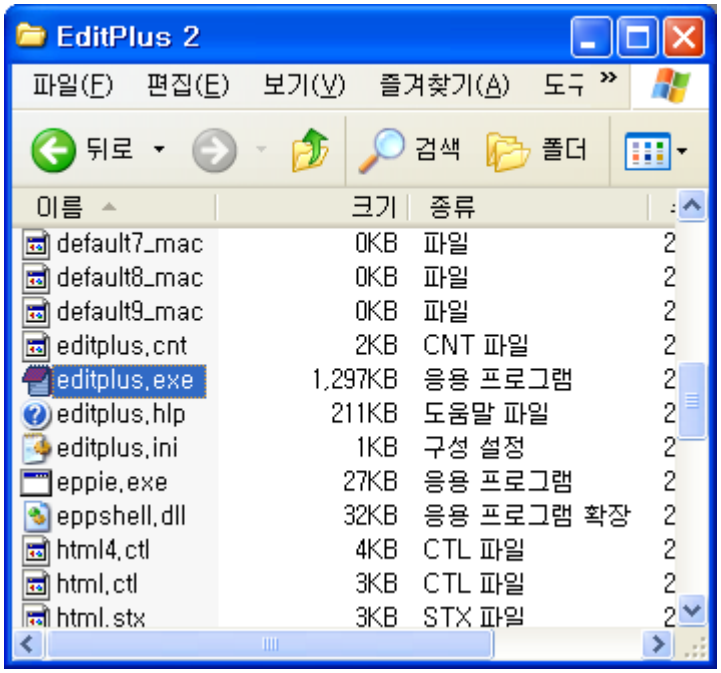
JAVA

- 쓰레드 (thread)

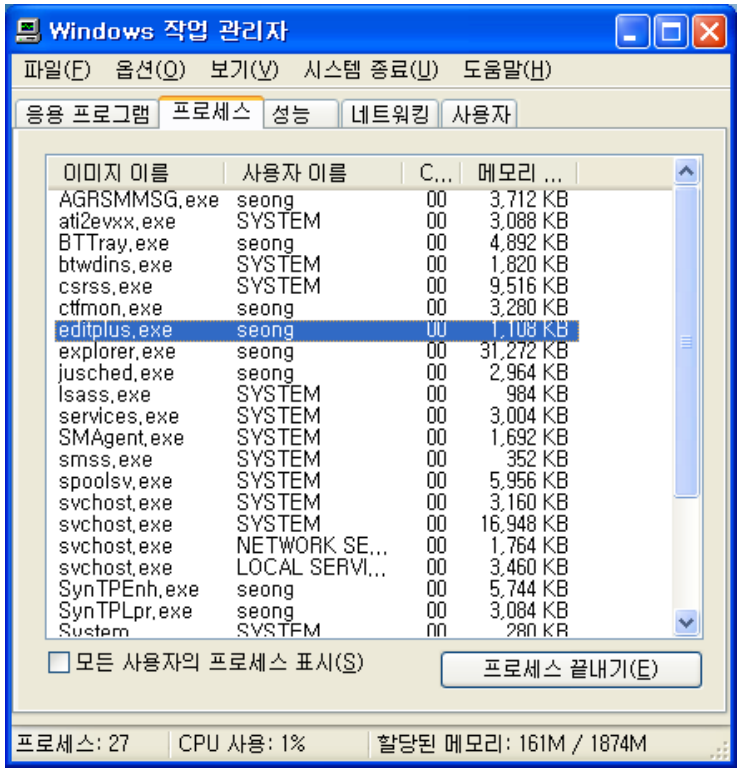
쓰레드

프로세스와 쓰레드(process & thread)

▶ 프로그램 : 실행 가능한 파일(HDD)



▶ 프로세스 : 실행 중인 프로그램(메모리)



쓰레드의 이해와 Thread 클래스의 상속

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

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

쓰레드는 모든 일의 기본 단위이다.

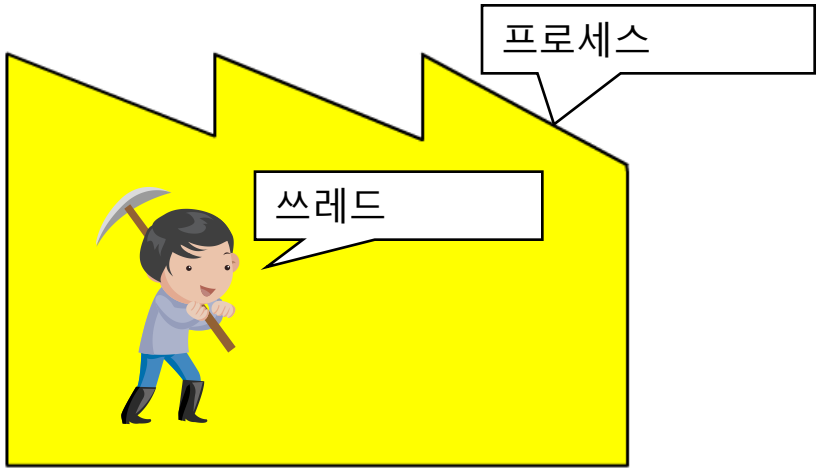
main 메소드를 호출하는 것도 프로세스 생성시 함께 생성되는 main 쓰레드를 통해서 이뤄진다.

프로세스와 쓰레드(process & thread)

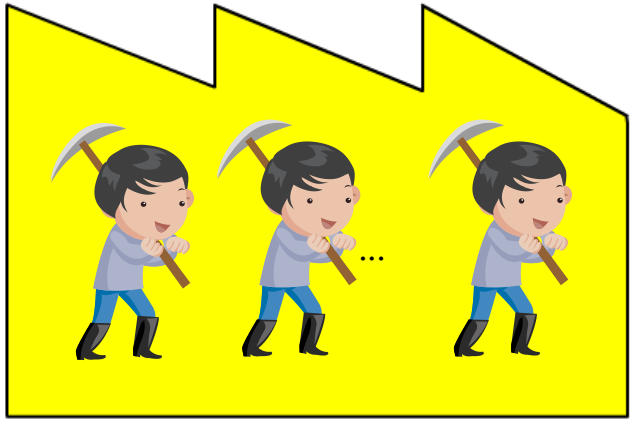
- 프로세스 : 실행 중인 프로그램, 자원(resources)과 쓰레드로 구성
- 쓰레드 : 프로세스 내에서 실제 작업을 수행.
  모든 프로세스는 하나 이상의 쓰레드를 가지고 있다.

