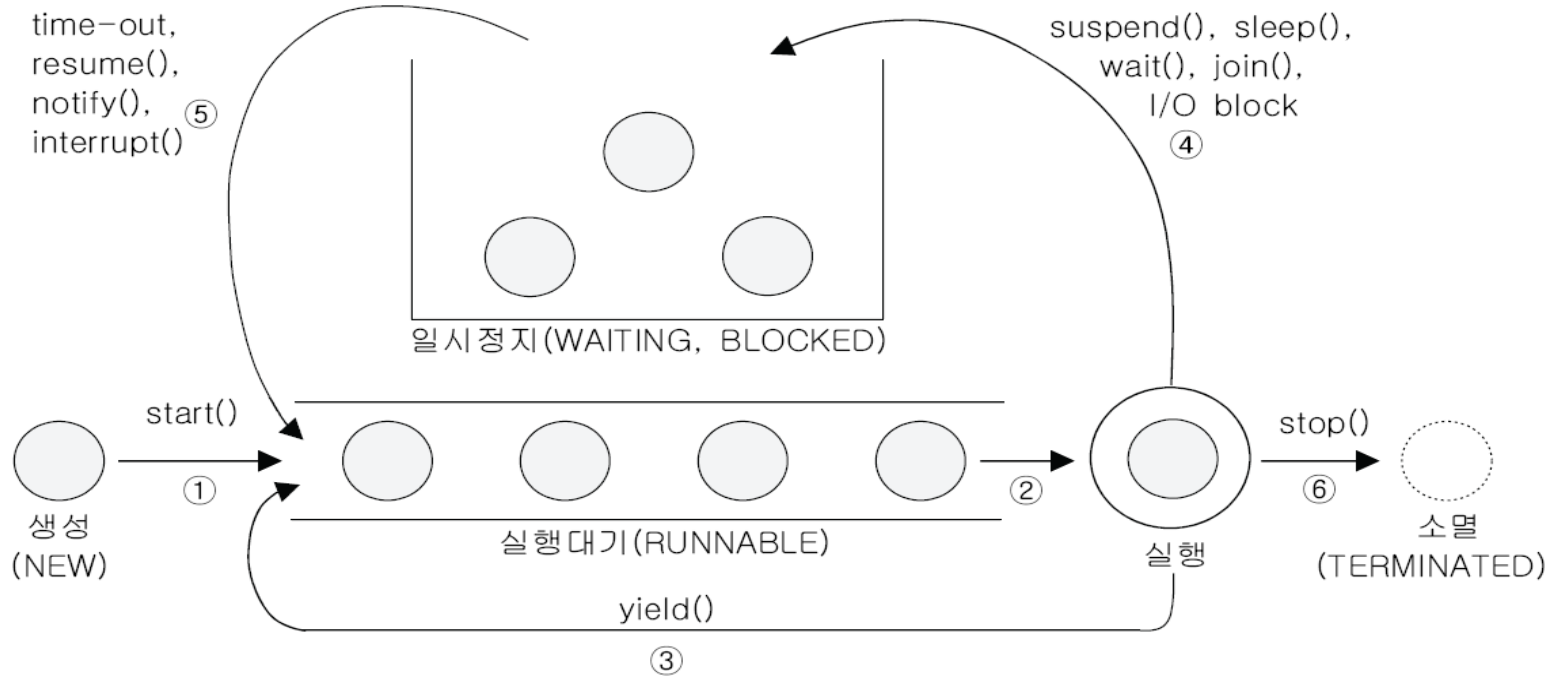
```
public class BankCustomer01 extends Thread {
    private BankAccount account;

    public BankCustomer01(BankAccount account) {
        setName("BankCustomer01");
        this.account = account;
    }

    @Override
    public void run() {
        for(int i=1; i<4; i++) {
            account.withdraw(i*200);
        }
    }
}
```

스레드의 동기화 예제: BankAccountTest.java

```
public class BankAccountTest {

    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        BankCustomer01 customer1 = new BankCustomer01(account);
        BankCustomer02 customer2 = new BankCustomer02(account);
        customer1.start();
        customer2.start();
    }
}
```

스레드의 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) 일시정지상태. TIMED_WAITING은 일시정지시간이 지정된 경우를 의미한다.
TERMINATED	쓰레드의 작업이 종료된 상태



스레드의 실행제어

생성자 / 메서드	설 명
void interrupt()	sleep()이나 join()에 의해 일시정지상태인 쓰레드를 실행대기상태로 만든다. 해당 쓰레드에서는 InterruptedException이 발생함으로써 일시정지상태를 벗어나게 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 resume()	suspend()에 의해 일시정지상태에 있는 쓰레드를 실행대기상태로 만든다.
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 stop()	쓰레드를 즉시 종료시킨다. 교착상태(dead-lock)에 빠지기 쉽기 때문에 deprecated되었다.
void suspend()	쓰레드를 일시정지시킨다. resume()을 호출하면 다시 실행대기상태가 된다.
static void yield()	실행 중에 다른 쓰레드에게 양보(yield)하고 실행대기상태가 된다.

