제 1 장 멀티쓰레딩

- 쓰레드란 무엇인가?
- 쓰레드의 특성
- 쓰레드 우선순위
- 협력(cooperation) 쓰레드와 탐욕(selfish) 쓰레드
- 동기화
- 타이머들
- 쓰레드와 스윙
- 쓰레드간의 통신을 위한 파이프(Pipe)의 사용

여러분들은 이미 멀티태스킹에 대해 친숙해져 있을 수 있다:동시에 하나 이상의 프로그램을 수행할 수 있는 능력을 말한다. 예를 들면, 여러분은 편집을 하거나 팩스를 보내는 동안 프린트를 할 수 있다. 물론 멀티프로세스 컴퓨터를 갖지 않았다면, 실제로는 운영체제에서 각각의 프로그램에 자원들을 할당함으로써 마치 동시에 수행되는 것처럼 보여지게 하는 것이다. 이러한 자원의 할당은 사용자가 데이터를 입력하는 등의 일로 컴퓨터를 굉장히 바쁘게 한다고 생각할 지라도 컴퓨터는 실제로 별로 바쁘지 않기 때문에 가능하다. (제 아무리 빠른 타이피스트라도 초당 20타 정도의 속도밖에 되지 않는다). 멀티태스킹은 운영체제가 제어권을 가지고 프로그램들을 일방적으로 수행하게 하든지 아니면 프로그램들이 제어권을 가지고 서로 양보해가며 수행을 하는 두 가지 방법으로 실행될 수 있다. 전자의 경우 선점형(Preemptive) 멀티태스킹이라 하고 후자는 협력형(Cooperative) 멀티태스킹이라 한다.(또는 단순히 비선점형 멀티태스킹이라고도 한다.) 윈도우 3.1 은 협력형 멀티태스킹 시스템이고 윈도우 NT(와 32 비트 프로그램을 위한 윈도우 95)는 선점형 멀티태스킹 시스템이다. (비록 구현하기가 더욱 어렵지만 선점형 멀티태스킹이 더욱 효과적이다. 협력형 멀티태스킹하에서 서투르게 작동하는 프로그램들은 모든 것을 차지할 수 있다.)

멀티쓰레드 프로그램은 한단계 수행단위를 낮춤으로서 멀티태스킹의 개념을 확장한다.: 개별 프로그램들은 동시에 복수의 연산을 수행할 수 있는 능력을 가지고 있다. 일반적으로 각각의 연산 단위를 쓰레드라 한다. 동시에 하나이상의 쓰레드를 수행하는 프로그램을 멀티쓰레드라고 말한다. 각각의 쓰레드는 분리된 컨텍스(Context)에서 수행하고 있다고 간주한다.: 컨텍스들은 각각의 쓰레드가 레지스터, 메모리 그리고 자신의 코드를

갖는 CPU 를 소유하고 있는 것처럼 만든다.

멀티 프로세스들과 멀티 쓰레드들 사이에는 하나의 기본적인 차이점이 존재한다. 각각의 프로세스는 자신의 완전한 변수들을 갖는다. 그러나 쓰레드들은 그들이 활동하고 있는 프로그램내의 데이터를 공유한다. 이 말은 다수 위험하게 들린다. 실제 위험할 수 있다. 우리는 이 장에서 이런 사실을 보게 될 것이다. 그러나 새로운 프로세스를 구동시키는 것보다 개별적인 쓰레드들을 생성하고 파괴하는 것이 훨씬 오버헤드가 적다. 이 점이 대부분의 현대 운영체제에서 멀티쓰레딩을 지원하는 이유이다. 더군다나 프로세스간의 통신은 쓰레드간의 통신보다 더욱 느리고 제한적이다.
실제 멀티쓰레딩은 무척 유용하다: 예를 들면, 부라우져는 복수 호스트를 처리하는 능력을 갖을 수도 있고 데이터를 받고 있는 동안 전자우편 윈도우를 열거나 다른 페이지를 볼 수도 있다. 자바는 그 자체 시스템에서 백그라운드 실행을 통해 사용하지 않는 메모리인 가베지(garbage)를 수집하는 쓰레드를 사용한다. - 이것은 여러분이 직접 메모리를 관리하는 부담을 덜어준다. GUI 프로그램은 사용자로부터 발생하는 이벤트를 수집하기 위해 독립적인 쓰레드를 갖고 있다.
이 장은 여러분에게 자바 응용 프로그램과 애플릿에 멀티쓰레딩 능력을 추가하는 방법을 보여준다.

노트 : 많은 프로그래밍 언어에서 여러분은 멀티쓰레드 프로그래밍을 위해 외부의 쓰레드 패키지를 사용해야 한다. 그러나 자바는 언어내에 멀티쓰레딩 기능을 포함하고 있다. 이것은 여러분의 작업을 보다 쉽게 만들어 준다.

쓰레드란 무엇인가?

먼저 멀티 쓰레드들을 사용하지 않고 작동되는 자바 프로그램을 살펴보고 그 결과 사용자들이 그 프로그램을 통해 여러 작업들을 수행하는 것을 어렵게 만드는 자바 프로그램을 살펴보는 것부터 시작해보자. 이것을 분석하고 나서 우리는 여러분에게 이 프로그램을 분리된 쓰레드에서 수행시키는 것이 얼마나 쉬운지 보여줄 것이다. 이 프로그램은 끊임없이 공을 이동시키고 만약 벽에 부딪치는 지를 검사하고 나서 다시 공을 그리는 것에 의해 공을 뛰게 만드는 애니메이션이다.(그림 1-1 을 참조하라.)

그림 1-1 : 뛰는 공 애니메이션에 쓰레드 사용하기

여러분이 시작 버튼을 누르자마자 프로그램은 스크린의 좌측 상단 모서리로 부터 공을 그려낸다. 그리고 뛰기 시작한다. 시작 버튼의 핸들러는 Ball 클래스의 bounce() 메소드를 호출한다. bounce() 메소드는 1000 번 이동하는 루프를 포함하고 있다. 각각의 이동후

우리는 Thread 클래스의 정적 sleep 메소드를 호출하여 5 밀리 초 동안 볼을 멈춘다.

```
class Ball
{       ....
        public void bounce()
        {       draw();
                for( int i = 1; i <= 1000; i++)
`               {       move();
                        try{    Thread.sleep(5);    }
                        catch(InterruptedException e) {}
                }
        }
}
```

Sleep() 메소드의 호출은 새로운 쓰레드를 생성하지 않는다. – sleep() 메소드는 현재의 쓰레드를 쉬게 하는 Thread 클래스의 정적 메소드이다.
Sleep 메소드는 InterruptedException 을 발생할 수 있다. 우리는 나중에 이 Exception 과 적절한 조치를 논의 하게 될 것이지만 지금은 무시하고 지나 가기로 한다.
여러분이 프로그램을 수행할 때 공이 잘 튀는 것을 발견할 것이다. 그러나 이것은 전적으로 프로그램에 달려있다. 만일 여러분이 1,000 반복이 끝나기 전에 튀는 공이 싫증이 나면 종료버튼을 누른다. 그러나 그 공은 이에 상관없이 계속 튀고 있다. 여러분은 공이 반복해서 튀기를 종료하기 전까지 프로그램과 상호작동할 수 없다.

이것은 실제 좋은 상황이 아니다. 특히 네트웍상의 연결을 통해 데이터를 읽는 소프트웨어일 경우 여러분이 인터럽트를 걸고싶은 때에, 시간을 허비하는 일반적인 작업들로 고착될 것이다. 예를 들면, 여러분이 큰 이미지를 다운로드한다고 가정하자. 이 그림의 일부를 본 다음에 나머지를 보지 않기로 했다면 여러분은 Stop 버튼이나 Back 버튼을 선택하여 데이터를 받는 과정을 중단할 수 있어야 한다. 다음 절에서 우리는 사용자들이 분리된 쓰레드안에서 수행중인 코드의 일부분을 조정하는 방법을 보게 될 것이다.
다음 예제 1-1 은 프로그램의 완전한 코드를 제공한다.

예제 1-1 : Bounce.java

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

```
public class Bounce
{   public static void main(String[] args)
    {   JFrame frame = new BounceFrame();
        frame.show();
     }
}

class BounceFrame extends JFrame
{   public BounceFrame()
    {   setSize(300, 200);
        setTitle("Bounce");

        addWindowListener(new WindowAdapter()
            {   public void windowClosing(WindowEvent e)
                {   System.exit(0);
                }
            } );

        Container contentPane = getContentPane();

        canvas = new JPanel();
        contentPane.add(canvas, "Center");
        JPanel p = new JPanel();
        addButton(p, "Start",
            new ActionListener()
            {   public void actionPerformed(ActionEvent evt)
                {   Ball b = new Ball(canvas);
                    b.bounce();
                }
            });

        addButton(p, "Close",
            new ActionListener()
            {   public void actionPerformed(ActionEvent evt)
                {   System.exit(0);
```

```
                }
            });
        contentPane.add(p, "South");
    }

  public void addButton(Container c, String title,
      ActionListener a)
  {  JButton b = new JButton(title);
       c.add(b);
       b.addActionListener(a);
    }

  private JPanel canvas;
}

class Ball
{  public Ball(JPanel b) { box = b; }

  public void draw()
  {  Graphics g = box.getGraphics();
     g.fillOval(x, y, XSIZE, YSIZE);
       g.dispose();
    }

  public void move()
  {  Graphics g = box.getGraphics();
       g.setXORMode(box.getBackground());
     g.fillOval(x, y, XSIZE, YSIZE);
      x += dx;
      y += dy;
     Dimension d = box.getSize();
     if (x < 0)
     { x = 0; dx= -dx; }
    if (x + XSIZE >= d.width)
     { x = d.width - XSIZE; dx = -dx; }
     if (y < 0)
```

```
        { y = 0; dy = -dy; }
        if (y + YSIZE >= d.height)
        { y = d.height - YSIZE; dy = -dy; }
        g.fillOval(x, y, XSIZE, YSIZE);
          g.dispose();
      }

    public void bounce()
     {   draw();
       for (int i = 1; i <= 1000; i++)
          {   move();
            try { Thread.sleep(5); }
             catch(InterruptedException e) {}
          }
     }

    private JPanel box;
   private static final int XSIZE = 10;
    private static final int YSIZE = 10;
    private int x = 0;
    private int y = 0;
    private int dx = 2;
    private int dy = 2;
}
```

노트 : 이 예제에서는 공이 움직이는 것을 보이기 위해 XOR 모드를 사용했다. 이것은 프로그램을 매우 단순화 해주기 때문에 예제에서 만든 쓰레드만 중점적으로 볼 수 있다. 그러나 중첩되는 공들이 정확히 표현되지 않아 화질은 다소 떨어진다. 더 좋은 방법은 BouncePanel 에서 모든 공들을 Vector 로 관리하는 것이다. paintComponent 메소드가 모든 공들이 그려지도록 할 것이고 move 메소드에서 repaint 메소드를 부르면 화면에 보이게 될 것이다. 그러나 이 부분은 독자의 연습으로 남겨 둔다.

다른 태스크들에게 기회를 주기위해 쓰레드 사용

우리는 분리된 쓰레드내에서 공을 이동하는 코드를 수행함으로서 우리의 튀는 공 프로그램을 보다 민감하게 만들 것이다.

노트 : 대부분의 컴퓨터들이 다중 프로세스를 가지고 있지 않기 때문에 자바는 각각의 쓰레드가 잠시동안 수행될 기회를 얻기 위한 메카니즘을 사용한다. 그런 다음 다른 쓰레드를 수행한다. 자바 가상 머신은 쓰레드 스케쥴링 패키지를 제공하기 위해 보통 호스트의 OS 에 의존한다.

다음 프로그램에서 우리는 두개의 쓰레드들을 사용한다.: 하나는 튀는 공을 위한 것이고 다른 하나는 사용자 인터페이스를 위한 메인 쓰레드이다. 각각의 쓰레드가 수행할 기회를 얻기때문에 메인 쓰레드는 공이 튀는 동안 여러분이 종료버튼을 눌렀을 때 이를 알아채는 기회를 가지고 있다. 메인 쓰레드는 "종료"를 처리할 수 있다.

분리된 쓰레드내에서 코드를 수행하기 위한 절차는 간단하다.: Thread 로 부터 상속받은 클래스의 run 메소드안에 코드를 위치시킨다.

우리의 튀는 공 프로그램을 분리된 쓰레드에서 수행하기 위해 우리는 Thread 클래스로부터 ball 클래스를 상속받게 만들기만 하면 된다. 그리고 다음의 코드처럼 run 메소드를 재명명한다.

```
class Ball extends Thread
{       . . .
        public void run()
        {       draw();
                for( int i=1; i <= 1000, i++)
                {       move();
                        try{    sleep(5);   }
                        catch(InterruptedException e) {}
                }
        }
}
```

여러분은 우리가 InterruptedException 이라 불리는 예외상황을 감지하고 있다는 사실을 알 수 있을 것이다. sleep 과 wait 같은 메소드들은 여러분의 쓰레드가 다른 쓰레드의 interrupt 호출에 의해 중단됐을 때 이 예외상황을 투구한다. 쓰레드에 인터럽트를 거는 것은 쓰레드의 주의를 끄는 아주 과감한 방법이다. 전형적으로 쓰레드를 종료하기 위해 쓰레드에 인터럽트를 건다. 그래서 위에서는 InterruptedException 이 발생하면 종료를 한다.

쓰레드의 시작과 구동

여러분이 Thread 로 부터 상속받은 객체를 구성할 때 자바는 자동적으로 run 메소드를 수행하지 않는다.

```
Ball b = new Ball( . . .); // 아직 수행되지 않는다.
```

여러분은 실제 쓰레드를 수행하기 위해 여러분의 객체내에 start 메소드를 호출해야 할 것이다.

```
b.start();
```

노트 : 직접 run 메소드를 수행하지 말라-쓰레드가 준비되고 실행할 준비가 될 때 start 가 호출될 것이다. 직접 run 메소드를 수행하면 새로운 쓰레드가 수행되지 않고 동일한 쓰레드가 수행된다.

자바에서 쓰레드는 실행하고 있지 않을 때(idle) 다른 쓰레드에게 이것을 알릴 필요가 있다. 그리고 다른 쓰레드는 그들의 run 메소드내의 코드를 수행시킬 기회를 잡을 수 있다.(그림 1-2 참조) 이것을 수행하는 일반적인 방법은 sleep 메소드를 통하는 것이다. (sleep 은 Thread 내의 정적 메소드이다.) Ball 클래스의 run 메소드는 다음 5 밀리초 동안 아무 일도 수행하지 않을 것이라는 사실을 알리기 위해 sleep(5) 를 사용한다. 5 밀리초가 경과된 후 해당 쓰레드가 다시 수행될 것이다. 그러나 이 동안 다른 쓰레드들이 작업을 수행할 기회를 갖는다.

TIP : Thread 클래스에는 많은 정적 메소드가 있다. 정적 메소드들은 메소드를 실행하는 current Thread 에 영향을 미친다. 예를 들면 정적 메소드 sleep 은 sleep 메소드를 호출하는 쓰레드를 idle 한 상태로 변환한다.