프로세스 : 쓰레드 = 공장 : 일꾼

- 싱글 쓰레드 프로세스
  = 자원+쓰레드



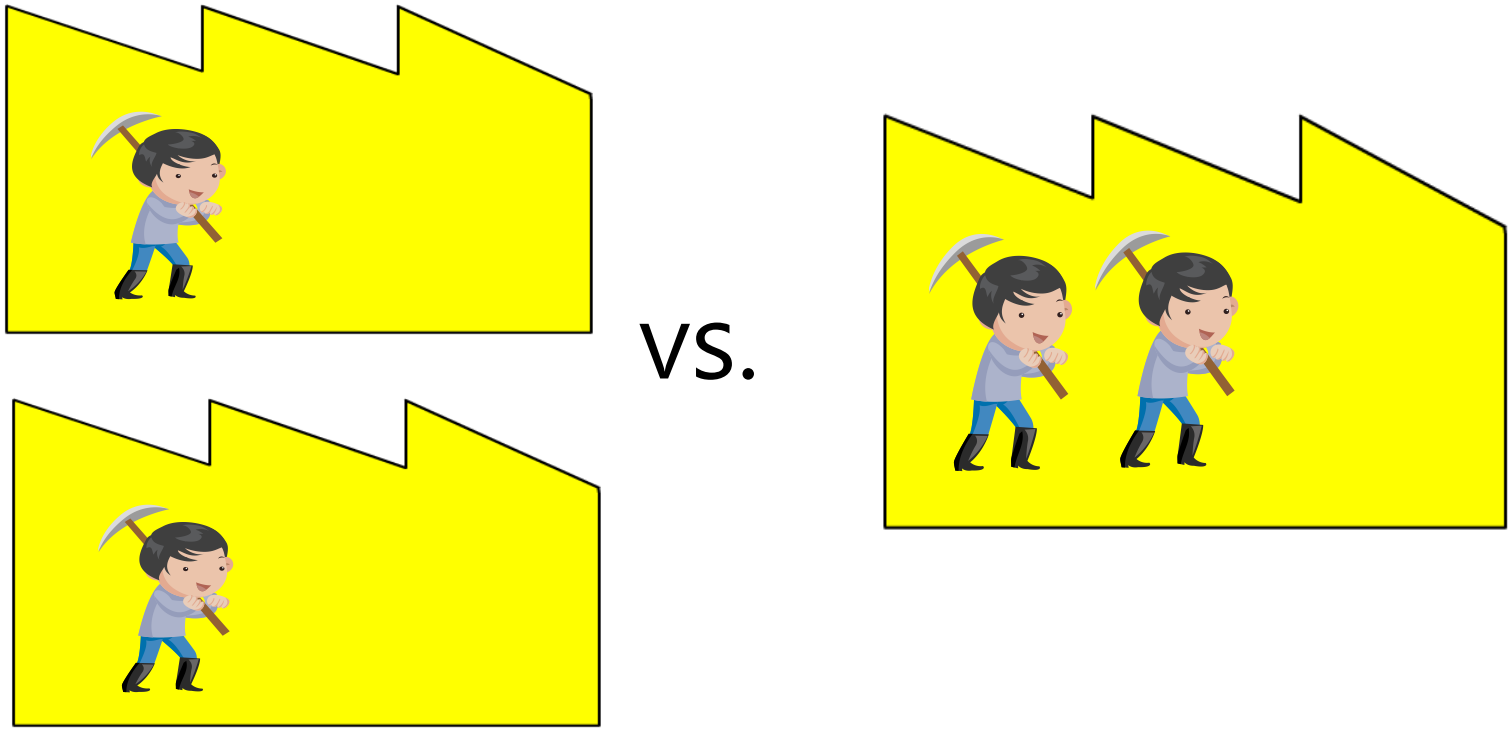
- 멀티 쓰레드 프로세스
  = 자원+쓰레드+쓰레드+…+쓰레드



멀티프로세스 vs. 멀티쓰레드

**"하나의 새로운 프로세스를 생성하는 것보다
하나의 새로운 쓰레드를 생성하는 것이 더 적은 비용이 든다."**

- 2 프로세스 1 쓰레드 vs. 1 프로세스 2 쓰레드



멀티쓰레드의 장단점

**"많은 프로그램들이 멀티쓰레드로 작성되어 있다.
그러나, 멀티쓰레드 프로그래밍이 장점만 있는 것은 아니다."**

장점	- 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."**

쓰레드의 이해와 생성

쓰레드의 구현과 실행

1. Thread클래스를 상속

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

2. Runnable인터페이스를 구현

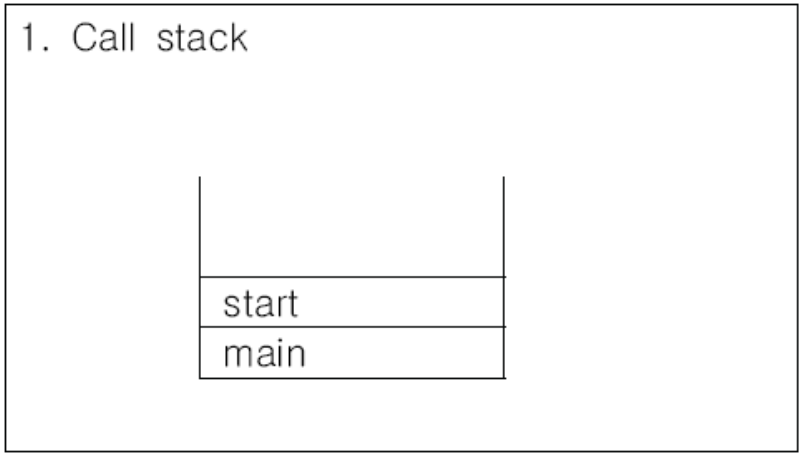
```
class MyThread implements Runnable {
    public void run() { /* 작업내용 */ } // Runnable인터페이스의 추상메서드 run()을 구현
}
```

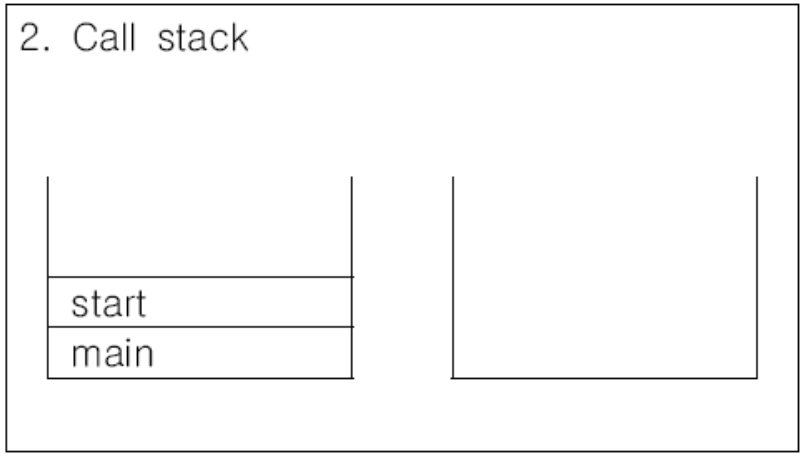
```
public interface Runnable {
    public abstract void run();
}
```