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

다른 스레드의 종료를 기다림: join()

- **다른 스레드가 종료될 때까지 기다렸다가 실행해야 하는 경우 사용**

  (예) 계산작업을 하는 스레드가 모든 계산 작업을 마쳤을 때 계산 결과값을 받아 작업을 해야 하는 경우

```
public class SumThread extends Thread {
   private int sum;
   public int getSum() { return sum; }
   public void setSum(int sum) { this.sum = sum; }
   @Override
   public void run() {for(int i=1; i<101; i++) sum += i; }
}
public class JoinEx {
   public static void main(String[] args) {
       SumThread sumThread = new SumThread();
       sumThread.start();
       try {
            sumThread.join();
       } catch (InterruptedException e) {}
       System.out.println("1-100 합: "+ sumThread.getSum());
} }
```

상호 협력 동기화 – wait(), notify(), notifyAll()

- 공유객체를 이용해서 번갈아 가면서 작업하는 경우
- Object클래스에 정의되어 있으며, 동기화 메소드나 블록 내에서만 사용 가능
- wait(): 공유객체 작업을 하는 스레드가 공유객체의 잠금을 풀어주고 자신을 일시정지 상태로 만듦(waiting pool로 들어감)
- notify(): 일시정지 상태에 있는 쓰레드를 실행 대기 상태로 만듦
- notifyAll(): 일시정지 상태에 있는 모든 쓰레드를 실행 대기 상태로 만듦

(예제) 교재 pp.611-613 생산자와 소비자 예제

스레드의 안전한 종료1

○ stop 플래그를 이용한 방법

```
public class StopFlagEx {
   public static void main(String[] args) {
      PrintThread pt = new PrintThread();
      pt.start();
      try { Thread.sleep(1000); } catch (InterruptedException e) {}
      pt.setStop(true);
   }}

public class PrintThread extends Thread {
private boolean stop;
   public void setStop(boolean stop) {this.stop=stop;}
      @Override
      public void run() {
      while(!stop) { System.out.println("실행중"); }
      System.out.println("자원 정리");
      System.out.println("실행 종료");
   }}
```

스레드의 안전한 종료2

○ **interrupt( ) 메소드를 이용한 방법:** InterruptedException예외 발생시킴

```
public class InterruptEx {
   public static void main(String[] args) {
      PrintThread2 pt = new PrintThread2();
      pt.start();
      try { Thread.sleep(1000); } catch (InterruptedException e) {}
      pt.interrupt();
   }}

public class PrintThread2 extends Thread {
private boolean stop;
   public void setStop(boolean stop) {this.stop=stop;}
      @Override
      public void run() {
      try{
         while(true) { System.out.println("실행중"); Thread.sleep(1); }
      } catch (InterruptedException e) {}
      System.out.println("자원 정리");  System.out.println("실행 종료");
   }}
```

스레드의 안전한 종료3

○ **interrupt( ) 메소드를 이용한 방법:** interrupt()메소드 호출 감지

```
public class InterruptEx {
   public static void main(String[] args) {
      PrintThread2 pt = new PrintThread2();
      pt.start();
      try { Thread.sleep(1000); } catch (InterruptedException e) {}
      pt.interrupt();
   }}

public class PrintThread2 extends Thread {
private boolean stop;
   public void setStop(boolean stop) {this.stop=stop;}
      @Override
      public void run() {
      while(true) { System.out.println("실행중");
                    if(Thread.interrupted()) break; }
      System.out.println("자원 정리");  System.out.println("실행 종료");
   }}
```

데몬 스레드(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.
- setDaemon(boolean on)은 반드시 start()를 호출하기 전에 실행되어야 한다. 그렇지 않으면 IllegalThreadStateException이 발생한다.

  (예제) 교재 p.619

연습문제

가족 구성원이 3명(구성원1-3)인 가정에 한 개의 화장실이 있습니다. 한 사람이 화장실을 사용하고 있으면 다른 사람이 화장실을 사용할 수 없도록 하고 "OOO가 사용 중"임을 출력해 주고 사용 후에는 "비어 있음"을 출력하는 프로그램을 다중 스레드를 이용해서 작성하시오. 단, 화장실은 각 사람이 한 번만 사용할 수 있도록 합니다.

[출력 예]

- 화장실 사용하고 있는 경우 : OOO가 사용 중
- 화장실이 비어있는 경우 : 비어 있음


수고했습니다