디자인 관점에서 Thread 클래스를 확장한 Ball 클래스를 갖는 것은 다소 이상해 보인다. 공은 스크린상에서 이동하고 구석에서 튀는 객체이다. 여기에 상속을 위한 is-a 규칙이 적용된 것인가? 공은 쓰레드인가? 실제 그렇지 않다. 여기서 우리는 기술적 이유에서 엄격하게 상속기능을 사용하고 있다. 여러분이 조정할 수 있는 쓰레드를 얻기 위해 여러분은 run 메소드를 갖는 쓰레드 객체를 필요로 한다. 우리는 run 메소드에서 사용하는 메소드와 인스턴스 필드들을 갖는 클래스에 run 메소드를 추가해도 된다. 그러므로 우리는 Ball 클래스를 Thread 의 하위 클래스로 만든다. 이장 뒷 부분에서는 Thread 클래스에서 상속을 받지 않기 위해 Runnable 인터페이스를 사용하는 것을 보일 것이다.

그림 1-2 : UI 와 Ball 쓰레드

다음 예제 1-2 는 완전한 코드이다.

예제 1-2 : BounceThread.java

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

public class BounceThread
{   public static void main(String[] args)
    {   JFrame frame = new BounceThreadFrame();
        frame.show();
    }
}

class BounceThreadFrame extends JFrame
{   public BounceThreadFrame()
    {   setSize(300, 200);
        setTitle("Bounce");

        addWindowListener(new WindowAdapter()
            {   public void windowClosing(WindowEvent e)
                {   System.exit(0);
                }
            } );

        Container contentPane = getContentPane();
        canvas = new JPanel();
        contentPane.add(canvas, "Center");
        JPanel p = new JPanel();
        addButton(p, "Start",
            new ActionListener()
            {   public void actionPerformed(ActionEvent evt)
```

```
                {   Ball b = new Ball(canvas);
                    b.start();
                }
            });

        addButton(p, "Close",
            new ActionListener()
            {   public void actionPerformed(ActionEvent evt)
                {   canvas.setVisible(false);
                    System.exit(0);
                }
            });
        contentPane.add(p, "South");
    }

    public void addButton(Container c, String title,
        ActionListener a)
    {   JButton b = new JButton(title);
        c.add(b);
        b.addActionListener(a);
    }

    private JPanel canvas;
}

class Ball extends Thread
{  public Ball(JPanel b) { box = b; }

    public void draw()
    {   Graphics g = box.getGraphics();
        g.fillOval(x, y, XSIZE, YSIZE);
        g.dispose();
    }

    public void move()
    {   if (!box.isVisible()) return;
```

```
    Graphics g = box.getGraphics();
      g.setXORMode(box.getBackground());
      g.fillOval(x, y, XSIZE, YSIZE);
    x += dx;
    y += dy;
    Dimension d = box.getSize();
    if (x < 0)
   { x = 0; dx = -dx; }
   if (x + XSIZE >= d.width)
    { x = d.width - XSIZE; dx = -dx; }
    if (y < 0)
   { y = 0; dy = -dy; }
     if (y + YSIZE >= d.height)
    { y = d.height - YSIZE; dy = -dy; }
    g.fillOval(x, y, XSIZE, YSIZE);
      g.dispose();
 }

public void run()
 {   try
     {   draw();
      for (int i = 1; i <= 1000; i++)
          {   move();
              sleep(5);
          }
     }
    catch(InterruptedException e) {}
 }

 private JPanel box;
private static final int XSIZE = 10;
private static final int YSIZE = 10;
private int x = 0;
private int y = 0;
private int dx = 2;
private int dy = 2;
```

}

java.lang.Thread

- Thread()

새로운 쓰레드를 생성한다. 쓰레드는 run 메소드를 갖어야 한다. 여러분은 run 함수를 실행하기 위해 쓰레드를 시작해야 한다.

- void run()

여러분은 이 메소드를 오버라이드해야 한다. 그리고 쓰레드내에서 수행하고자 하는 코드를 추가해야 한다.

- void start()

쓰레드를 동작시킨다. 이것는 run() 메소드를 호출한다. 즉시 반환될 것이다. 동시에 새로운 쓰레드가 수행된다.

- static void sleep(long millis)

현재 실행중인 쓰레드를 명시한 특정 밀리초만큼 자고 있게 만든다. 이 메소드가 정적 메소드임에 주목하라.

다중 쓰레드 구동

이전 절의 프로그램을 수행하라. 현재 공이 수행하는 동안 다시 시작 버튼을 선택하라. 몇번 더 시작 버튼을 선택하라. 여러분은 그림 1-3 에서 처럼 수많은 공이 사방으로 튀는 것을 보게 될 것이다. 각각의 공은 마지막 도착 지점까지 1,000 번 동안 이동할 것이다.

그림 1-3 : 다중 쓰레드들

이것은 자바 쓰레드 구조의 훌륭한 이점이다. 이것은 손쉽게 수많은 독립 객체를 병렬로 수행되는 것처럼 보이게 만든다.
여러분은 현재 수행되는 쓰레드들을 열거할 수 있다. - “쓰레드 그룹” 절의 API 를 참조하라.

쓰레드의 특성

쓰레드 상태

쓰레드들은 다음 4가지 상태중 하나일 수 있다.

- 신규(new)
- 수행가능한(runnable)
- 블록된(blocked)
- 종료(dead)

다음은 이들 각각에 대한 설명이다.

신규(New)

여러분이 new 연산자를 통해 새로운 쓰레드를 생성할 때- 예를 들면, new Ball() - 이것은 아직 수행하지 않는다. 이것은 new 상태에 있다는 것을 의미한다. 쓰레드가 new 상태에 있을 때 내부의 코드는 수행되지 않는다. 쓰레드가 수행되기 전에 특정한 양의 정보가 등록될 필요가 있다. 등록과 필요한 메모리를 할당하는 것은 start 메소드의 임무이다.

수행가능한(Runnable)

일단 여러분이 start 메소드를 호출하면 쓰레드는 수행가능한 상태이다. 수행가능한 쓰레드는 아직 수행될 수 없다. 쓰레드에게 수행할 기회를 주는 것은 운영체제에게 달려있다. 쓰레드내에 코드가 수행을 시작할 때 쓰레드는 작동(running)한다. (비록 자바 문서는 이것을 분리된 상태로 언급하지는 않았지만 수행하고 있는 쓰레드는 여전히 수행가능한 상태이다.)

이러한 일들이 발생하는 방법은 운영체제에 달려있다. 자바의 쓰레드 패키지는 하부의 운영체제와 작업할 필요가 있다. 오직 운영체제만이 CPU 사이클을 제공할 수 있다. 소위 "그린 쓰레드"라 불리는 패키지는 솔라리스상의 자바 1.0 에 의해서 사용된다. 예를 들면, 보다 높은 우선순위의 쓰레드가 깨어나서 컨트롤을 받을 때까지 수행중인 쓰레드의 활동을 유지한다. 윈도우 95 와 윈도우 NT 같은 다른 쓰레드 시스템들에서는 각각의 수행가능한 쓰레드에게 해당 작업을 수행하기 위한 일정한 시간을 제공한다. 할당받은 시간이 경과될 때 운영체제는 다른 쓰레드에게 작업할 기회를 제공한다. 이것은 보다 복잡하고 자바의 멀티드레딩 능력을 보다 좋게 사용하도록 만든다. 솔라리스용 다음 자바 버전에서는 시간 할당 기능을 수행하는 본래의 솔라리스 쓰레드들을 사용하리라고 예상한다.

항상 명심할 것은 수행가능한 쓰레드는 어떤 주어진 시간에 수행할 수도 아닐 수도 있다는 것이다. 그림 1-4 를 참조하라.(이것은 쓰레드의 상태를 "수행하는"이 아니라 "수행가능한"이라고 부르는 이유이다.)

그림 1-4 : 단일 CPU상에서의 시간 배분

블록된(Blocked)

다음 상황중 하나가 발생하면 쓰레드는 블록된 상태로 들어간다.:

1. 누군가 쓰레드의 sleep() 메소드를 호출한다.
2. 쓰레드가 입/출력상의 블록킹되는 연산을 호출한다. 즉, 연산은 입/출력 작업이 종료되고 나서야 호출자에게 반환된다.
3. 쓰레드가 wait() 메소드를 호출한다.
4. 쓰레드가 다른 쓰레드에 의해 lock 된 object 를 lock 하려 할 때. Object lock 은 이장의 뒷부분에서 언급 될 것이다.
5. 누군가 쓰레드의 suspend() 메소드를 호출한다. 그러나, 이 메소드는 deprecated method 이고 code 안에서 이 메소드를 부르면 안된다. 이장의 뒷 부분에서 이유를 설명하게 될 것이다.

그림 1-5 는 쓰레드가 갖을 수 있는 상태와 한 상태에서 다른 상태로의 가능한 전이를 보여준다. 쓰레드가 블록됐을 때 (또는 쓰레드가 종료됐을 때) 다른 쓰레드는 실행하기 위해 스케쥴된다. 블록된 쓰레드가 다시 수행될 때(예를 들면, 요구되는 밀리초동안 자고있거나 입/출력 연산이 완전해 질 때까지 기다리는 경우), 스케쥴러는 현재 수행되고 있는 쓰레드보다 높은 우선순위를 갖고 있는 지 검사한다. 만일 그렇다면 현재 쓰레드를 선점하고 블록된 쓰레드가 다시 수행을 시작한다.(다중 프로세스 머쉰상에서 각각의 프로세스는 분리된 쓰레드를 병렬적으로 수행한다. 그 경우에 쓰레드가 선점되는 경우는 높은 우선순위를 갖는 쓰레드가 runnable state 가 되고 실행할 프로세서가 없을 경우이다.)

그림 1-5 : 쓰레드 상태들

예를 들면, Ball 쓰레드의 run 메소드는 한번 완전히 이동한 다음에 5 밀리 초 동안 잠자고 있는 상태로 된다.

```
class Ball extends Thread
{       ....
        public void run()
        {       draw();
                for(int i = 1; i <= 1000; i++)
                {       move();
                        try { sleep(5); }
```

```
                    catch(InterruptedException e) {}
                }
            }
}
```

이것은 다른 쓰레드들에게(우리의 경우 다른 공들과 메인 쓰레드이다.) 실행할 기회를 제공한다. 만일 컴퓨터가 다중 프로세스 머쉰이라면 하나 이상의 쓰레드가 동시에 수행될 기회를 갖는다.

블록된 상태에서 다른 상태로의 이동

쓰레드는 블록된 상태로 몰아넣는 반대 절차를 통해 블록된 상태 바깥으로 이동하거나 수행가능한 상태로 되돌아 가야 한다.

1. 만일 쓰레드가 수면상태로 접어들면 명시된 밀리 초 숫자는 파기돼야 한다.
2. 만일 쓰레드가 입력이나 출력 연산이 끝나기를 기다리고 있으면 연산은 종료돼야 한다.
3. 만일 쓰레드가 wait 를 호출하였으면 다른 쓰레드가 반드시 notify 나 notifyAll 을 호출해야 한다.(이 장의 뒷부분에서 wait, notify/notifyAll 을 설명할 것이다)
4. 만일 쓰레드가 다른 쓰레드가 소유하는 object 의 lock 을 위해 대기중이면 object 의 소유 쓰레드가 object 의 lock 을 양보해야만 한다.(자세한 내용은 뒤에 언급 될 것이다)
5. 만일 쓰레드가 중단되면 누군가 해당 쓰레드의 resume 을 호출해야 한다. 그러나 suspend 메소드와 마찬가지로 resume 도 deprecated method 이므로 code 안에서 사용해서는 안된다.

노트 : 반드시 첫번째 장소에서 쓰레드를 블록시킨 경로와 일치하지 않는 방법으로만 블록된 쓰레드를 작동할 수 있다. 만일 쓰레드가 입력상에서 블록되었으면 쓰레드의 resume 메소드를 호출을 통해서는 블록상태를 벗어날 수 없다.

만일 여러분이 해당 쓰레드의 상태와 적합하지 않는 메소드를 호출하면 IllegalThreadStateException 이 투구된다. 예를 들면, 이러한 상황은 여러분이 현재 블록된 쓰레드의 suspend 메소드를 호출할 때 발생한다.

종료된 쓰레드

쓰레드가 종료되는 것은 다음 두가지 이유중 하나이다.

- run 메소드가 종료됐기 때문에 자연스럽게 종료한다.
- Uncaught exception 에 의해 run 메소드가 종료되어 갑작스레 종료한다.

Stop 메소드를 호출하여 쓰레드를 종료시킬 수 있다. Stop 메소드는 쓰레드를 종료시키는 ThreadDeath Error 객체를 투구한다. 그러나 stop 메소드는 사용해서는 안된다. 이장의 후반에서 왜 쓰레드의 stop 메소드가 위험한지 설명하게 될 것이다. 다음절은 어떻게 stop 메소드를 호출하지 않고 인터럽트를 발생하여 쓰레드를 종료하는지 설명할 것이다.

쓰레드의 상태 찾기

쓰레드가 현재 수행가능한지 블록되어 있는지 알기 위해서 isAlive 메소드를 사용한다. 이 메소드는 만일 쓰레드가 여전히 새것이고 아직 수행가능하지 않거나 종료되어 있으면 거짓을 반환한다.

노트 : 쓰레드가 수행가능한지, 블록되어 있는지 혹은 실제로 수행하고 있는지를 알 수는 없다. 또한 아직 수행가능하게 되지 않은 쓰레드와 이미 종료된 쓰레드 사이의 차이를 구별할 수 없다.

---

java.lang.Thread

- boolean isAlive()

만일 쓰레드가 시작하고 중단되지 않았으면 참을 반환한다.

- void suspend()

해당 쓰레드의 실행을 중단한다. Deprecated method

- void resume()

해당 쓰레드를 다시 실행한다. 이 메소드는 오직 suspend() 가 호출된 다음에만 쓸 수 있다. Deprecated method

- void stop()

쓰레드를 종료한다. Deprecated method

- void join()

쓰레드가 종료할 때까지 기다린다.

- void join(long millis)

쓰레드가 종료할 때까지 일정시간 동안 기다린다.

쓰레드 인터럽트

쓰레드는 run 메소드가 반환될 때 종료를 한다. Stop 메소드는 deprecated method 이므로 쓰레드를 종료 시킬 방법이 없다. 그러므로 run 메소드의 while loop 에서 종료를 해야 하는

지를 체크 해야 한다.

```
public void run()
{
        while ( no request to terminate && more work to do )
        {       do more work
        }
        // exit run method and terminate thread
}
```

위에서 살펴본 것 처럼 다른 쓰레드에게 일을 수행할 기회를 주기 위해서는 한 쓰레드가 계속해서 일을 수행해서는 안되고 sleep 이나 wait 를 통해 while loop 안에서 블록된 상태를 가져야 한다. 그러나 쓰레드가 블록된 상태에서는 현재 쓰레드를 종료해야 하는지를 체크 할 방법이 없다. 그래서 interrupt 메소드가 필요하다. 현재 블록된 쓰레드에서 interrupt 메소드가 불리면 sleep 이나 wait 와 같은 블록 함수가 InterruptedException 에 의해 종료된다.
인터럽트가 걸린 쓰레드가 반드시 종료를 해야 하는 것은 아니다. 쓰레드에 인터럽트를 거는 것은 단지 블록 된 쓰레드의 주의를 환기 시킬 뿐이다. 인터럽트된 쓰레드는 InterruptedException 을 처리하는 catch 블록에 적당한 action 을 지정함으로써 인터럽트에 어떤 반응을 보일지 결정할 수 있다. 매우 중요한 일을 수행하는 쓰레드들은 exception 을 무시하고 지나감으로써 인터럽트에 아무 반응을 하지 않는다. 하지만 쓰레드의 인터럽트를 종료에 대한 요구로 받아들이는 것이 아주 일반적인 경우이다. 그런 쓰레드들의 run 메소드는 아래와 같은 형식을 따른다.