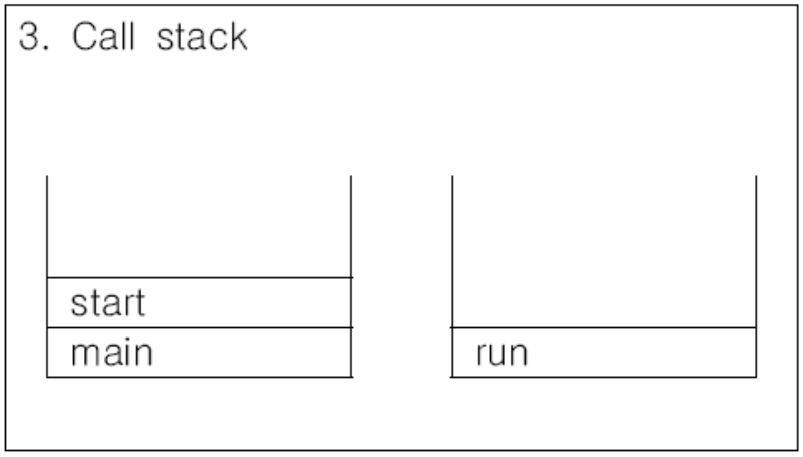
```
ThreadEx1_1 t1 = new ThreadEx1_1();

Runnable r = new ThreadEx1_2();
Thread t2 = new Thread(r);    // 생성자 Thread(Runnable target)
// Thread t2 = new Thread(new ThreadEx1_2());
```

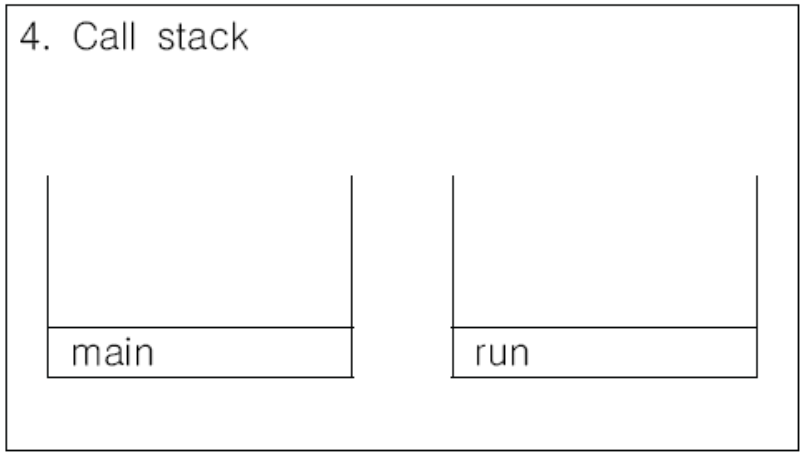
쓰레드의 이해와 Thread 클래스의 상속

start() & run()









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(); }
        }
    }
}
```

별도의 쓰레드 생성을 위해서는 별도의 쓰레드 클래스를 정의해야 한다. 쓰레드 클래스는 Thread를 상속하는 클래스를 의미 한다.

쓰레드의 이해와 Thread 클래스의 상속 예제

```
class ShowThread extends Thread
{
        String threadName;
        public ShowThread(String name) {  threadName=name; }

        public void run()  //run() 메소드 오버라이딩
        {
                for(int i=0; i<100; i++)
                {
                    System.out.println("안녕하세요. "+threadName+"입니다.");
                        try
                        {
                            sleep(100);   //static 메소드(일시적 멈춤)1/1000s
                        }
                        catch(InterruptedException e)
                        {
                                e.printStackTrace();
                        }
                }
        }
}
```

쓰레드의 이해와 Thread 클래스의 상속 예제

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

쓰레드를 생성하는 두 번째 방법

```
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.

쓰레드를 생성하는 두 번째 방법

```
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()));
}
```

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

쓰레드를 생성하는 두 번째 방법

```
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);
        }
}
```

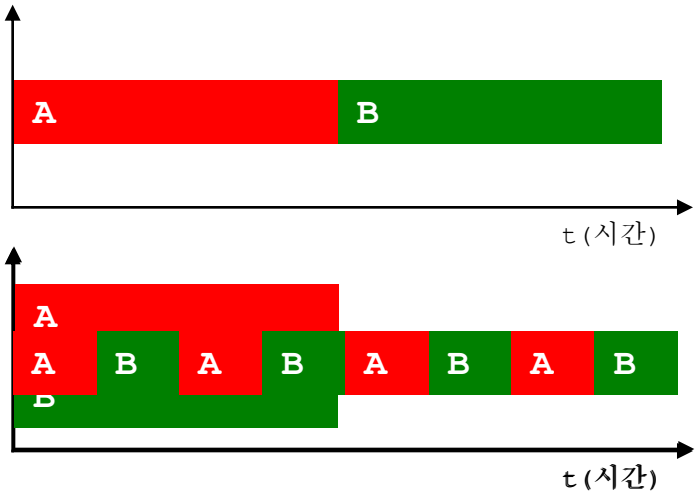
쓰레드를 생성하는 두 번째 방법

```
class RunnableThread {
        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();
                } //join() 해당 쓰레드가 종료될 때까지 실행을 멈출 때
                catch(InterruptedException e){
                        e.printStackTrace();
                }
                System.out.println("1~100까지의 합:
"+(at1.getNum()+at2.getNum()));
        }
}
```

싱글쓰레드 vs. 멀티쓰레드

▶ 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()
}
```

쓰레드

```
class ThreadEx4 {
	public static void main(String args[]) {
		long startTime = System.currentTimeMillis();
		for(int i=0; i < 300; i++) {
			System.out.print("-");
		}
		System.out.print("소요시간1:" +(System.currentTimeMillis()-
startTime));

		for(int i=0; i < 300; i++) {
			System.out.print("|");
		}

		System.out.print("소요시간2:"+(System.currentTimeMillis() -
startTime));
	}
}
```

쓰레드

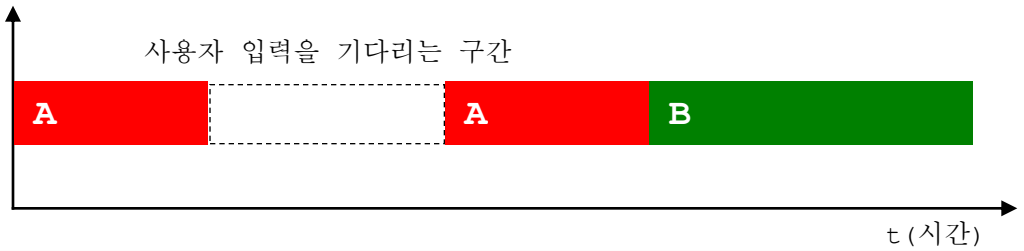
```
 class ThreadEx5 {
        static long startTime = 0;
        public static void main(String args[]) {
                ThreadEx5_1 th1 = new ThreadEx5_1();
                th1.start();
                startTime = System.currentTimeMillis();
                for(int i=0; i < 300; i++) {
                        System.out.print("-");
                }
                System.out.print("소요시간1:" + (System.currentTimeMillis() -
ThreadEx5.startTime));
        }
}
class ThreadEx5_1 extends Thread {
        public void run() {
                for(int i=0; i < 300; i++)
                        System.out.print("|");

                System.out.print("소요시간2:" + (System.currentTimeMillis()-
ThreadEx5.startTime));
        }
}
```

싱글쓰레드 vs. 멀티쓰레드

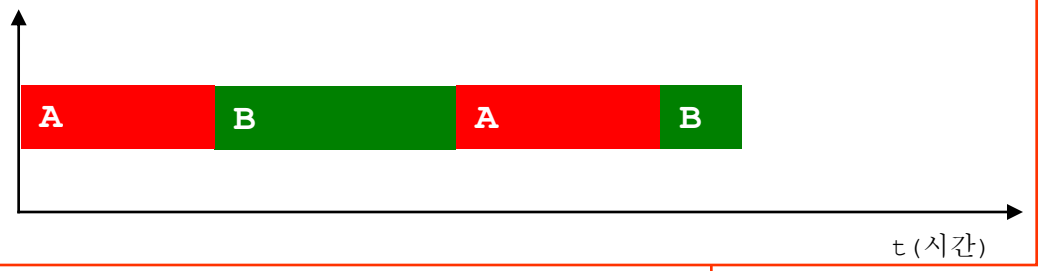
```
class ThreadEx6 {
    public static void main(String[] args){
        String input = JOptionPane.showInputDialog("아무 값이나 입력하세요.");
        System.out.println("입력하신 값은 " + input + "입니다.");

        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()
}
```

▶ 싱글쓰레드



▶ 멀티쓰레드



쓰레드

```
import javax.swing.JOptionPane;

class ThreadEx6
{
	public static void main(String[] args) throws Exception
	{
		String input = JOptionPane.showInputDialog("아무 값이나 입력하
세요.");
		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 ) {}
		}
	}
}
```

쓰레드

```
import javax.swing.JOptionPane;

class ThreadEx7 {
	public static void main(String[] args) throws Exception {
		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(Exception e ) {}
		}
	}
}
```

쓰레드

```
import javax.swing.JOptionPane;

class ThreadEx8 {
	static boolean inputCheck = false;

	public static void main(String[] args) throws Exception		{
		ThreadEx8_1 th1 = new ThreadEx8_1();
		ThreadEx8_2 th2 = new ThreadEx8_2();
		th1.start();
		th2.start();
	}
}

class ThreadEx8_1 extends Thread {
	public void run() {
		System.out.println("10초안에 값을 입력해야 합니다.");
		String input = JOptionPane.showInputDialog("아무 값이나 입력하
세요.");
		ThreadEx8.inputCheck = true;
		System.out.println("입력값은 " + input + "입니다.");
	}
}
```

쓰레드

```
class ThreadEx8_2 extends Thread {
	public void run() {
		for(int i=9; i >= 0; i--) {
			if(ThreadEx8.inputCheck) return;
			System.out.println(i);

			try {
				sleep(1000);
			} catch(InterruptedException e ) {}
		}
		System.out.println("10초 동안 값이 입력되지 않아 종료합니다.");
		System.exit(0);
	}
}
```

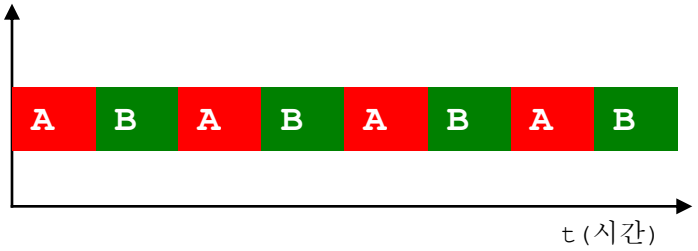
쓰레드의 특성

쓰레드의 우선순위(priority of thread)

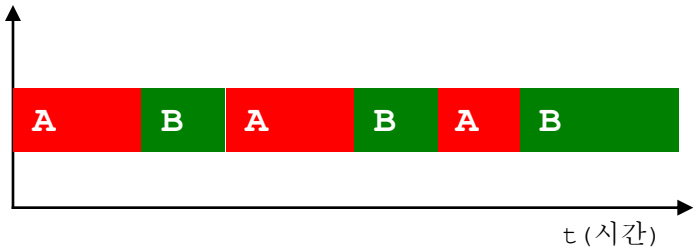
"작업의 중요도에 따라 쓰레드의 우선순위를 다르게 하여
특정 쓰레드가 초반에 더 많은 작업시간을 갖도록 할 수 있다."

```
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



쓰레드의 스케줄링과 우선순위 컨트롤

- 우선순위가 높은 쓰레드의 실행을 우선시 한다.
- 우선순위가 동일할 때는 CPU의 할당 시간을 나눈다.

```
class MessageSendingThread extends Thread
{
        String message;
        int priority;

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

쓰레드의 스케줄링과 우선순위 컨트롤

메소드 getPriority의 반환 값을 통해서 쓰레드의 우선순위를 확인할 수 있다.

실행결과에서 보이듯이, 우선 순위와 관련해서 별도의 지시를 하지 않으면, 동일한 우선순위의 쓰레드들이 생성 된다.

쓰레드의 스케줄링과 우선순위 컨트롤

```
class MessageSendingThread extends Thread
{
	String message;
	int priority;

	public MessageSendingThread(String str)
	{
		message=str;
	}

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

쓰레드의 스케줄링과 우선순위 컨트롤

```
class PriorityTestOne
{
	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();
	}
}
```

우선순위가 다른 쓰레드들의 실행

```
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()+")");
    }
}
```