```
public void run()
{
        try
        {
                while (more work to do )
                {       do more work
                }
        }
        catch(InterruptedException e)
        {       //thread was interrupted during sleep or wait
        }
        // exit run method and terminate thread
```

```
}
```

하지만 이런 코드 구조에는 약간의 문제가 있다. 만약 블록되지 않은 상태에서 interrupt 메소드가 호출되면 InterruptedException 이 발생하지 않는다. 따라서 쓰레드는 최근에 인터럽트가 걸렸는지 확인하기 위해 interrupted 메소드를 호출해야 한다.

```
while( !interrupted() && more work to do)
{           do more work
}
```

특히 입/출력에 작업에 의해 블록 되었을 때는 interrupt 메소드의 호출을 통해서 입/출력 작업을 종료 할 수 없기 때문에 블록 함수가 반환 되었을 때 최근에 인터럽트가 걸렸는지 확인하기 위해 interrupted 메소드를 호출해야 한다.

노트 : 흥미롭게도 매우 비슷한 interrupted 와 isInterrupted 메소드가 있다. Interrupted 메소드는 current thread 가 인터럽트 걸렸는지를 검사하는 정적 메소드이다. 그리고 interrupted 상태를 해제한다. 반면에 isInterrupted 메소드는 단지 인터럽트가 걸렸는지를 체크만 하는 인스턴스 메소드이다.

---

java.lang.Thread

- void interrupt()

쓰레드에 인터럽트 요구를 보낸다. 쓰레드의 "interrupted" 상태는 true 가 된다.
만약 쓰레드가 sleep 이나 wait 메소드의 호출로 블록 상태에 있으면 InterruptedException 이 발생된다.

- static boolean interrupted()

Current thread (메소드를 호출한 쓰레드) 가 인터럽트 걸렸는지를 검사한다. 또한 current thread 의 "interrupted" status 를 false 로 바꾼다.

- Boolean isInterrupted()

쓰레드가 인터럽트 걸렸는지를 검사한다. 이것은 interrupted 메소드와 달리 "interrupted" status 를 바꾸지 않는다.

쓰레드 우선순위

자바에는 모든 쓰레드는 우선순위를 가지고 있다. 기본적으로 쓰레드는 그것의 부모 쓰레드의 우선순위를 상속받는다. 여러분은 setPriority 메소드를 통해 특정 쓰레드의 우선순위를 증가하거나 감소할 수 있다. 여러분은 MIN_PRIORITY(Thread 클래스내에서 1

로 정의된다.)와 MAX_PRIORITY(10 으로 정의된다.)사이의 어떤 값이라도 우선순위로 설정할 수 있다. NORM_PRIORITY 는 5 로 정의된다.

쓰레드 스케쥴러가 새로운 쓰레드를 선택할 기회를 갖을 때마다 현재 수행가능한 가장 높은 우선순위의 쓰레드를 선택한다.

주의 : 쓰레드 우선순위에 관한 규칙은 매우 시스템에 종속적이다. 자바 가상 머쉰이 호스트 플랫폼의 쓰레드 구현에 종속적인한 쓰레드 스케쥴링은 그 쓰레드의 구현에 종속적이다.
자바 가상 머쉰은 호스트 플랫폼의 우선순위 등급 (자바 쓰레드 등급 보다 많거나 적은)에 매핑시킨다. 이 절에서는 모든 가상머쉰의 구현이 어떤 등급에 거의 근사하다고 보는 이상적인 경우를 설명한다.

가장 높은 우선 순위의 수행가능한 쓰레드는 다음중 하나의 경우에 해당할 때까지 계속 수행한다.

- yield 메소드를 호출하여 양보하거나
- 수행가능한 것을 중단하거나(종료하거나 블록된 상태로 진입하는 것에 의해)
- 보다 높은 우선 순위의 수행가능하게 된 쓰레드로 대체된다.(수면에서 깨거나 입/출력 연산이 완결됐기 때문에, 또는 누군가 notify 를 호출했기 때문이다.)

그런 다음 스케쥴러는 실행할 새로운 쓰레드를 선택한다. 실행가능한 쓰레드들중 남아있는 가장 높은 우선순위의 쓰레드가 선발된다.

만일 하나 이상의 쓰레드가 같은 우선 순위를 갖는다면 무슨 일이 발생할까? 가장 높은 우선순위의 쓰레드중 하나가 선발된다. 이것은 전적으로 동일한 우선순위의 쓰레드를 조정하는 쓰레드 스케쥴러에 달려있다. 자바는 모든 쓰레드가 공평하게 취급되어진다고 보장하지 못한다. 물론 같은 등급의 우선 순위를 갖는 쓰레드가 차례로 수행을 한다면 바람직한 일이다. 그러나 어떤 플랫폼에서는 쓰레드 스케쥴러가 무작위로 혹은 첫번째 가용 쓰레드를 선택하기 때문에 이론적으로만 가능하다. 이것이 자바의 약점이고 또한 모든 가상머쉰에서 동일하게 실행되는 멀티쓰레드 프로그램을 만드는 것은 어렵다.

주의 : 어떤 플랫폼에서는 자바에서 정해진 10 개의 등급보다 적은 우선순위가 있다. 특히 Windows NT 는 7 개의 우선순위 등급을 갖는다. 가상머쉰에서 어떻게 매핑을 하건 10 개의 JVM 등급 중 몇 개의 등급은 동일한 NT 등급에 매핑된다. 그래서 멀티쓰레드 프

로그램에서 우선순위에 의존할 수 없다.

Yield 의 호출에 대한 예를 고려해 보자. 아마도 어떤 구현의 경우는 전혀 영향을 미치지 않을 것이다. 쓰레드의 우선순위는 동일한 호스트의 우선순위에 매핑 될 지 모른다.
호스트의 쓰레드 스케쥴러는 공평하게 쓰레드를 선택하지 않을 수 도 있고 양보한 쓰레드를 다시 실행시킬 수 도 있다. 최소한 현재의 쓰레드가 곧 바로 다시 선택되어지는 것을 바라지 않는다면 차라리 sleep 메소드를 선택하는 것이 현명한 선택이다.

가상머쉰은 꼭 10 개일 필요는 없지만 몇 개의 우선순위 등급을 갖고 있고 쓰레드 스케쥴러는 가능하면 같은 우선순위의 쓰레드를 번갈아 선택하려고 노력한다.
예를 들어 앞의 예제에서의 공 쓰레드를 본다면 거의 비슷한 비율로 동작되어 지는 것처럼 보일 것이다. 물론 보장할 수는 없다.

만약 공평하게 스케쥴링하는 정책을 원한다면 스스로 구현을 해야 만 한다. 다음의 테스트 프로그램을 고려해보자. 이 프로그램은 이전의 프로그램에 하나의 공(붉은 공)의 쓰레드가 다른 쓰레드들 보다 높은 우선순위를 갖는 것을 허락한다. (굵은 글씨체로 써 있는 부분이 어떻게 쓰레드의 우선순위를 높이는가를 보여준다). 만일 Start 버튼을 선택하면 다섯 개의 쓰레드가 동일한 우선 순위를 갖고 검정 공을 움직이면서 구동 될 것이다. Express 버튼을 선택하면 일반적인 공들보다 높은 우선순위를 갖고 실행하는 쓰레드를 갖는 다섯 개의 붉은 공을 구동시킨다.

```
public BounceExpress
{ ….
        addButton( p, “Start”, new ActionListener()
        {       public void actionPerformed(ActionEvent e)
                {       for( int i=0; i<5 ; i++)
                        {       Ball b= new Ball( canvas, Color.black);
                                b.setPriority(Thread.NORM_PRIORITY);
                                b.start();
                        }
                }
        });
        addButton( p, “Express”, new ActionListener()
        {       public void actionPerformed(ActionEvent e)
                {       for( int i=0; i<5 ; i++)
                        {       Ball b= new Ball( canvas, Color.red);
                                b.setPriority(Thread.NORM_PRIORITY +2);
                                b.start();
                        }
                }
        });
}
```

이것을 수행해 보라. 몇몇의 정상적인 공들과 몇몇의 익스프레스 공을 구동시킨다. 여러분은 익스프레스 공들이 보다 빠르게 수행되는 것을 알 수 있을 것이다. 이것은 독자적으로 보다 높은 우선순위를 갖는 결과이다. 익스프레스 공들을 이동시키는 코드는 일반 공들을 이동시키는 코드와 같다.

다음은 어떻게 이러한 작업이 수행되는 지에 대한 이유이다. 익스프레스 쓰레드가 수면

상태에 들어간 5 초 후 깨어난다. 그런 다음 스케쥴러는 다시 모든 쓰레드들의 우선순위를 평가하고 보다 높은 우선순위를 갖는 익스프레스 쓰레드를 발견한다. 이렇게 해서 익스프레스 쓰레드는 다른 순서를 얻게 된다. 만일 여러분이 두개의 익스프레스 공들을 구동시키면(그림 1-6 처럼) 그들은 번갈아 가면 기회를 얻는다. 오직 그들이 자고 있는 동안 낮은 우선순위의 쓰레드들이 실행할 기회를 얻는다. 예제 1-3 의 코드를 참조하라. 만약 익스프레스 쓰레드가 sleep 대신 yield 를 호출했다면 우선순위가 낮은 쓰레드들은 실행할 기회를 갖지 못하게 됨에 주의하라.

이 프로그램은 NT 와 Solaris 에서는 잘 실행되지만 다른 곳에서는 장담할 수 없다.

TIP : 만약 code 가 작동하게 하기 위해서 어설프게 우선순위를 조작하면 잘 못 하고 있는 것이다. 대신 멀티쓰레드를 제어하기 위한 신뢰성있는 기법들을 더 배우기 위해 이 장의 나머지를 읽거나 인용된 참고 문헌들을 자세히 보는 것이 좋을 것 이다.

그림 1-6 : 서로 다른 우선 순위의 쓰레드들

예제 1-3 : BounceExpress.java

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

public class BounceExpress
{  public static void main(String[] args)
   {  JFrame frame = new BounceExpressFrame();
      frame.show();
   }
}

class BounceExpressFrame extends JFrame
{   public BounceExpressFrame()
   {   setSize(300, 200);
       setTitle("Bounce");

       addWindowListener(new WindowAdapter()
          {  public void windowClosing(WindowEvent e)
```

```
            {    System.exit(0);
            }
        } );

 Container  contentPane = getContentPane();
canvas = new JPanel();
 contentPane.add(canvas, "Center");
JPanel p = new JPanel();
 addButton(p, "Start",
       new ActionListener()
      {   public void actionPerformed(ActionEvent evt)
            {  for (int i = 0; i < 5; i++)
              {   Ball b = new Ball(canvas, Color.black);
                    b.setPriority(Thread.NORM_PRIORITY);
                    b.start();
                }
            }
       });

 addButton(p, "Express",
       new ActionListener()
      {   public void actionPerformed(ActionEvent evt)
        {   for (int i = 0; i < 5; i++)
              {   Ball b = new Ball(canvas, Color.red);
                  b.setPriority(Thread.NORM_PRIORITY + 2);
                    b.start();
                }
            }
       });

 addButton(p, "Close",
       new ActionListener()
      {   public void actionPerformed(ActionEvent evt)
            {   canvas.setVisible(false);
                System.exit(0);
            }
```

```
            });
        contentPane.add(p, "South");
    }

  public void addButton(Container c, String title,
        ActionListener a)
  {  JButton b = new JButton(title);
        c.add(b);
        b.addActionListener(a);
    }

   private JPanel canvas;
}

class Ball extends Thread
{  public Ball(JPanel b, Color c) { box = b; color = c; }

   public void draw()
   {  Graphics g = box.getGraphics();
        g.setColor(color);
      g.fillOval(x, y, XSIZE, YSIZE);
        g.dispose();
    }

   public void move()
    {   if (!box.isVisible()) return;
      Graphics g = box.getGraphics();
        g.setXORMode(box.getBackground());
        g.setColor(color);
      g.fillOval(x, y, XSIZE, YSIZE);
       x += dx;
       y += dy;
      Dimension d = box.getSize();
      if (x < 0)
      { x = 0; dx = -dx; }
     if (x + XSIZE >= d.width)
```

```
        { x = d.width - XSIZE; dx = -dx; }
        if (y < 0)
       { y = 0; dy = -dy; }
       if (y + YSIZE >= d.height)
        { y = d.height - YSIZE; dy = -dy; }
        g.fillOval(x, y, XSIZE, YSIZE);
          g.dispose();
     }

    public void run()
     {   try
          {   draw();
           for (int i = 1; i <= 1000; i++)
               {   move();
                   sleep(5);
               }
          }
         catch(InterruptedException e) {}
     }

    private JPanel box;
     private static final int XSIZE = 10;
   private static final int YSIZE = 10;
    private int x = 0;
    private int y = 0;
    private int dx = 2;
    private int dy = 2;
     private Color color;
}
```

java.lang.Thread

- void setPriority (int newPriority)

해당 쓰레드의 우선순위를 설정한다. Thread.MIN_PRIORITY **와** Thread.MAX_PRIORITY **사이의 값이여야 한다**. **일반 우선순위를 위해** Thread.NORM_PRIORITY **를 사용하라**.

- static int MIN_PRIORITY

쓰레드가 갖을 수 있는 최대 우선순위이다. **최소 우선순위는** 1 **이다**.

- static int NORM_PRIORITY

쓰레드의 기본 우선순위이다. 기본 우선순위의 값는 5이다.

- static int Max_PRIORITY

쓰레드가 갖을 수 있는 최대 우선순위이다. 최대 우선순위의 값은 10이다.

- static void yield()

현재의 쓰레드가 양보를 하도록 한다. 만약 현재 쓰레드 다음의 우선순위를 갖는 쓰레드가
있다면 쓰레드 스케듈의 다음 대상은 그 쓰레드이다. 이것은 정적 메소드임에 주의.

탐욕(selfish) 쓰레드

우리의 공 쓰레드들은 서로 잘 협력하며 행동한다. 그것들은 이를 위해 자신의 차례를 기다리기 위한 sleep 함수를 사용한다. Sleep 함수는 쓰레드를 블록하고 다른 쓰레드들이 스케줄되기 위한 기회를 준다. 비록 쓰레드가 일정한 시간동안 자체적으로 수면상태에 입수하길 원하지 않더라도 인터럽트되길 원하지 않을 때마다 yield()를 호출할 수 있다. 쓰레드는 항상 긴 루프를 수행할 때 시스템을 독점하지 않기 위해 sleep 이나 yield 를 호출해야 할 것이다. 이러한 규칙을 따르지 않는 쓰레드를 “탐욕 쓰레드”라고 부른다.

다음의 프로그램은 쓰레드가 타이트한 루프를 포함하고 있을 때 무슨 일이 발생하는 지를 보여준다. 루프는 다른 쓰레드에게 기회를 주지않고 많은 작업을 수행한다. 여러분이 탐욕 버튼을 선택할 때 이 타이트한 루프를 포함하고 있는 run 메소드를 갖는 푸른 공을 구동시킨다.

```
class SelfishBall extends Ball
{       . . .
        public void run()
        {       draw();
                for(int i = 1; i <= 1000; i++)
                {       move();
                        long t = new Date().getTime();
                        while(new Date().getTime() < t + 5);
                }
        }
}
```

run 메소드는 반환되기 전에 5 초 동안 지속되다 쓰레드를 종료할 것이다. 이 동안 결코 sleep 이나 yield 를 호출하지 않는다.