Thread.MAX_PRIORITY는 상수로 10,

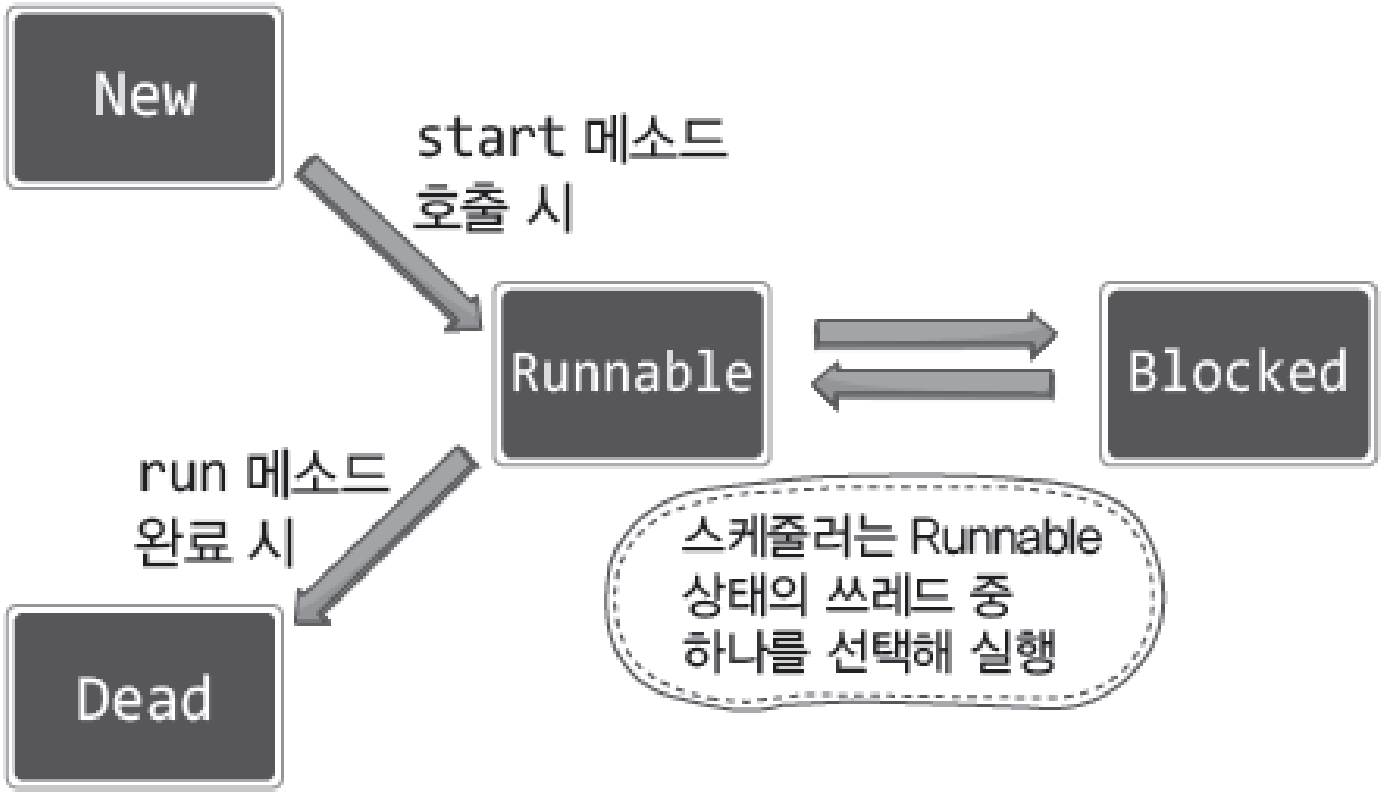
Thread.NORM_PRIORITY는 상수로 5,

Thread.MIN_PRIORITY는 상수로 1

우선순위가 다른 쓰레드들의 실행

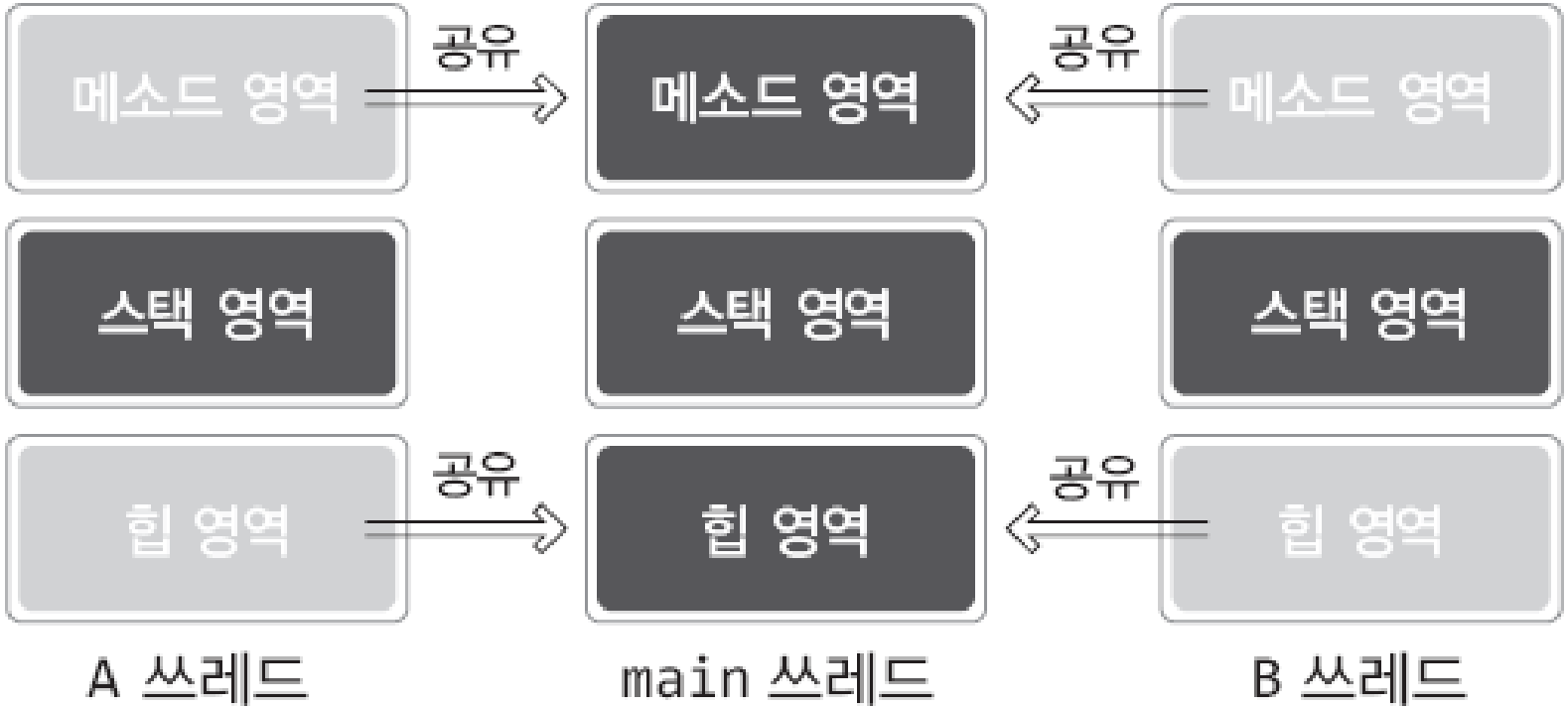
```
class PriorityTestTwo
{
	public static void main(String[] args)
	{
		MessageSendingThread tr1
			=new MessageSendingThread("First", Thread.MAX_PRIORITY);

		MessageSendingThread tr2
			=new MessageSendingThread("Second", Thread.NORM_PRIORITY);

		MessageSendingThread tr3
			=new MessageSendingThread("Third", Thread.MIN_PRIORITY);

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

쓰레드의 라이프 사이클



Runnable 상태의 쓰레드만이 스케줄러에 의해 스케줄링이 가능하다.
그리고 앞서 보인 sleep, join 메소드의 호출로 인해서 쓰레드는 Blocked 상태가 된다.
한번 종료된 쓰레드는 다시 Runnable 상태가 될 수 없지만,
Blocked 상태의 쓰레드는 조건이 성립되면 다시 Runnable 상태가 된다.

쓰레드의 메모리 구성



모든 쓰레드는 스택을 제외한 메소드 영역과 힙을 공유한다.
따라서 이 두 영역을 통해서 데이터를 주고 받을 수 있다.
스택은 쓰레드 별로 독립적일 수 밖에 없는 이유는, 쓰레드의 실행이 메소드의 호출을 통해서 이뤄지고, 메소드의 호출을 위해서 사용되는 메모리 공간이 스택이기 때문이다.

쓰레드간 메모리 영역의 공유 예제

```
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);
        }
}
```

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

쓰레드간 메모리 영역의 공유 예제

```
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);
        }
}
```

쓰레드간 메모리 영역의 공유 예제

```
class ThreadHeapMultiAccess
{
	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());
	}
}
```

쓰레드의 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.

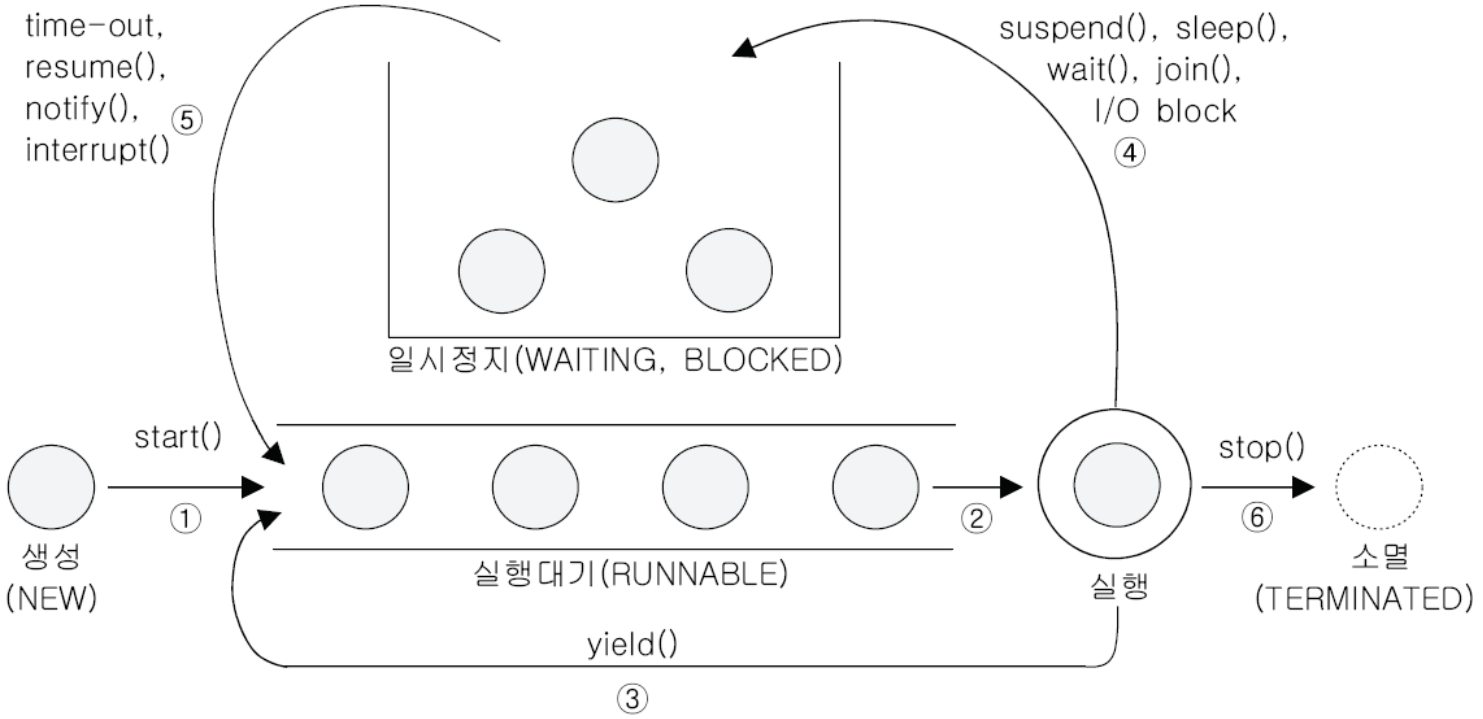
[표 12-2] 쓰레드의 스케줄링과 관련된 메서드

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

쓰레드의 라이프 사이클

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



쓰레드의 특징

```
class ThreadEx13 {
        static long startTime = 0;

        public static void main(String args[]) {

                ThreadEx13_1 th1 = new ThreadEx13_1();
                ThreadEx13_2 th2 = new ThreadEx13_2();

                th1.start();
                th2.start();
                startTime = System.currentTimeMillis();

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

                System.out.print("소요시간:" + (System.currentTimeMillis() -
ThreadEx13.startTime));
        } // main
}
```

쓰레드의 특징

```
class ThreadEx13_1 extends Thread {
	public void run() {
		for(int i=0; i < 300; i++) {
			System.out.print("-");
		}
	} // run()
}

class ThreadEx13_2 extends Thread {
	public void run() {
		for(int i=0; i < 300; i++) {
			System.out.print("|");
		}
	} // run()
}
```

쓰레드의 특징

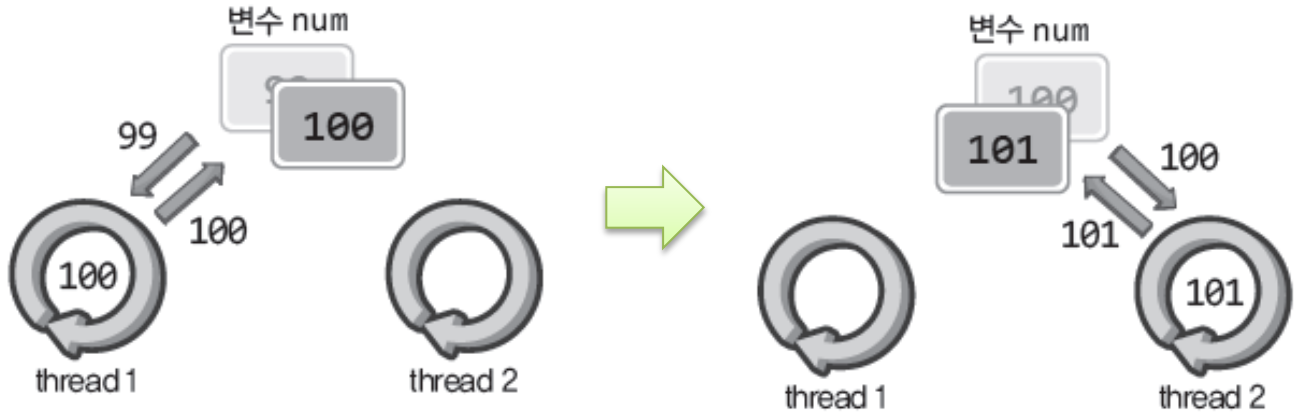
```
class ThreadEx14 {
	public static void main(String args[]) {
		ThreadEx14_1 th1 = new ThreadEx14_1();
		ThreadEx14_2 th2 = new ThreadEx14_2();
		th1.start();
		try {
			th1.join();
		} catch(InterruptedException e) {}
		th2.start();
	}
}