여러분이 이 프로그램을 수행시킬 때 여러분의 운영체제에 따라 실제 어떤 일이 발생할까? 예를 들면, 솔라리스상에서 이 프로그램을 수행시켰을 때 여러분은 탐욕스런 공이 정말로 전체 프로그램을 차지하는 것을 발견하게 될 것이다. 프로그램을 종료하거나 다른 공을 구공시키려고 해보라. 여러분은 해당 프로그램내에서 마우스를 클릭하는 것도 어려울 것이다. 그러나 여러분이 윈도우 95 나 NT 하에서 같은 프로그램을 수행할 때 앞선 상황은 발생하지 않는다. 푸른 공은 다른 공들과 병렬로 수행될 것이다.

그 이유는 윈도우상의 쓰레드 패키지하에서는 "타임 슬라이싱(time slicing)" 을 지원하기 대문이다. 이것은 비록 쓰레드들이 서로 협력하지 않더라도 중간에 주기적으로 쓰레드들을 인터럽트해준다. 하나의 쓰레드가 인터럽트될 때 다른 쓰레드(수행가능한 쓰레드들사이의 가장 높은 우선순위를 갖는)가 수행된다. 솔라리스상의 자바 1.0 에 의해서 사용되는 "그린 쓰레드" 구현은 타임 슬라이싱을 지원하지 않는다. 그러나 향후 버전에서는 지원할 것이다. 만일 여러분이 여러분의 프로그램이 타임 슬라이싱을 지원하는 운영체제를 갖는 머쉰상에서 수행되는 것을 안다면 여러분의 쓰레드를 공손하게 만들기 위해 걱정할 필요가 없다. 그러나 인터넷 컴퓨팅 관점에서 일반적으로 여러분의 프로그램을 사용하는 사람들의 환경을 알지 못한다. 그러므로 여러분은 최악의 상황에 대비하기 위해 모든 루프안에 sleep 이나 yield 호출을 삽입해야 한다. 예제 1-4 를 참조하라.

예제 1-4 : BounceSelfish.java

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

public class BounceSelfish
{  public static void main(String[] args)
   {  JFrame frame = new BounceSelfishFrame();
      frame.show();
   }
}

class BounceSelfishFrame extends JFrame
{  public BounceSelfishFrame()
   {  setSize(300, 200);
```

```
 setTitle("Bounce");

 addWindowListener(new WindowAdapter()
    {  public void windowClosing(WindowEvent e)
         {  System.exit(0);
         }
     } );

Container contentPane = getContentPane();
canvas = new JPanel();
 contentPane.add(canvas, "Center");
JPanel p = new JPanel();
 addButton(p, "Start",
      new ActionListener()
    {  public void actionPerformed(ActionEvent evt)
       {  Ball b = new Ball(canvas, Color.black);
            b.setPriority(Thread.NORM_PRIORITY);
            b.start();
       }
      });

 addButton(p, "Selfish",
      new ActionListener()
    {  public void actionPerformed(ActionEvent evt)
       {  Ball b = new SelfishBall(canvas, Color.blue);
          b.setPriority(Thread.NORM_PRIORITY + 2);
            b.start();
       }
      });

 addButton(p, "Close",
      new ActionListener()
    {  public void actionPerformed(ActionEvent evt)
         {  canvas.setVisible(false);
            System.exit(0);
         }
```

```
            });
        contentPane.add(p, "South");
    }

   public void addButton(Container c, String title,
        ActionListener a)
   {  JButton b = new JButton(title);
        c.add(b);
        b.addActionListener(a);
    }

   private JPanel canvas;
}

class Ball extends Thread
{  public Ball(JPanel b, Color c) { box = b; color = c; }

   public void draw()
   {  Graphics g = box.getGraphics();
        g.setColor(color);
       g.fillOval(x, y, XSIZE, YSIZE);
        g.dispose();
    }

   public void move()
   {  if (!box.isVisible()) return;
       Graphics g = box.getGraphics();
        g.setColor(color);
        g.setXORMode(box.getBackground());
       g.fillOval(x, y, XSIZE, YSIZE);
       x += dx;
       y += dy;
        Dimension d = box.getSize();
       if (x < 0)
       { x = 0; dx = -dx; }
       if (x + XSIZE >= d.width)
```

```
        { x = d.width - XSIZE; dx = -dx; }
        if (y < 0)
       { y = 0; dy = -dy; }
       if (y + YSIZE >= d.height)
        { y = d.height - YSIZE; dy = -dy; }
        g.fillOval(x, y, XSIZE, YSIZE);
          g.dispose();
     }

    public void run()
     {   try
          {   draw();
           for (int i = 1; i <= 1000; i++)
               {   move();
                   sleep(5);
               }
          }
         catch(InterruptedException e) {}
     }

    private JPanel box;
   private static final int XSIZE = 10;
   private static final int YSIZE = 10;
    private int x = 0;
    private int y = 0;
    private int dx = 2;
    private int dy = 2;
     private Color color;
}

class SelfishBall extends Ball
{  public SelfishBall(JPanel b, Color c) { super(b, c); }

    public void run()
     {   draw();
       for (int i = 1; i <= 1000; i++)
```

```
        {    move();
             long t = System.currentTimeMillis();
             while (System.currentTimeMillis() < t + 5)
                  ;
        }
    }
}
```

쓰레드 그룹들

몇몇 프로그램들은 많은 쓰레드들을 포함하고 있다. 그것들을 기능에 의해 분류하는 것은 유용하게 해준다. 예를 들면, 인터넷 부라우져를 생각해보자. 만일 많은 쓰레드들이 서버로 부터 이미지들을 얻기 위해 노력하고 사용자가 현재 로딩하고 있는 페이지를 중단하는 중단 버튼을 누르면 이러한 모든 쓰레드들은 동시에 종료시키는 방법을 가지는 것은 편리하다.
자바는 이 기능을 지원하기 위해 thread group 을 제공한다.

생성자를 통해 쓰레드 그룹을 생성한다.

```
String groupName = ...;
Thread Group g = new ThreadGroup(groupName);
```

문자열은 그룹의 이름을 의미하며 유일해야 한다. 쓰레드의 생성시에 쓰레드 그룹을 명시 함으로써 쓰레드 그룹에 쓰레드를 추가할 수 있다.

```
Thread t = new Thread ( g, threadName );
```

특정한 그룹의 쓰레드가 여전히 수행가능 한지 아닌지를 발견하기 위해 activeCount 메소드를 사용한다.

```
if(g.activeCount() == 0)
{          // 그룹 g 내의 모든 쓰레드들은 중단되어 있다.
}
```

쓰레드 그룹내의 모든 쓰레드들을 인터럽트 걸기 위해 간단하게 그룹 객체의 interrupt 메소드를 호출한다.

```
g.interrupt();          // g 의 모든 쓰레드들을 인터럽트 건다.
```

쓰레드 그룹들은 하위 서브 그룹들을 갖을 수 있다. 기본적으로 새롭게 생성된 쓰레드 그룹은 현재 쓰레드 그룹의 자식이다. 그러나 여러분은 극명하게 생성자내에서 부모 그룹의 이름을 명시할 수 있다.(API 노트 참조). activeCount 나 interrupt 같은 메소드들은 그들 그룹내과 모든 자식 그룹들내의 모든 쓰레드들을 참조한다.

java.lang.ThreadGroup

- ThreadGroup(String name)

새로운 쓰레드 그룹을 생성한다. 부모 쓰레드 그룹은 현재 쓰레드의 쓰레드 그룹이된다.

전달인자:

name 새로운 쓰레드 그룹의 이름

- ThreadGroup(ThreadGroup parent , String name)

새로운 쓰레드 그룹을 생성한다.

전달인자:

parent 새로운 쓰레드 그룹의 부모 쓰레드 그룹

name 새로운 쓰레드 그룹의 이름

- int activeCount()

쓰레드 그룹내의 활동하고 있는 쓰레드들의 수를 위한 상위 경계를 반환한다.

- int enumerate(Thread list[])

해당 쓰레드 그룹내의 활동하고 있는 모든 쓰레드들의 참조를 얻는다. 여러분은 배열의 갯수를 예측하기 위해 activeCount() 메소드를 사용할 수 있다. 이 메소드는 배열내에 쓰레드의 참조를 채우고 쓰레드의 수를 반환한다.

전달인자:

list 쓰레드 참조들로 채워질 배열

- ThreadGroup gerParent()

해당 Thread 그룹의 부모를 얻는다.

- void interrupt()

해당 쓰레드 그룹과 모든 자식 그룹들내의 모든 쓰레드들을 인터럽트 건다.

java.lang.Thread

- Thread(ThreadGroup g, String name)

해당 쓰레드그룹에 속하는 쓰레드를 생성한다.

전달인자:

g 새로운 쓰레드가 속할 쓰레드 그룹

name 새로운 쓰레드의 이름

- ThreadGroup getThreadGroup()

해당 쓰레드의 쓰레드 그룹을 반환한다.

동기화

수행중인 쓰레드는 참조를 갖고 있는 어떤 객체에 대해서도 접근할 수 있다. 만일 두개의 쓰레드가 같은 객체에 접근하고 각각 객체의 상태를 변경시키는 메소드를 호출 호출할 때 무슨 일이 발생할까? 여러분이 상상하는 것처럼 쓰레드들은 불일치된 상태의 객체들을 이끌어내면서 각각 자신의 작업을 수행할 수 있다. 이러한 상황을 경쟁 조건(race condition)이라 한다.

동기화 없이 쓰레드간 통신

이 문제를 풀기위해 우리는 먼저 공유하는 객체들에 동기화된 접근을 하는 방법을 배워야 한다. 이 절에서 우리는 만일 동기화 방법을 사용하지 않을 때 무슨 일이 발생하는지를 살펴본다. 그리고 다음 절에서 우리는 자바에서 동기화된 객체에 접근하는 방법을 배우게 된다.

다음 테스트 프로그램에서 우리는 10 개의 계정들 갖는 은행을 시뮬레이션할 것이다. 우리는 임의대로 이러한 계정들사이의 돈을 이체하는 트렌젝션을 생성해낼 것이다. 여기에는 각각의 계정마다 하나씩 10 개의 쓰레드가 존재할 것이다. 각각의 트렌젝션은 임의대로 일정한 양의 돈을 쓰레드에 의해서 제공되는 계정으로 부터 다른 임의 계정으로 이체시킬 것이다.

시뮬레이션 코드는 간단하다. 우리는 transfer 메소드를 갖는 Bank 클래스를 작성한다. 이 메소드는 일정한 양의 돈을 한 계정에서 다른 계정으로 전송한다. 만일 본래의 계정이 전송할 만큼 충분한 돈을 갖고 있지 않다면 transfer 메소드를 호출하는 쓰레드는 바로 함수를 반환한다.

다음은 Bank 클래스의 transfer 메소드를 위한 코드이다.

```
public void transfer(int from, int to, int amount)
{       // 주의 : 여러 개의 쓰레드가 동시에 부를 때 안전하지 않다.
        if(accounts[from] < amount) return;
        account[from] -= amount;
        account[to] += amount;
        ntransacts++;
        if(ntransacts % NTEST== 0) test();
}
```

다음은 TransferThread 클래스이다. Run 는 계속해서 특정한 계정에서 계정으로 돈을 나른다. 또한 run 메소드안에서 원하는 계정을 임의로 선택하고 또한 돈의 양도 임의로 정한 뒤 bank object 의 tranfer 메소드를 부르고 수면을 취한다.

```
class TransferThread extends Thread
{
	public TransferThread(Bank b, int from, int max)
	{
		bank=b;
		fromAccount=from;
		maxAmount=max;
	}

	public void run()
	{
		try
		{
			while(!interrupted())
			{
				int toAccount=(int)(bank.size()*Math.random());
				int amolunt=(int)(maxAmount*Math.random());
				bank.transfer(fromAccount,toAccount,amount);
				sleep(1);
			}
		}
		catch(InterruptedException e)
		{
		}

		private Bank bank;
		private int fromAccount;
		private int maxAccount;
	}
}
```

이 시뮬레이션을 수행할 때 우리는 항상 어떤 은행 계정에 얼마나 많은 돈이 있는지 알지 못한다. 그러나 우리는 변하지 않고 남아있는 모든 계정의 총합계 금액을 알 수 있다. 왜냐하면 우리가 한일은 한 계정에서 다른 계정으로 돈을 이체시키는 것이기 때문이다.

매 10,000 트렌젝션마다 transfer 메소드는 합계를 계산하고 이를 출력하는 test 메소드를 호출한다.

이 프로그램은 결코 중단하지 않는다. 단지 CTRL-C를 눌러 프로그램을 종료해야 한다.

다음은 일반적인 출력결과이다.

```
Transction:10000 Sum: 100000
Transction:20000 Sum: 100000
Transction:30000 Sum: 100000
Transction:40000 Sum: 100000
Transction:50000 Sum: 100000
Transction:60000 Sum: 100000
Transction:70000 Sum: 100000
Transction:80000 Sum: 100000
Transction:90000 Sum: 100000
Transction:100000 Sum: 100000
Transction:110000 Sum: 100000
Transction:120000 Sum: 100000
Transction:130000 Sum: 94792
Transction:140000 Sum: 94792
Transction:150000 Sum: 94792
```

여러분이 볼 수 있듯이 무엇인가 아주 잘못되어 있다. 수 천번의 트렌젝션동안 은행 합계가 $100,000 으로 남아있다. 이것은 10 개의 계정이 각각 $10,000 를 갖기때문에 정확한 합계이다. 그러나 몇 번 더 지나고 나면 합계는 약간씩 변한다. 이것은 확신을 주지 못한다. 그리고 우리는 아마 고생해서 벌은 우리의 돈을 이 은행에 맡기려고 원하지 않을 것이다.

다음 예제 1-5 는 완전한 소스 코드이다. 만일 여러분이 코드에 대해 문제를 발견할 수 있는지 살펴보라. 우리는 다음 절에서 이 미스테리를 풀 것이다.

예제 1-5 : UnsynchBankTest.java

```
public class UnsynchBankTest
{  public static void main(String[] args)
   {  Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
      int i;
      for (i = 0; i < NACCOUNTS; i++)
      {  TransferThread t = new TransferThread(b, i,
            INITIAL_BALANCE);
         t.setPriority(Thread.NORM_PRIORITY + i % 2);
         t.start();
      }
   }

   public static final int NACCOUNTS = 10;
   public static final int INITIAL_BALANCE = 10000;
}

class Bank
{  public Bank(int n, int initialBalance)
   {  accounts = new int[n];
      int i;
      for (i = 0; i < accounts.length; i++)
         accounts[i] = initialBalance;
      ntransacts = 0;
   }

   public void transfer(int from, int to, int amount)
   {  if (accounts[from] < amount) return;
      accounts[from] -= amount;
      accounts[to] += amount;
      ntransacts++;
      if (ntransacts % NTEST == 0) test();
   }
```

```
    public void test()
    {   int sum = 0;

        for (int i = 0; i < accounts.length; i++)
            sum += accounts[i];

        System.out.println("Transactions:" + ntransacts
           + " Sum: " + sum);
     }

    public int size()
     {   return accounts.length;
     }

   public static final int NTEST = 10000;
    private int[] accounts;
   private long ntransacts = 0;
}

class TransferThread extends Thread
{  public TransferThread(Bank b, int from, int max)
    {   bank = b;
        fromAccount = from;
        maxAmount = max;
     }

    public void run()
     {   try
         {   while (!interrupted())
            {   int toAccount = (int)(bank.size() * Math.random());
               int amount = (int)(maxAmount * Math.random());
                 bank.transfer(fromAccount, toAccount, amount);
                   sleep(1);
              }
          }
        catch(InterruptedException e) {}
```

```
    }

    private Bank bank;
    private int fromAccount;
    private int maxAmount;
}
```

노트 : 이 프로그램에서 절반의 쓰레드의 우선 순위등급을 1 씩 올려 맞추었다. (나머지는 정상적인 우선순위 등급). 윈도우 98 에서는 이렇게 하면 쓰레드끼리 서로 충돌하기 더 좋은 조건이어서 transfer 메소드의 에러를 더욱 가속화한다. 하지만 여러분의 플랫폼에서는 아무런 영향을 끼치지 못 할지 모른다.

공유자원에 대한 동기화 접근

앞 절에서 우리는 몇몇 쓰레드들이 은행 계정의 평균을 갱신하는 프로그램을 수행해보았다. 수천 번의 트렌젝션후에 에러가 잠복했으며 일정한 양의 돈이 분실되거나 자연발생하였다. 이 문제는 두개의 쓰레드들이 동시에 계정에 갱신을 시도했기 때문에 발생한다.

두개의 쓰레드들이 동시에 다음 기능을 수행한다고 가정하자.