class ThreadEx14_1 extends Thread {
	public void run() {
		for(int i=0; i < 300; i++)
			System.out.print("-");
	}
}

class ThreadEx14_2 extends Thread {
	public void run() {
		for(int i=0; i < 300; i++)
			System.out.print("|");
	}
}
```

쓰레드의 특징

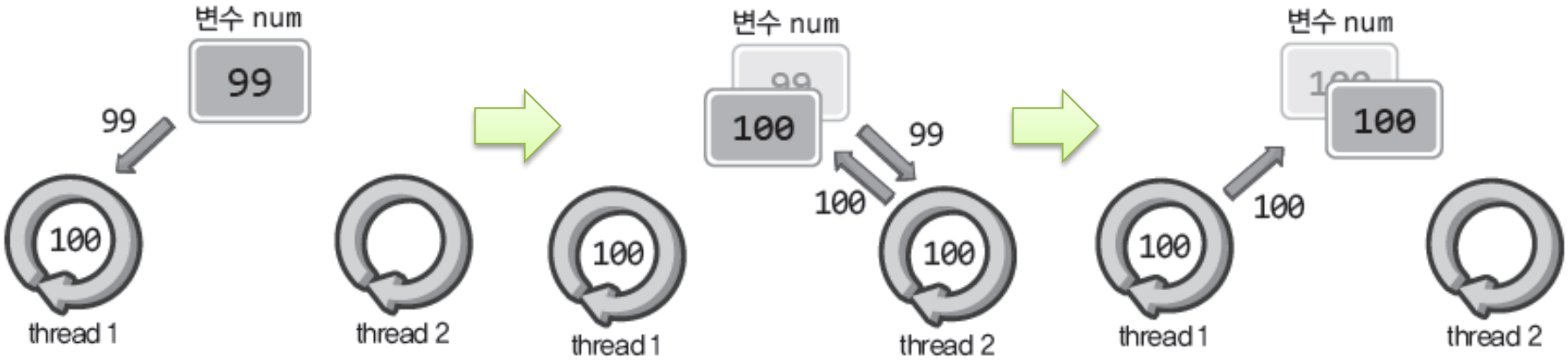
```
 class ThreadEx15 {
        public static void main(String args[]) {
                ThreadEx15_1 th1 = new ThreadEx15_1();
                ThreadEx15_2 th2 = new ThreadEx15_2();
                th1.start(); th2.start();
                try {
                        th1.sleep(5000);
                } catch(InterruptedException e) {}
                System.out.print("<<main 종료>>");
        } // main
}
class ThreadEx15_1 extends Thread {
        public void run() {
                for(int i=0; i < 300; i++)
                        System.out.print("-");
                System.out.print("<<th1 종료>>");
        } // run()
}
class ThreadEx15_2 extends Thread {
        public void run() {
                for(int i=0; i < 300; i++)
                        System.out.print("|");
                System.out.print("<<th2 종료>>");
        } // run()
}
```

동기화(Synchronization)

쓰레드의 메모리 접근방식과 그에 따른 문제점



둘 이상의 쓰레드가 하나의 메모리 공간에 동시 접근하는 것은 문제를 일으킨다.



Thread-safe

Note that this implementation is not synchronized

API 문서에는 해당 클래스의 인스턴스가 둘 이상의 쓰레드가 동시에 접근을 해도 문제가 발생하지 않는지를 명시하고 있다.
따라서 쓰레드 기반의 프로그래밍을 한다면, 특정 클래스의 사용에 앞서 쓰레드에 안전한지를 확인 해야 한다.

쓰레드의 동기화 - synchronized

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

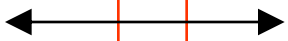
1. 특정한 객체에 lock을 걸고자 할 때

```
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)
}
```

Thread-safe 예제

```
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();
        }
}
```

Thread-safe 예제

```
class ThreadSyncError
{
        public static void main(String[] args)
        {
                Increment inc=new Increment();
                IncThread it1=new IncThread(inc);
                IncThread it2=new IncThread(inc);
                IncThread it3=new IncThread(inc);

                it1.start(); it2.start(); it3.start();

/*              try
                {
                        it1.join(); it2.join(); it3.join();
                }
                catch(InterruptedException e)
                {
                        e.printStackTrace();
                } */
                System.out.println(inc.getNum());
        }
}
```

Thread-safe 예제

```
class ThreadEx24 {
	public static void main(String args[]) {
		Runnable r = new RunnableEx24();
		Thread t1 = new Thread(r);
		Thread t2 = new Thread(r);

		t1.start();
		t2.start();
	}
}

class Account {
	int balance = 1000;

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

Thread-safe 예제

```
class RunnableEx24 implements Runnable {
	Account acc = new Account();

	public void run() {
		while(acc.balance > 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.balance);
		}
	} // run()
}
```