```
Accounts[to]  += amount;
```

이 문제는 원자(atomic) 연산이 아니다.(주: 원자란 더 이상 나눠지지 않는 것을 말한다.) 이 연산은 다음과 같이 처리된다.

1. 레지스터에 accounts[to]을 로드한다.
2. amount 를 더한다.
3. 그 결과를 다시 account[to]에 반환한다.

지금, 첫번째 쓰레드가 과정 1 과 2 를 수행한다고 가정하자. 그리고 나서 인터럽트된다. 두번째 쓰레드가 수행되고 account 배열의 같은 항목에 갱신을 한다고 가정하자. 그런 다음 첫번째 쓰레드가 깨어나고 과정 3 을 마친다.

이것은 다른 쓰레드에 의한 갱신을 간과하는 것이다. 그 결과 총 합계는 더이상 정확한 것이 아니다 (그림 1-7 참조).

우리의 테스트 프로그램은 이 부정확성을 감지한다.(물론 , 여기에는 만일 쓰레드가 테스트를 수행하다가 인터럽트되면 잘못됐다는 신호를 주는 기회가 있다. !)

그림 1-7 : 두개의 쓰레드에 의한 동시 접근

노트 : 여러분은 우리의 클래스내의 각 문장이 실행되는 자바 바이트 코드를 실제로 살펴볼 수 있다. Bank.class 파일을 역컴파일하기 위해 다음의 명령어를 수행하라.

```
javap -c -v Bank
```

예를 들면, 다음 라인은

```
accounts[to]  +=  amount;
```

다음의 바이트 코드로 전환된다.

```
aload_0
getfield #16 <Field Bank.accounts [J>
iload_1
dup2
laload
iload_3
i2l
lsub
lastore
```

이 코드가 무엇을 의미하는 지는 중요하지 않다. 중요한 점은 증가 명령은 몇 개의 명령어로 이루어져 있다는 것이다. 그리고 그 명령어들을 수행하는 쓰레드는 어떤 명령어를 수행하는 지점에서도 인터럽트 될 수 있다.

오류가 발생하는 기회란 무엇인가? 만약 쓰레드가 비슷한 우선순위를 갖는다면 쓰레드가 재수면을 하기 전에 스케쥴러가 다른 쓰레드에게 선점의 기회를 제공하지 않을 정도로 아주 적은 양의 일을 하므로 다른 쓰레드와의 충돌에 의해 오류가 발생할 기회는 덜하다.
우리는 실험을 통해 절반의 쓰레드에게 나머지 보다 높은 우선 순위를 제공할 때 오류의 확률이 더 켜진다는 것을 알았다.

쓰레드의 우선순위가 높은 쓰레드가 수면에서 깨어났을 때 우선순위가 낮은 쓰레드를 선점하기 때문이다.

노트 : 이것은 단지 쓰레드의 우선 순위를 다룰 때를 말하는 것이지 꼭 그래서는 안된다는 것을 말하는 것은 아니다. 데이터의 오류가 발생하는 것을 보이려고 했던 것이다. 오류가 꼭 발생한다고 볼 수는 없지만 일반적인 플랫폼에서 우선순위를 조정함으로써 오류 발생의 기회가 많아 진다는 것을 알았다. 여러분의 프로그램에서 오류 발생의 기회를 높이지 않으려면 위와 같이 해서는 안된다.

문제는 transfer 메소드의 중간에 인터럽트가 발생할 수 있다는 것이다.
우리는 일단 쓰레드가 한 구성요소를 버퍼내에 삽입하기 시작했다는 것을 확신하기 위한 몇가지 방법을 갖고 있어야 한다. 이것은 쓰레드가 붕괴되지 않고 연산이 종료되게 할 수 있다. 대부분의 프로그래밍 환경들은 프로그래머에게 자원에 대한 접근시 인터럽트 되지않도록 이른바 세마포어(Semaphore)와 임계영역(Critical Section)이라 불리는 기술을 강제한다. 이것은 고통스러운 과정이고 에러도 발생하기 쉽다. 자바는 토니 호러(Tony Hoare)에 의해서 고안된 *"모니터(monitors)"* 라는 방법에서 아이디어를 얻은 보다 깔끔한 방법을 가지고 있다.

여러분은 다음과 같이 Synchronized 라는 것을 통해 인터럽트 받지 말아야 할 메소드를 지정할 수 있다.

```
public synchronized void transfer(int from , int to, int amount)
{           ....
            accounts[from]    -= amount;
            accounts[to] += amount;
            ntransacts++;
            if(ntransac % NTEST == 0) test();
}
```

하나의 쓰레드가 동기화된 메소드에 진입할 때 자바는 다른 쓰레드가 같은 객체상의 어떤 동기화된 메소드를 실행하기 전에 종료하는 것을 보장해준다. (그림 1-8 참조)
하나의 쓰레드가 transfer 를 호출하고 나서 다른 쓰레드가 transfer 를 호출하면 두번째 쓰레드는 계속 수행할 수 없다. 대신, 두번째 쓰레드는 활동할 수 없고 첫번째 쓰레드가 transfer 메소드의 실행을 종료할 때까지 기다려야 한다.

그림 1-8 : 비동기화 된 쓰레드와 동기화 된 쓰레드의 비교

Transfer 메소드를 동기화(synchronized 키워드를 추가함)하고 실행을 시키면 더 이상 오류가 생기지 않음을 발견할 것이다.

객체의 락(Lock)

쓰레드가 동기화 된 메소드를 호출할 때 객체는 "락(locked)"된다. 각각의 객체가 문 앞에 열쇠를 갖고 있다고 생각해보자. 처음에 열쇠는 문턱에 있다. 쓰레드가 동기화 메소드 안으로 들어왔을 때 열쇠를 갖고 객체안으로 들어가서 안에서 문을 잠근다. 다른 쓰레드가 동일한 객체의 같은 동기화 메소드를 호출했을 때 이제는 문을 열수가 없다. 그러면 열쇠를 찾을 것이고 열쇠를 찾을 수 없어서 곧 동작을 멈출 것이다. 결국은 첫번째 쓰레드가 동기화 함수를 빠져 나오고 객체를 떠나면서 열쇠를 다시 문턱에 놓아둔다.

주기적으로 스케쥴러는 열쇠를 기다리는 쓰레드를 앞서 언급했던 활성화 규칙에 따라 활성화 시킨다. 객체를 사용하기 원하는 쓰레드가 재기동 될 때 마다 객체가 "락(locked)"되어 있는지를 검사하고 "락(locked)" 되어 있지 않으면 다음 쓰레드가 객체를 독점한다.

하지만 다른 쓰레드들은 여전히 "락(locked)"된 객체일지라도 비동기화된 메소드를 호출할 수 있다. 예를 들면, BankAccount 클래스의 size 메소드는 동기화될 필요가 없다.

TIP : 생성시 외에는 변화할 필요가 없는 인스턴스 변수들을 선언할 때 final 로 선언하는 것은 매우 좋은 생각이다. 예를 들면 위에서는 accounts 변수를 final 로 정의했다.

```
private final int[] accounts;
```

만약 final 로 선언된 변수의 상태를 보는 메소드를 갖고 있다면 확실히 synchronized 를 생략할 수 있다.

만일 쓰레드가 예외 상황을 발생함으로써 동기화된 메소드를 떠날 때 여전히 객체의 락을 포기해야 한다. 이것은 좋은 방안이다- 여러분은 쓰레드가 동기화된 메소드를 종료한 다음에 객체를 붙잡고 있길 원치 않는다.

만일 쓰레드가 특정 객체의 락을 소유하고 있고 같은 객체의 다른 동기화된 메소드를 호출한다면 자동적으로 접근은 승인된다. 쓰레드는 오직 최종 동기화 메소드를 종료할 때 락을 내놓게 된다.

노트 : 기술적으로 각 객체는 락의 소유자가 동기화된 메소드드들을 얼마나 많이 호출했는지를 기록하는 *락 카운트*를 가지고 있다. 매 번 새로운 동기화 메소드를 호출할 때마다 락 카운트가 증가한다. 동기화된 메소드가 중단될 때마다 락 카운트는 감소한다. 락 카운트가 0에 도달할 때 쓰레드는 락을 포기한다.

쓰레드는 한 번에 복수 객체들의 락을 갖을 수 있다. 그저 다른 객체의 동기화 메소드를 실행하는 동안 특정 객체의 동기화 메소드를 호출하면 된다. 그러나 물론 객체의 락은 오직 특정 시점에 한 쓰레드가 소유할 수 있다.

wait와 notify 메소드들

우리의 은행 시뮬레이션을 재작성해 보자. 우리는 전송할 만한 금액을 갖지 않는 계정으로 부터 돈이 전송되길 원치 않는다. 다음과 같이 코드를 사용할 수 없다는 점에 주목하라.:

```
if ( bank.getBalance(from) >= amount)
        bank.transfer(from, to, amount);
```

이 코드는 현재 쓰레드가 성공적인 검사 결과와 transfer 를 호출하는 사이에 중단될 가능성이 있다.

```
if ( bank.getBalance(from) >= amount)
        // 쓰레드가 이 시점에서 중단될 가능성이 있다.
        bank.transfer(from, to, amount);
```

쓰레드가 다시 수행될 때 계정의 합계는 amount 아래로 떨어질 것이다. 우리는 검사와 추가사이에 쓰레드가 중단되지 않도록 해야 한다. 여러분은 이것을 동기화된 transfer 메소드내에 검사하도록 추가할 수 있다:

```
public synchronize void transfer ( int from , int to, int amount)
{       while(accounts[from] < amount)
        { // 대기
```

```
            }
            // 자금 전송
        }
```

계정에 충분한 돈이 없을 때 우리가 할 일은 무엇인가? 우리는 몇몇 다른 쓰레드가 돈을 추가할 때까지 기다린다. 그러나 transfer 메소드는 synchronized 되어 있다. 이 쓰레드는 단지 은행 객체에 대한 배타적인 접근권한을 얻는다. 그래서 다른 쓰레드는 예금할 수 있는 기회를 갖을 수 없다. 두번째 동기화된 메소드들의 기능은 이 상황을 처리하는 것이다.
만일 여러분이 동기화된 메소드내부에서 대기할 필요가 있다면 쓰레드 클래스내에서 wait 메소드를 사용한다.

동기화된 메소드내에서 wait 가 호출될 때 현재의 쓰레드는 블록되고 객체 락을 포기한다. 이것은 다른 쓰레드가 계좌 잔액를 증가하게 해준다. Wait 메소드는 Thread 클래스가 아니라 Object 클래스의 메소드라는 사실을 기억하라. Wait 가 호출될 때 Bank 객체는 현재 쓰레드를 블록하고 스스로 언락(unlock) 한다.

동기화된 메소드내에서 대기하고 있는 쓰레드와 wait 를 호출하는 쓰레드사이에는 기본적인 차이점이 존재한다. 일단 쓰레드가 wait 메소드를 호출하면 대기 목록(wait list)에 포함된다. 쓰레드가 대기 목록에서 제거될 때까지 스케줄러는 해당 쓰레드를 무시하고 해당 쓰레드는 계속 수행할 기회를 갖질 못한다.

대기 목록에서 쓰레드를 제거하기 위해 몇몇 다른 쓰레드는 같은 객체상의 notify 나 notifyall 메소드를 호출해야 한다. notify 메소드는 대기 목록에서 임의로 선택한 하나의 쓰레드를 제거한다. NotifyAll 메소드는 대기 목록에 있는 모든 쓰레드를 제거한다. 쓰레드들이 대기 목록에서 제거된 다음 스케쥴러는 최종적으로 반복해서 이들 쓰레드를 작동시킬 것이다. 이 시점에서 쓰레드들은 객체에 대한 재진입을 시도할 것이다. 객체 락을 이용할 수 있을 때 쓰레드중의 하나는 객체로 재진입할 것이고 나머지는 계속해서 wait 을 호출한 다음에 남게 된다.

몇몇 다른 쓰레드가 주기적으로 notify 나 notufyAll 메소드를 호출하는 것은 아주 중요하다. 한 쓰레드가 wait 을 호출할 때 그 자체를 언블록킹할 방법은 없다. 만일 쓰레드중의 어느 것도 대기중인 쓰레드의 언블록을 간섭하지 않는다면 결코 다시 수행되지 않을 것이다. 이것은 결코 바라지않는 데드락 상황을 발생시킬 수 있다. 만일 모든 다른 쓰레드들이 블록되고 최종 실행된 쓰레드가 다른 쓰레드들중의 하나를 언블록킹하지 않고 wait 을 호출한다면 이 또한 블록된다. 다른 쓰레드들을 언블록으로 남겨놓고 프로그램을

중단시키는 쓰레드는 존재하지 않는다. 대기중인 쓰레드들은 해당 객체상에서 작동중인 다른 쓰레드가 없을 때 자동적으로 재작동되지 않는다. 우리는 이 장의 뒷부분에서 데드락에 대해 논의할 것이다.

실제로, 언블록된 쓰레드를 조정할 수 없기 때문에 notify 를 호출하는 것은 위험하다. 만일 잘못된 쓰레드가 언블록되면 쓰레드는 진행되지 못할 것이다. 우리는 notifyAll 메소드를 사용하고 모든 쓰레드들을 언블록하길 추천한다.

언제 notifyAll 을 호출할 것인가? 기본 규칙은 객체의 상태가 기다리고 있는 쓰레드들에게 유리하게 변할 때마다 notifyAll 을 호출하는 것이다. 예를 들면, 계정 잔액이 변할 때마다 대기중인 쓰레드는 잔액을 검사할 다른 기회를 제공받는다. 우리의 예제에서, 우리는 자금 전송을 마쳤을 때 notifyAll 을 호출할 것이다.

```
public synchronized void transfer(int from , int to, int amount)
{
        accounts[from] -= amount;
        accounts[to] += amount;
        ntransacts++;
        notifyAll();
}
```

이것은 기다리고 있는 쓰레드들에게 다시 실행할 수 있는 기회를 제공한다. 보다 높은 잔액을 위해 기다리고 있는 쓰레드에게 다시 잔액을 검사할 수 있는 기회를 준다. 만일 잔액이 충분하면 전송을 수행한다. 만일 그렇지 않다면 다시 wait 을 호출한다.

notifyAll 을 호출하는 것이 대기중인 쓰레드를 즉시 구동시키는 것은 아니다. 오직 현재의 쓰레드가 동기화된 메소드를 종료한 다음, 객체내에 항목들에 대해 경쟁할 수 있도록 대기중인 쓰레드들을 언블록한다.

팁 : 만일 여러분의 멀티쓰레드 프로그램이 잘 작동하지 않으면 모든 wait 메소드가 notifyAll 와 일치하는 지 이중 검사를 하라.

만일 여러분이 transfer 메소드의 동기화된 버전을 사용하는 예제 프로그램을 수행하면 아무것도 잘못되는 것이 없다는 것을 알게 될 것이다. 총 합계는 영원히 $100,000 으로 남는다. (반복하자면 여러분은 프로그램을 종료하기 위해 CTRL+C 를 눌러야 한다.)

또한 여러분은 예제 1-6 에서 해당 프로그램이 약간 느리게 수행된다는 것을 눈치채게 될 것이다.– 이것은 동기화 메카니즘내에 계정 작업을 추가한 대가이다.

예제 1-6 : SynchBankTest.java

```
public class SynchBankTest
{  public static void main(String[] args)
   {  Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
      int i;
      for (i = 0; i < NACCOUNTS; i++)
      {  TransferThread t = new TransferThread(b, i,
            INITIAL_BALANCE);
         t.setPriority(Thread.NORM_PRIORITY + i % 2);
         t.start();
      }
   }

   public static final int NACCOUNTS = 10;
   public static final int INITIAL_BALANCE = 10000;
}

class Bank
{  public Bank(int n, int initialBalance)
   {  accounts = new int[n];
      int i;
      for (i = 0; i < accounts.length; i++)
         accounts[i] = initialBalance;
      ntransacts = 0;
   }

   public synchronized void transfer(int from, int to, int amount)
   {  try
      {  while (accounts[from] < amount)
            wait();
         accounts[from] -= amount;
         accounts[to] += amount;
         ntransacts++;
```

```
            notifyAll();
          if (ntransacts % NTEST == 0) test();
         }
        catch(InterruptedException e) {}
    }

   public synchronized void test()
   {   int sum = 0;

      for (int i = 0; i < accounts.length; i++)
           sum += accounts[i];

        System.out.println("Transactions:" + ntransacts
          + " Sum: " + sum);
    }

   public int size()
    {   return accounts.length;
    }

    public  static final int NTEST = 10000;
   private final int[] accounts;
   private long ntransacts = 0;
}

class TransferThread extends Thread
{  public TransferThread(Bank b, int from, int max)
    {   bank = b;
        fromAccount = from;
        maxAmount = max;
    }

   public void run()
    {   try
        {   while (!interrupted())
          {   int toAccount = (int)(bank.size() * Math.random());
```

```
            int amount = (int)(maxAmount * Math.random());
              bank.transfer(fromAccount, toAccount, amount);
               sleep(1);
            }
         }
        catch(InterruptedException e) {}
    }

   private Bank bank;
   private int fromAccount;
   private int maxAmount;
}
```

다음은 동기화 메카니즘이 작동하는 방법에 대한 정리이다.

1. 동기화 메소드를 호출할려면 암묵적인 전달인자는 락이 걸려야 한다. 메소드를 호출하는 것은 객체에 락을 건다. 호출에서 복귀하는 것은 암묵적인 전달인자 객체를 언락한다. 따라서 오직 한번에 한 개의 쓰레드만이 특정 객체상의 동기화 메소드들을 실행할 수 있다.
2. 쓰레드가 wait 호출을 실행할 때, 주변의 객체는 락이 걸리고 대기 목록에 포함된다.
3. 대기 목록에서 쓰레드를 제거하기 위해 몇몇 다른 쓰레드는 같은 객체에 대해 notify 나 notifyAll 을 호출해야 한다.

스케쥴링 규칙들은 복잡하다. 그러나 이러한 규칙들을 실제 적용하는 것은 상당히 간단하다. 다음의 네가지 규칙들을 따른다.

1. 만일 둘 또는 그 이상의 쓰레드들이 객체를 변경하면 메소드에 synchronized 로서 변경을 수행하도록 선언한다. 객체 변경시 영향을 받는 읽기만 하는 메소드들 또한 동기화되어야 한다.
2. 만일 한 쓰레드가 객체의 상태를 변경하기 위해 기다려야 한다면 동기화된 메소드에 진입하고 wait 을 호출하는 것에 의해서 외부에서가 아니라 객체의 내부에서 기다려야 한다.
3. 메소드가 객체의 상태들을 변경할 때마다 notifyAll 을 호출해야 한다. 이것은 기다리고 있는 쓰레드에게 주변 환경이 변했는지 알 수 있는 기회를 제공한다.
4. wait 와 notify/notifyAll 은 Thread 클래스가 아니라 Object 클래스의 멤버라는 것을 기억하라. 여러분의 wait 호출이 같은 객체상의 통보(notification) 의해서 일치하는 지 이중 검사하라.

노트 : 때때로 새로운 동기화 메소드를 작성하지 않고 단지 몇 개의 지시사항을 위해 객체에 락을 걸고 배타적 접근을 얻는 경우가 유용할 수 있다. 여러분은 이러한 접근을 위해 동기화 블록을 사용한다. 동기화 블록은 synchronized(obj)로 시작하고 { …}로 둘러쌓여있는 일련의 구문들로 구성된다. 여기서 obj 는 락을 걸고자 하는 객체이다. 다음은 구문 예이다:

```
public void run()
{       …
        synchronized(bank)  // lock the bank object
        {       if(bank.getBalance(from) >= amount)
                        bank.transfer(from, to, amount);
        }
        …
}
```

위의 예제 코드에서 동기화 블록은 다른 쓰레드가 bank 객체상의 동기화 메소드를 호출하기 전에 완벽하게 수행할 것이다.

대부분의 사람들은 동기화 블록이 혼란스럽다고 생각하고 가급적 사용을 하지 않으려고 노력한다. 이러한 동기화 블록의 유일한 장점은 메소드 호출을 줄이는 것이다. 따라서 약간의 효율성을 제공한다.

---

java.lang.Object

- void notifyAll

현재 객체상의 wait 을 호출하는 쓰레드들을 언블록한다. 이 메소드는 오직 동기화 메소드나 블록 내부에서부터 호출할 수 있다. 이 메소드는 만일 현재 쓰레드가 객체의 락에 대한 소유자가 아니라면 IllegalMonitorStateException 을 발생한다.

- void notify

현재 객체상에서 wait 을 호출하는 쓰레드들 사이에서 임의로 선택한 쓰레드를 언블록한다. 이 메소드는 오직 동기화된 메소드나 블록 내부에서 호출될 수 있다. 이 메소드는 현재 쓰레드가 해당 객체 모니터의 소유주가 아니면 IllegalMonitorStateException 을 투구한다.

- void wait()

쓰레드가 알려질까지(notified) 기다리게 만든다. 이 메소드는 오직 동기화된 메소드 내부에서 호출될 수 있다. 이 메소드는 현재 쓰레드가 해당 객체 모니터의 소유주가 아니면 IllegalMonitorStateException 을 투구한다.

데드락

자바 언어에서 동기화 기능은 편리하고 강력하다. 그러나 멀티쓰레딩에서 발생할 수 있는 모든 문제를 풀 수는 없다. 다음의 상황을 생각해보자:

계정 1 : $2000
계정 2: $3000
쓰레드 1: 계정 1 에서 계정 2 로 $3,000 을 전송한다.
쓰레드 2: 계정 2 에서 계정 1 로 $4,000 을 전송한다.

그림 1-9 에서 보여지듯이 , 쓰레드 1 과 2 는 분명하게 블록되어 있다. 계정 1 과 2 에 충분한 돈이 없기 때문에 양쪽 모두 처리될 수 없다.

그림 1-9 : 데드락 상황

각각의 쓰레드들이 보다 많은 돈을 기다리고 있기 때문에 10 개의 모든 쓰레드들이 블록되는 것이 가능할까? 이와 같은 상황을 *"데드락(deadlock)"* 이라 부른다.

우리의 프로그램에서 데드락은 간단한 이유에서 발생하지 않았다. 각각의 전송된 양은 고작 $10,000 이었다. 거기에는 10 개의 계정들과 그것들의 총합계는 $100,000 이기 때문에 적어도 계정들 중 하나는 항상 $10,000 이상을 가져야 만 했다. 그러므로 $10,00 이상을 갖는 계정으로 부터 돈을 이체하는 쓰레드는 처리될 수 있었다.

그러나 여러분이 쓰레드의 run 메소드를 $10,000 트렌젝션 제한을 없애버리면 데드락들은 자주 발생할 것이다. 이를 시도해 보라. 각 maxAmount 가 14,000 을 갖는 TransferThread 를 구성하고 프로그램을 수행하라. 이 프로그램을 잠시 동안 수행하다 중단될 것이다.

데드락을 발생시키는 다른 방법은 i 번째 계정으로 부터 돈을 가져오는 것보다 i 번째 계정에 돈을 입금할 책임이 있는 i 번째 쓰레드를 만드는 것이다. 이 경우 모든 쓰레드들이 한 계정을 잠식할 수 있는 작은 가능성이 있다. 각각의 잠식 노력은 한 계정이 가지고 있는 것보다 더 많은 금액을 잠식한다. 이를 시도해 보라. SynchBankTest 프로그램에서 TransferThread 클래스의 run 메소드를 변경하라. transfer 호출에서 fromAccount 와

toAccount 를 뒤집어 보라. 프로그램을 수행하면 거의 바로 데드락이 발생하는 과정을 보게 될 것이다.

다음은 데드락을 쉽게 발생시키는 또 다른 상황이다: SynchBankTest 프로그램에서 notifyAll 메소드를 notify 로 변경한다. 여러분은 프로그램이 재빨리 중단되는 것을 발견하게 될 것이다. 자금을 추가하기 위해 대기하고 있는 모든 쓰레드들에게 알리는 notifyAll 과 달리 notify 메소드는 오직 하나의 쓰레드만을 언블록한다. 만일 해당 쓰레드가 처리될 수 없다면 모든 쓰레드들은 블록되어 질 것이다. 데드락을 발생하는 다음의 예제 시나리오를 고려해 보자.

계정 1 : $19,000
모든 다른 계정 : 각각 $9,000
쓰레드 1 : 계정 1 에서 계정 2 로 $9,500 전송
모든 다른 쓰레드 : 해당 계정에서 다른 계정로 $9,100 전송

분명, 쓰레드 1 을 제외한 모든 쓰레드들은 그들 계정에 충분한 돈이 없기 때문에 블록된다.

쓰레드 1 이 처리되었다. 다음으로, 우리는 다음 상황을 맞는다.

계정 1 : $9,500
계정 2 : $19,500
모든 다른 계정 : 각 $9,000

그런 다음, 쓰레드 1 은 notify 를 호출한다. notify 는 임의대로 언블록할 쓰레드를 선정한다. 쓰레드 3 을 선택했다고 가정해보자. 쓰레드 3 이 깨어난 후, 해당 계정에 충분한 돈이 없다는 것을 발견한다. 그리고는 다시 wait 을 호출한다. 그러나 쓰레드 1 은 여전히 수행중이다. 새로운 임의 트렌젝션이 생성된다. 즉,

쓰레드 1 : 계정 1 에서 계정 2 로 $9,600 을 전송한다.

이제, 쓰레드 1 은 다시 wait 을 호출한다. 그리고 모든 쓰레드들이 블록된다. 시스템은 데드락에 빠진다.

여기서의 문제는 notify 호출이다. 오직 하나의 쓰레드가 언블록된다. 그리고 이 쓰레드가 처리를 위해 필요한 쓰레드가 아닐 수 있다.(우리의 시나리오에서 쓰레드 2 는 계정 2 로 ㅂ터 돈을 가져와야 한다.) 반대로 notifyAll 은 모든 쓰레드들을 언블록한다.

불운하게도 자바에는 이러한 데드락들을 피하거나 중단시키는 방법이 없다. 여러분은 여러분의 쓰레드들이 데드락 상황이 발생하지 않도록 디자인해야 한다. 여러분의 프로그램을 분석하고 객체를 기다리고 있는 모든 쓰레드가 결국 활동할 수 있게 끔 확실하게 해야 한다.

쓰레드 동기화와 데드락 방지는 어려운 주제들이다. 그리고 우리는 관심있는 독자들에게 스티브 클라이만 등이 집필한 "Programming with Threads(Sunsoft Press/Printice-Hall,1996)" 라는 책을 권한다.

Stop과 suspend 메소드가 디프리케이드된 이유는?

자바 1.0 플랫폼은 stop 메소드를 단순히 쓰레드를 종료하고 suspend 메소드를 다른 쓰레드가 resume 을 호출할 때까지 한 쓰레드를 블록하도록 정의하였다. 이들 메소드는 모두 자바 2 플랫폼에서 재정의되었다. stop 메소드는 본래 불안정하다. 그리고 suspend 메소드는 자주 데드락을 발생하게 하는 원인이 되는 걸 경험했다. 이 절에서 여러분은 왜 이들 메소드들이 문제를 발생하고 여러분이 어떻게 해야 이들 문제를 어떻게 피하는 지 보여준다.

먼저 stop 메소드를 살펴보자. 쓰레드가 중단했을 때 즉시 모든 객체들에 대한 락을 포기한다. 이것은 객체들을 불일치 상태로 남겨둘 수 있다. 가령, TransferThread 가 한 계정에서 다른 계정으로 돈을 이동하는 중간, 철회한 다음 예금하기 전에 멈추었다고 가정해보자. 이제 은행 객체는 손상되었다.

주의: 기술적으로 말해서, stop 메소드는 ThreadDeath 타입의 예외 객체를 투구하기 위해 쓰레드를 중단하는 원인이 된다. 이 예외상황은 run 메소드를 포함한 모든 보류된 메소드들을 종료한다. 동기화 메소드가 이 방법으로 종료될 때마다 객체는 언락된다.

같은 이유에서, 동기화 메소드에서 감지되지 않는 모든 예외상황은 완전히 처리되지 않은 메소드를 종료하고 손상된 객체들을 만들어내는 원인이 된다.

한 쓰레드가 다른 스레드를 중단하길 원할 때, stop 메소드가 안전할 때와 손상된 객체를 만들 때를 알 방법이 없다. 그러므로 이 메소드가 디프리케이트되었다.

만일 여러분이 한 쓰레드를 안전하게 중단할 필요가 있다면 중단 요청이 왔는지를 알려주는 변수를 주기적으로 검사하는 쓰레드를 갖을 수 있다.