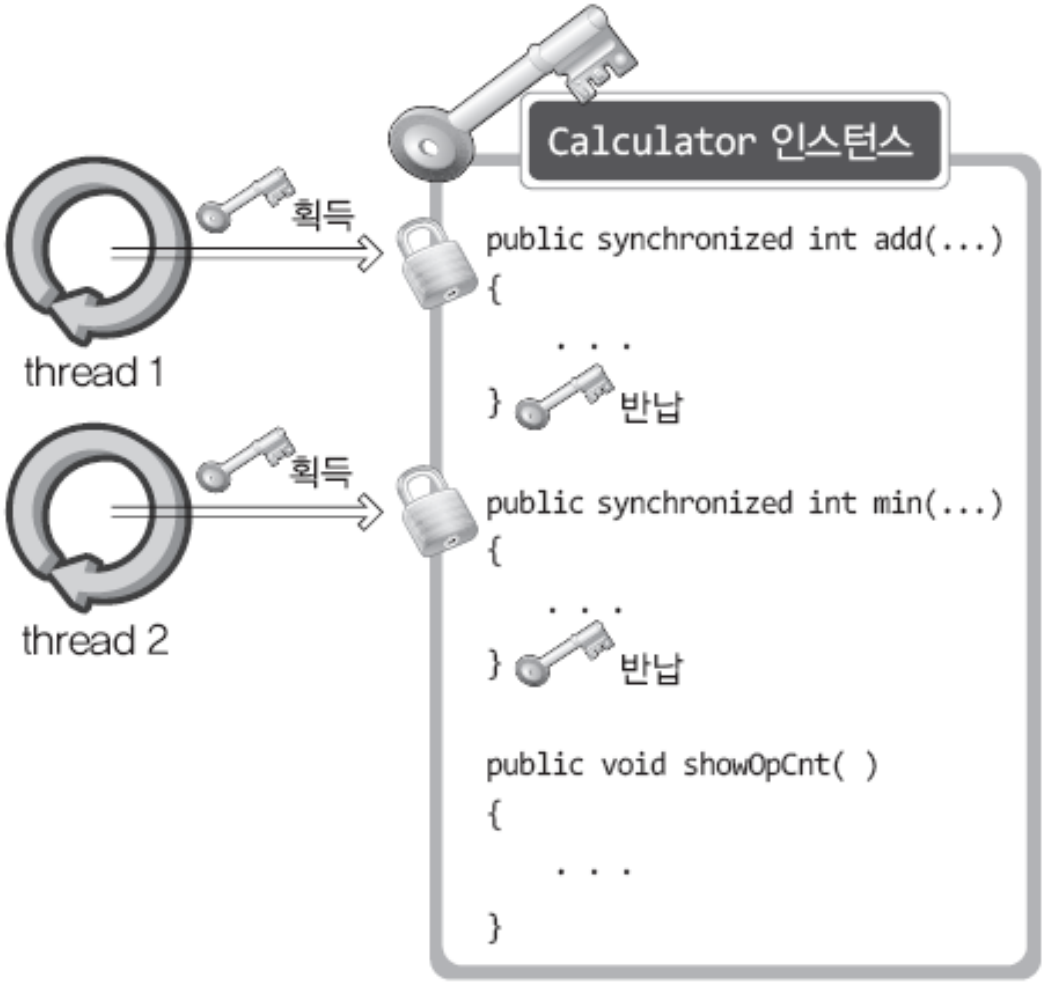
쓰레드의 동기화 기법1: synchronized 기반 동기화 메소드

```
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();
        }
}
```

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 존재한다.

synchronized 기반 동기화 메소드의 정확한 이해

```
class Calculator
{
        int opCnt=0;

        public int add(int n1, int n2) // public synchonized int add(int n1, int n2)
        {
                opCnt++;
                return n1+n2;
        }
        public int min(int n1, int n2) // public synchonized int min(int n1, int n2)
        {
                opCnt++;
                return n1-n2;
        }
        public void showOpCnt()
        {
                System.out.println("총 연산 횟수: "+opCnt);
        }
}
```

synchronized 기반 동기화 메소드의 정확한 이해

```
class AddThread extends Thread
{
	Calculator cal;
	public AddThread(Calculator cal) { this.cal=cal; }
	public void run()
	{
		System.out.println("1+2="+cal.add(1, 2));
		System.out.println("2+4="+cal.add(2, 4));
	}
}

class MinThread extends Thread
{
	Calculator cal;
	public MinThread(Calculator cal) { this.cal=cal; }
	public void run()
	{
		System.out.println("2-1="+cal.min(2, 1));
		System.out.println("4-2="+cal.min(4, 2));
	}
}
```

synchronized 기반 동기화 메소드의 정확한 이해

```
class ThreadSyncMethod
{
	public static void main(String[] args)
	{
		Calculator cal=new Calculator();
		AddThread at=new AddThread(cal);
		MinThread mt=new MinThread(cal);

		at.start(); mt.start();

		try
		{
			at.join();  mt.join();
		}
		catch(InterruptedException e)
		{
			e.printStackTrace();
		}
		cal.showOpCnt();
	}
}
```

synchronized 기반 동기화 메소드의 정확한 이해

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

쓰레드의 동기화 기법2: synchronized 기반 동기화 블록

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

쓰레드의 동기화 기법2: synchronized 기반 동기화 블록

```
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;
}
```

synchronized(this)에서 this는 동기화의 대상을 알리는 용도로 사용이 되었다.

즉, 메소드가 호출된 인스턴스 자신의 열쇠를 대상으로 동기화를 진행하는 문장이다.

쓰레드의 동기화 기법2: synchronized 기반 동기화 블록

```
class IHaveTwoNum
{
        int num1=0;
        int num2=0;

        public void addOneNum1() { num1+=1; }
        public void addTwoNum1() { num1+=2; }

        public void addOneNum2() { num2+=1; }
        public void addTwoNum2() { num2+=2; }

        public void showAllNums()
        {
                System.out.println("num1: "+num1);
                System.out.println("num2: "+num2);
        }
}
```

쓰레드의 동기화 기법2: synchronized 기반 동기화 블록

```
class AccessThread extends Thread
{
	IHaveTwoNum twoNumInst;

	public AccessThread(IHaveTwoNum inst)
	{
		twoNumInst=inst;
	}

	public void run()
	{
		twoNumInst.addOneNum1();
		twoNumInst.addTwoNum1();

		twoNumInst.addOneNum2();
		twoNumInst.addTwoNum2();
	}
}
```

쓰레드의 동기화 기법2: synchronized 기반 동기화 블록

```
class SyncObjectKey
{
	public static void main(String[] args)
	{
		IHaveTwoNum numInst=new IHaveTwoNum();

		AccessThread at1=new AccessThread(numInst);
		AccessThread at2=new AccessThread(numInst);

		at1.start(); at2.start();

		try
		{
			at1.join();at2.join();
		}
		catch(InterruptedException e)
		{
			e.printStackTrace();
		}
		numInst.showAllNums();
	}
}
```

동기화 블록의 예

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

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

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

동기화 블록의 예

```
class IHaveTwoNum
{
        int num1=0;
        int num2=0;
        public void addOneNum1() { synchronized(key1) { num1+=1; } }
        public void addTwoNum1() { synchronized(key1) { num1+=2; } }

        public void addOneNum2() {  synchronized(key2) { num2+=1;  } }
        public void addTwoNum2() {  synchronized(key2) { num2+=2;  } }

        public void showAllNums()
        {
                System.out.println("num1: "+num1);
                System.out.println("num2: "+num2);
        }

        Object key1=new Object();
        Object key2=new Object();
}
```

동기화 블록의 예

```
class AccessThread extends Thread
{
        IHaveTwoNum twoNumInst;

        public AccessThread(IHaveTwoNum inst)
        {
                twoNumInst=inst;
        }

        public void run()
        {
                twoNumInst.addOneNum1();
                twoNumInst.addTwoNum1();

                twoNumInst.addOneNum2();
                twoNumInst.addTwoNum2();
        }
}
```

동기화 블록의 예

```
class SyncObjectKeyAnswer{

        public static void main(String[] args){

                IHaveTwoNum numInst=new IHaveTwoNum();

                AccessThread at1=new AccessThread(numInst);
                AccessThread at2=new AccessThread(numInst);

                at1.start();            at2.start();

                try{
                        at1.join();
                        at2.join();
                }
                catch(InterruptedException e){
                        e.printStackTrace();
                }
                numInst.showAllNums();
        }
}
```