```
public class MyThread extends Thread
{       public void run()
        {       while (!stopRequested && more work to do)
                {       do more work
                }
        public void requestStop()
        {       stopRequested = true;
        }

        private Boolean stopRequested;
}
```

이 코드는 run 메소드가 종료할 때를 조절하게 해준다. 그리고 손상된 상태로 남아있는 객체가 없다는 것을 보증하는 것은 전적으로 run 메소드에 달려 있다.

쓰레드의 메인 루프에서 stopRequested 변수를 검사하는 것은 해당 쓰레드가 현재 블록되는 상황을 제외하고 것은 잘 작동한다. 이 경우, 쓰레드는 오직 언블록된 다음 종료할 것이다. 여러분은 인터럽트하여 해당 쓰레드를 블록된 상태에서 강제로 벗어나게 한다. 따라서 여러분은 interrupt 를 호출하는 requestStop 메소드를 정의해야 한다.

```
        public void requestStop()
        {       stopRequested = true;
                interrupt();
        }
```

여러분은 InterruptedException 을 위해 catch 절에서 stopRequested 변수를 검사할 수 있다. 예를 들면,

```
try
{       wait();
}
catch(InterruptedException e)
{       if(stopRequested)
        return; // exit the run method
}
```

실제, 많은 프로그래머들은 쓰레드를 중단하기 위해 인터럽트를 사용하는 경향을 갖는다. 그러면 여러분은 stopRequested 변수를 검사할 필요가 없다 - 인터럽트할 때마다 단순히 run 메소드를 종료하면 된다.

다음으로, suspend 메소드에 무슨 잘못이 있는 지 살펴보자. stop 과 달리, suspend 는 객체에 손상을 주지 않는다. 그러나 만일 객체에 대한 락을 소유하고 있는 한 쓰레드가 일시 중단하면 객체는 쓰레드가 재작동할 때까지 이용할 수 없다. 만일, 특정 쓰레드가 resume 을 호출하기 전에 같은 객체에 대한 락을 얻기 위해 suspend 메소드를 호출하면 프로그램은 데드락에 빠진다: 일시 중단된 쓰레드는 재작동 되길 기다린다. 그리고 일시 중단중인 쓰레드는 객체가 언블록되길 기다린다.

이러한 상황은 그래픽 사용자 인터페이스에서 자주 발생한다. 우리의 은행 예제의 그래픽 시뮬레이션을 갖는다고 가정해 보자. 우리는 전송 쓰레드를 일시 중단하는 "pause" 라벨의 버튼과 이들 쓰레드를 재가동하는 "Resume" 라벨의 버튼을 갖고 있다.

```
public void actionPerformed(ActionEvent event)
{       Object source = event.getSource();
        if(source == pauseButton)
                for( int i = 0; i < threads.length; i++)
                        threads[i].suspend(); // Don't do this
        else if(source == resumeButton)
                for( int i = 0; i < threads.length; i++)
                        threads[i].resume(); // Don't do this
}
```

paintComponent 메소드는 bank.getAccount 메소드를 호출하여 각 계정의 챠트를 그린다. 그리고 bank.getAccount 메소드는 동기화되어 있다.

다음 절에서 보게 될 것처럼 , 버튼 액션들과 리페인팅은 모두 같은 쓰레드인 ***이벤트 디스패치 쓰레드***에서 발생한다.

이제 다음의 시나리오를 고려해 보자:

1. 전송 쓰레드들중 하나는 bank 객체상에 락을 획득한다.
2. 사용자는 "Pause" 버튼을 클릭한다.
3. 모든 전송 쓰레드들은 일시 중단된다. 이들 중 하나는 여전이 bank 객체상의 락을 유지한다.

4. 몇가지 이유에서, 계정 챠트는 다시 그려질 필요가 있다.
5. paintComponent 메소드는 동기화 메소드 bank.getBalance 를 호출한다.

이제 프로그램은 중단된다.

이벤트 디스패치 쓰레드는 bank 객체가 일시 중단된 쓰레드들중의 하나에 의해 락되기 때문에 작동할 수 없다. 따라서 사용자는 "Resume" 버튼을 클릭할 수 없으며 쓰레드들은 더 이상 다시 재개되지 못한다.

만일 여러분이 안전하게 쓰레드를 일시중단하길 원한다면 suspendRequested 변수를 제공하고 run 메소드의 메일 루프에서 이를 테스트해야 한다. 여러분은 변수가 설정됐음을 알았을 때 다시 이용 가능해 질 때까지 기다리고 있어야 한다. 그러나 여러분은 기다리도록 구현하길 원하지 않는다:

```
while(suspendRequested)
        sleep(1); // No
```

대신에, wait 을 호출하고 notify 를 호출하도록 requestResume 메소드를 정의한다. 그런 다음 쓰레드는 재작동될 때까지 블록되어 대기한다.

```
class MyThread extends Thread
{       public void requestSuspend()
        {       suspendRequested = true;
        }
        private synchronized void checkSuspended()
                throws InterruptedException
        {       while(suspendRequested)
                wait();
        }
        public synchronized void requestResume()
        {       suspendRequested = false;
                notify();
        }
        public void run()
        {       while(more work to do)
                {       checkSuspended();
                        do more work;
```

```
                }
        }
        private boolean suspendRequested;
}
```

무슨 이유에서 이 접근방법이 작동되지 않는 사용자 인터페이스의 문제를 해결하는 것일까? run 메소드는 임의의 다른 객체들에 락이 걸려있지 않는 공전 시점(idle moment)에 checkSuspended 를 호출한다. 그러므로 쓰레드는 단지 안전하게 작동될 때 스스로 블록된다.

노트 : 만일 여러분이 앞선 코드를 주의깊게 살펴보았다면 requestResume 과 checkSuspended 메소드들이 동기화되어 있다는 것을 알았을 것이다. 이들은 wait 와 notify 가 오직 동기화 메소드나 블록들내에서 호출될 수 있기 때문에 동기화될 필요가 있다. 그러나 어느 객체가 락이 걸린 것일까?

락이 걸린 객체는 쓰레드 그 자체이다. 이것은 우연한 락이다- 우리는 resume 과 checkSuspended 에 대한 연속적인 호출에 대해서는 관심을 갖지 않는다. 락 객체의 유일한 목적은 wait 와 notify 에 대한 호출을 조화시키는 것이다. notify 호출은 일시 중단된 쓰레드인 MyThread 객체상에 대기하고 있는 쓰레드를 언블록한다. 몇몇 프로그래머들은 이 같은 상호매칭 객체를 위해 별도의 더미 변수를 사용하길 선호한다:

```
private void checkSuspended()
        throws InterruptedException
{       synchronized (dummy)
        // synchronized necessary for calling wait
        {       while (suspendedRequested)
                dummy.wait(); // block this thread
        }
}
public void requestResume()
{       suspendRequested = false;
        synchronized(dummy)
        {       dummy.notify(); // unblock the thread waiting on dummy
        }
}
```

```
private Integer dummy = new Integer(0);
        // any non-null object will work
```

마지막으로, 여러분은 notifyAll 대신에 notify 를 사용할 수 있다. 왜냐하면 이 특별한 객체상에서는 오직 단일 쓰레드만이 기다리고 있기 때문이다.

물론, Thread.suspend 메소드를 피하는 것이 자동적으로 데드락을 피하는 것은 아니다. 만일 한 쓰레드가 wait 을 호출하면 그 어느 것도 깨어나지 않는다. 그러나 여기에는 필수적인 차이점이 존재한다. 한 쓰레드는 wait 을 호출할 때 조정할 수 있다. 그러나 Thread.suspend 메소드는 쓰레드의 동의없이 언제라도 쓰레드상의 외부에서 호출될 수 있다. Thread.stop 메소드에 대해서도 동일하다. 이러한 이유에서 이들 두 메소드는 디프리케이트되었다.

노트 : 이 절에서, 우리는 requestStop, requestSuspend 그리고 requestResume 메소드를 정의했다. 이들 메소드는 디프리케이트된 stop,suspend 그리고 resume 메소드와 유사한 기능을 제공하는 반면에 이들 디프리케이트된 메소드들의 위험을 피하게 해준다. 여러분은 많은 프로그래머들이 유사한 메소드를 구현하는 것을 알게 될 것이다. 그러나 이들에게 다른 이름들을 제공하는 것 대신에 간단히 stop,suspend 그리고 resume 메소드를 오버라이드 한다. 디프리케이트된 메소드를 디프리케이트되지 않는 다른 메소드로 오버라이드하는 것은 완전히 합법적이다. 만일 여러분이 프로그램내에서 stop,suspend 그리고 resume 에 대한 호출을 보게 된다면 프로그램이 잘못됐다고 가정할 필요는 없다. 먼저 프로그래머들이 안전한 버전으로 디프리케이트된 메소드들을 오버로드했는 지 검사하라.

애니메이션

앞 절에서 우리는 한 프로그램을 여러 개의 동시성 작업들로 나눌 때 필요한 것이 무엇인지 배웠다. 각각의 작업은 Thread 를 확장한 클래스의 run 메소드안에 위치돼야 한다. 그러나 만일 우리가 이미 다른 클래스를 확장한 클래스내에 run 메소드를 추가하길 원한다면 무슨 일이 필요할까? 이러한 경우는 우리가 애플릿에 멀티쓰레딩 능력을 추가할 때 가장 자주 발생한다. 애플릿 클래스는 이미 JApplet 클래스로 부터 상속을 받는다. 자바에서 우리는 두개의 상위 클래스들로 부터 상속을 받을 수 없다. 그래서 우리는 인터페이스를 사용할 필요가 있다. 필요한 인터페이스는 자바내에 만들어져 있다. 이

를 Runnable 라고 부른다. 우리는 이 중요한 인터페이스를 다음에 까지 사용한다.

Runnable 인터페이스

여러분이 이미 Thread 이외에 다른 클래스로 부터 이미 상속받은 클래스에 멀티쓰레딩을 사용하고자 할 때마다 Runnable 인터페이스를 구현한 클래스를 만든다. 비록 여러분이 Thread 로 부터 상속받았지만 run 메소드내에서 수행할 필요가 있는 코드를 삽입해야 한다. 예를 들면,

```
class Animation extends JApplet implements Runnable
{       ...
        public void run()
        {       // 여기에 쓰레드의 액션이 존재한다.
        }
}
```

여러분은 여전히 쓰레드를 구동하기 위해 쓰레드 객체를 만들 필요가 있다. 해당 객체의 생성자에 Runnable 객체에 대한 참조하는 쓰레드를 제공하라. 그리고나서 해당 객체의 run 메소드들 호출한다.

이 호출은 다음의 예제에서 처럼 애플릿의 start 메소드내에서 가장 일반적으로 수행된다:

```
class Animation extends JApplet implements Runnable
{       ...
        public void start()
        {       if(runner == null)
                {       runner = new thread(this);
                        runner.start();
                }
        }
        ....
        private Thread runner;
}
```

이 경우에 Thread 생성자에 대한 this 전달인자는 쓰레드가 실행할 때 호출돼야하는 run 메소드를 갖고 있는 객체가 Animation 객체의 인스턴스라는 것을 명시해준다.

만일 우리가 Thread 로 부터 다른 클래스를 정의하고 애플릿내에서 이것을 구동시킨다면 이 방버이 보다 쉬운 방법이 아닐까?

```
class AnimationThread extends Thread
{       public void run()
        {       // 여기에 쓰레드의 액션이 존재한다.
        }
}

class Animation extends JApplet
{       ...
        public void start()
        {       if(runner == null)
                {       runner = new AnimationThread();
                        runnner.start();
                }
        }
        ...
        private Thread runner;
}
```

이것은 정말로 깔끔하고 간단한 것 같다. 그러나 만일 run 메소드가 애플릿의 private 데이터에 접근해야 한다면 run 메소드에서 애플릿를 관리하기 위해 많은 노력을 기울이게 만든다. 그리고 이 방법 대신에 Runnable 인터페이스를 사용한다.

java.lang.thread

- Thread(Runnable target)

명시된 target 의 run() 메소드를 호출하는 새로운 쓰레드를 생성한다.

java.lang.runnable

- void run()

여러분은 이 메소드를 오버라이드해야하고 쓰레드내에서 실행하고자 하는 코드를 추가해야 한다.

프레임들의 로딩과 디스플레이

이 절에서 우리는 자바 애플릿내에서 가장 일반적으로 사용하는 쓰레드중에서 하나를 분석할 것이다: 애니메이션. 애니메이션은 이미지들의 연속적인 출력이고 동작하는 것 같은 환영을 제공한다. 연속적인 출력중 각각의 이미지들을 *프레임*이라 부른다. 물론 프레임들은 미리 준비돼야 한다.–오늘 날 개인용 컴퓨터들은 실시간 애니메이션을 위해 충분히 빠르게 그림그리는 과정을 계산해낼 만한 힘을 갖고 있지 못하다.

여러분은 각각의 프레임을 분리된 파일내에 위치시키거나 한 파일내에 모두 위치시킬 수 있다. 우리는 후자를 사용한다. 이것은 보다 이미지를 로딩하는 과정을 쉽게 만든다. 우리의 예제에서 우리는 지구가 회전되는 36 개의 이미지들을 담은 한 파일을 사용한다. 다음 그림 1-10 은 첫번째 몇몇 프레임들이다.

그림 1-10 : 이 파일은 36 개의 지구 이미지를 갖고 있다.

애니메이션 애플릿은 먼저 모든 프레임을 얻어야 한다. 그런 다음 교대로 고정된 시간 동안 각각의 프레임을 보여준다. 여러분은 이미지를 로드하기 위해 MediaTracker 객체를 사용한다. 장면들 뒤는 프로그래머에게 투명하다. addImage 메소드는 이미지를 얻기위해 새로운 쓰레드를 생성한다. 이미지를 로딩하는 것은 아주 느릴 수 있다. 특히, 만일 이미지가 많은 프레임을 갖거나 네트웍상으로 떨어진 곳에 위치하고 있을 경우 느리다. WaitForID 호출은 이미지가 완전히 로드될 때가지 블록한다.

일단 이미지가 로드되면 우리는 한 번에 하나의 프레임을 연출한다. i 번째 프레임을 그리기 위해 우리는 다음과 같이 메소드를 호출한다:

```
g.drawImage(image, 0, - i * imageHeight , imageCount , null);
```

그림 1-11 에서, y 축의 *음수* 오프셋을 주목하라.
이것은 본래의 캔버스위에 첫번째 프레임이 잘 표현되게 한다. i 번째 프레임의 가장 위쪽 부분은 캔버스의 가장 위쪽 부분이 된다.

그림 1-11 : 연속된 프레임에서 한 프레임 뽑기

약간의 지연시간이 흐른 다음, 우리는 i 를 증가시키고 다음 프레임을 그린다.

애니메이션을 조정하기 위한 쓰레드 사용

우리의 애플릿은 단 하나의 쓰레드를 갖을 것이다.

```
class Animation implements Runnable
{       ...
        Thread runner = null;
}
```

여러분은 많은 자바 애플릿들에서 이같은 쓰레드를 볼 것이다. 종종 이것은 kicker 라 불린다. 그리고 우리는 변수 이름으로서 killer 를 보았었다. 우리는 runner 가 보다 합리적인 이름이라고 생각한다.

가장 먼저, 우리는 이 쓰레드를 다음과 같이 사용할 것이다.:

- 사용자가 애플릿을 지켜보고 있을 때 애니메이션을 시작한다. 그리고
- 사용자가 부라우져내에서 다른 페이지로 전환할 때 멈춘다.

우리는 애플릿의 start 메소드내의 쓰레드를 만드는 것과 stop 메소드내에서 이를 파괴하는 것에 의해서 이 작업을 할 수 있다. 여러분은 많은 애플릿들에서 발견할 다음의 코드로 이 작업을 수행할 수 있다.

```
class Animation extends JApplet implements Runnable
{       public void start()
        {       if(runner == null)
                {       runner = new thread(this);
                        runner.start();
                }
        }
        public void stop()
        {       runner.interrupt()
                runner = null;
        }
        ...
}
```

다음 메소드 호출에 주목하라.

```
runner = new Thread(this)
```

이것은 위의 애플릿의 run 메소드를 호출하는 쓰레드를 생성한다.
다음은 run 메소드이다. 이 메소드는 루프를 돌며 화면에 이미지를 출력하고 작업이 끝나면 수면상태로 접어든다.

```
public void run()
{       try
        {       while(!Thread.interrupted())
                {       repaint();
                        current = (current + 1) % imageCount;
                        Thread.sleep(200);
                }
        }
        catch(InterruptedException e) {}
}
```

마지막으로, 우리는 애니메이션을 중단하고 재시작하기 위해 다른 메소드를 구현하였다. 여러분이 애플릿 윈도우상에서 마우스를 선택했을 때 애니메이션은 멈춘다. 여러분이 다시 클릭했을 때 다시 시작한다. 우리는 현재 쓰레드가 작동하고 있는지 아닌지에 대한 여부를 알려주는 쓰레드 변수로 runner 를 사용한다. 쓰레드가 종료될 때마다 변수는 null 로 설정된다. 이것은 많은 멀티쓰레드 기반의 애플릿들에서 발견할 수 있는 일반적인 방식이다.

```
public void init()
{   addMouseListener(new MouseAdapter()
                {   public void mousePressed(MouseEvent evt)
        {   if (runner == null)
                        start();
                else
                        stop();
        }
        });
        ....
}
```

애플릿은 HTML 파일내의 PARAM 부분으로 부터 이미지의 이름과 프레임들의 숫자를 읽는다.(CD-ROM 상에 포함되어 있음)

```
<applet code=Animation.class width=100 height=100>
<param name=imagename value="globe.gif">
<param name=imagecount value="36">
</applet>
```

예제 1-7 은 애플릿의 코드이다. Start,stop 메소드는 애플릿을 시작하고 중단한다 – 이것은 쓰레드가 생성해내는 메소드가 아니다.

이 애니메이션 애플릿은 장면들 뒤에서 무슨 일이 발생하는 지를 여러분에게 보여주기 위해 단순화되었다. 만일 여러분이 오직 여러분의 웹 페이지에 움직이는 이미지를 넣는 것에 관심이 있다면 animator 애플릿 대신에 JDK 의 데모 절을 참고하라. 여기에 있는 애플릿은 우리의 것보다 많은 옵션을 가지고 있으면 사운드를 추가할 수 있게 해준다.

예제 1-7 : Animation.java

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

public class Animation extends JApplet
    implements  Runnable
{    public void init()
    {    addMouseListener(new  MouseAdapter()
            {    public  void  mousePressed(MouseEvent  evt)
                {    if  (runner  ==  null)
                            start();
                        else
                            stop();
                }
            });

          try
```

```
    {   imageName = getParameter("imagename");
        if (imageName == null) imageName = "";

         imageCount = 1;
         String param = getParameter("imagecount");
         if (param != null)
              imageCount = Integer.parseInt(param);
      }
    catch (Exception e)
    {   showStatus("Error: " + e);
      }

    current = 0;
    image = null;
     loadImage();
}

public void loadImage()
{   try
   {   URL url = new URL(getDocumentBase(), imageName);
         image = getImage(url);
        MediaTracker tracker = new MediaTracker(this);
          tracker.addImage(image, 0);
           tracker.waitForID(0);
         imageWidth = image.getWidth(null);
         imageHeight = image.getHeight(null);
          resize(imageWidth, imageHeight / imageCount);
      }
    catch (InterruptedException e)
        // thrown by MediaTracker.waitFor
     {   showStatus("Loading interrupted");
      }
     catch(MalformedURLException e)
     {   showStatus("Bad URL");
      }
}
```

```
 public void paintComponent(Graphics g)
{   if (image == null) return;
     g.drawImage(image, 0, - (imageHeight / imageCount)
          * current, null);
 }

public void start()
{   runner = new Thread(this);
      runner.start();
      showStatus("Click to stop");
 }

public void stop()
 {    runner.interrupt();
     runner = null;
     showStatus("Click to restart");
 }

public void run()
 {    try
     {    while (!Thread.interrupted())
          {    repaint();
              current = (current + 1) % imageCount;
               Thread.sleep(200);
          }
      }
     catch(InterruptedException e) {}
 }

private Image image;
private int current;
private int imageCount;
private int imageWidth;
private int imageHeight;
 private String imageName;
```

```
    private Thread runner;
}
```

타이머

많은 프로그래밍 환경들에서 여러분은 타이머들을 설정할 수 있다. 타이머는 여러분의 프로그램 구성요소에게 정해진 시간간격을 알린다. 예를 들면, 윈도우에서 시계를 출력하기 위해서 시계 객체는 매 초를 알아야 한다.

스윙은 쉽게 사용할 수 있는 내장된 타이머를 가지고 있다. 여러분은 ActionListener 인터페이스를 구현한 클래스의 객체와 타이머 경고사이의 밀리초 단위의 지연시간을 제공함으로써 타이머를 생성한다.

```
Timer t = new Timer(listener , 1000);
t.start();
```

그런 다음, 타이머 간격이 경과할 때마다 리스너 클래스의 actionPerformed 메소드가 호출된다. actionPerformed 메소드는 여러분들이 콜 백내에서 스윙 메소드들을 자유롭게 호출할 수 있도록 타이머 쓰레드가 아니라 이벤트 디스패티 쓰레드상에서 자동적으로 호출된다.

노트 : JDK1.3 은 지연 실행을 위한 TimerTask 를 스케쥴하는 java.util.Timer 클래스를 가지고 있다. TimerTask 클래스는 Runnable 인터페이스를 구현했으며 작업을 취소하기 위한 cancel 메소드를 제공한다. 그러나 java.util.Timer 클래스는 주기적인 콜백을 위한 제공하지 않는다.

때때로 스윙 타이머를 사용하지 않고 여러분 자신의 타이머를 사용하는 것은 유용하다. 가령, 스윙을 사용하지 않는 응용 프로그램내에서 처럼. 이 절에서 우리는 이것을 어떻게 구현하는지 보여준다.

우리의 타이머는 자신이 소유하고 있는 쓰레드내에서 작동된다. 그래서 Thread 를 확장해야 한다. run 메소드내에서 명시된 간격동안 수면을 한 다음에 목표에 이를 통지한다.

```
class Timer extends Thread
```

```
{           ....
            public void run()
            {   try
                {   while  (!interrupted())
                        {   sleep(interval);
                        target.timeElapsed(this);
                        }
                }
                catch(InterruptedException  e)
                {
                }
        }

            private int interval;
}
```

우리는 ActionListener 가 타이머 통지를 위해 아주 적합한 인터페이스다 라고 생각하지는 않는다. 그래서 우리는 TimerListener 라 부르는 새로운 리스너 인터페이스를 사용한다.

```
interface TimerListener
{           void timeElapsed(timer t);
}
```

따라서 타이머 틱(tick)을 받길 원하는 객체는 TimerListener 인터페이스를 구현해야 한다. 일정한 간격으로 반복되는 액션은 timeElapsed 메소드내에 포함돼야 한다.

그림 1-12 는 6 개의 다른 시계들을 보여준다.

그림 1-12 : 시계 쓰레드들

각각의 시계는 TimerListener 를 구현한 ClockCanvas 클래스의 인스턴스이다. Clock 클래스의 timeElapsed 메소드는 시계를 다시 그린다.

```
class ClockCanvas extends Canvas implements Timelistener
{           ....
```

```
        public void timeElapsed(Timer t)
        {   calendar.setTime(new Date());
        seconds = calendar.get(Calendar.HOUR) * 60 * 60
                + calendar.get(Calendar.MINUTE) * 60
                + calendar.get(Calendar.SECOND);
        repaint();
    }
```

timeElapsed 메소드가 실제 타이머 객체 t 로 부터 시간을 얻지 못한다는 것에 주목하라. 타이머 틱(tick)은 오직 1 초를 *추정해서* 진행한다. 정확한 시간을 출력하기 위해서는 여전히 우리는 시스템 시간을 필요로 한다.

TimeElapsed 메소드는 이벤트 디스패치 쓰레드가 아니라 타이머 쓰레드로부터 나온다. 따라서 스윙 메소드 호출에서 할 수 있는 것보다는 제약이 있다. 그러나 이 메소드는 오직 repaint 메소드를 호출한다. 이 메소드는 모든 쓰레드로부터 안전한다.

여러분은 예제 1-8 에서 완전한 코드를 발견할 수 있다.

예제 1-8 : TimerTest.java

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

public class TimerTest
{  public static void main(String[] args)
   {  JFrame f = new TimerTestFrame();
      f.show();
   }
}

class TimerTestFrame extends JFrame
{  public TimerTestFrame()
   {  setSize(450, 300);
```

```
        setTitle("TimerTest");

        addWindowListener(new WindowAdapter()
            {   public void windowClosing(WindowEvent e)
                {   System.exit(0);
                }
            } );

        Container c = getContentPane();
        c.setLayout(new GridLayout(2, 3));
        c.add(new ClockCanvas("San Jose", "GMT-8"));
        c.add(new ClockCanvas("Taipei", "GMT+8"));
        c.add(new ClockCanvas("Berlin", "GMT+1"));
        c.add(new ClockCanvas("New York", "GMT-5"));
        c.add(new ClockCanvas("Cairo", "GMT+2"));
        c.add(new ClockCanvas("Bombay", "GMT+5"));
    }
}

interface TimerListener
{   void timeElapsed(Timer t);
}

class Timer extends Thread
{   public Timer(int i, TimerListener t)
    {   target = t;
        interval = i;
        setDaemon(true);
    }

    public void run()
    {   try
        {   while (!interrupted())
            {   sleep(interval);
                target.timeElapsed(this);
            }
```

```
            }
            catch(InterruptedException e) {}
        }

    private TimerListener target;
    private int interval;
}

class ClockCanvas extends JPanel
    implements TimerListener
{   public ClockCanvas(String c, String tz)
    {   city = c;
        calendar = new GregorianCalendar(TimeZone.getTimeZone(tz));
        Timer t = new Timer(1000, this);
        t.start();
        setSize(125, 125);
    }

    public void paintComponent(Graphics g)
    {   super.paintComponent(g);
        g.drawOval(0, 0, 100, 100);
        double hourAngle = 2 * Math.PI
            * (seconds - 3 * 60 * 60) / (12 * 60 * 60);
        double minuteAngle = 2 * Math.PI
            * (seconds - 15 * 60) / (60 * 60);
        double secondAngle = 2 * Math.PI
            * (seconds - 15) / 60;
        g.drawLine(50, 50, 50 + (int)(30
            * Math.cos(hourAngle)),
            50 + (int)(30 * Math.sin(hourAngle)));
        g.drawLine(50, 50, 50 + (int)(40
            * Math.cos(minuteAngle)),
            50 + (int)(40 * Math.sin(minuteAngle)));
        g.drawLine(50, 50, 50 + (int)(45
            * Math.cos(secondAngle)),
            50 + (int)(45 * Math.sin(secondAngle)));
```

```
        g.drawString(city, 0, 115);
    }

  public void timeElapsed(Timer t)
   {   calendar.setTime(new Date());
      seconds = calendar.get(Calendar.HOUR) * 60 * 60
           + calendar.get(Calendar.MINUTE) * 60
            + calendar.get(Calendar.SECOND);
        repaint();
    }

  private int seconds = 0;
   private String city;
   private int offset;
   private GregorianCalendar calendar;

  private final int LOCAL = 16;
}
```

데몬 쓰레드

만일 여러분이 타이머 클래스의 생성자를 주의깊게 살펴봤으면 여러분은 다음과 같은 메소드 호출을 눈치챘을 것이다.:

```
setDaemon(true);
```

이것은 타이머 쓰레드를 *데몬 쓰레드*로 만든다. 데몬 쓰레드에는 귀신이나 흉물스러운 그 어떤 것도 없다. 데몬은 다른 것들에게 서비스를 제공하는 것외에는 그 어떤 다른 역할도 없는 간단한 쓰레드이다. 만일 남아있는 쓰레드들이 모두 데몬이라면 현 시점에서 프로그램의 수행을 유지하는 것은 없다.

그래픽 응용 프로그램에서 타이머 클래스 쓰레드들은 프로그램의 종료에 영향을 미치지 못한다. 이것은 사용자가 프로그램을 종료할 때까지 살아있다.(이벤트 디스패치 쓰레드는 데몬 쓰레드가 아니다.) 그러나 여러분이 텍스트 응용 프로그램에서 타이머 클래스를 사용할 때 여러분은 타이머 쓰레드들을 종료하기 위해 걱정할 필요가 없다. 타이머 쓰

레드가 아닌 쓰레드들이 그들의 run 메소드를 수행을 종료할 때 응용 프로그램은 자동적으로 종료된다.

javax.swing.Timer

- Timer(int delay, ActionListener listener)

리스너에게 이벤트들을 보내는 새로운 타이머를 생성한다.

전달인자 : delay 이벤트 통지 사이의 지연시간, 밀리초

listener 지연 시간이 경과됐을 때 통보해야 될 액션 리스너

- void start()

타이머를 시작한다. 이 호출 다음에, 타이머는 액션 리스너에게 이벤트 전달을 시작한다.

- void stop()

타이머를 중지한다. 이 호출 다음에, 타이머는 액션 리스너에게 이벤트 전달을 중지한다.

java.lang.Thread

- void setDaemon(boolean on)

해당 Thread 를 데몬 쓰레드나 사용자 쓰레드로 표시한다. 시스템내에서 오직 데몬 쓰레드만이 남아 수행하고 있을 때 프로그램은 종료한다. 이 메소드는 쓰레드가 시작하기 전에 호출돼야 한다.

쓰레드와 스윙

소개에서도 언급했듯이, 여러분의 프로그램에서 쓰레드를 사용하는 이유들중 하나는 여러분의 프로그램을 보다 민감하게 만드는 것이다. 여러분의 프로그램이 무엇인가 시간을 소비하는 작업을 필요로 할 때 사용자 인터페이스를 블록킹하는 대신에 다른 작업 쓰레드를 생성한다.

그러나, 여러분은 작업 쓰레드에서 무엇을 할 것인지 주의깊게 다루어야 한다. 왜냐하면 놀랍게도 스윙은 쓰레드에 안전하지 않기 때문이다. 즉, 스윙 클래스들의 대다수 메소드들이 동기화되지 않았기 때문이다. 만일 여러분이 다중 쓰레드로부터 사용자 인터페이스 요소들을 조작하려 한다면 여러분의 사용자 인터페이스는 붕괴될 것이다.

예를 들면, 이 절의 마지막 부분에 발견하게 될 코드를 갖는 테스트 프로그램을 수행하라. 여러분이 “Bad” 버튼을 클릭할 때 마다 리스트 박스를 편집하고 값을 추가,제거하는 새로운 쓰레드가 시작한다.

```
class BadWorkerThread extends Thread
{   public BadWorkerThread(DefaultListModel aModel)
    {   model = aModel;
        generator = new Random();
     }

    public void run()
    {   while (true)
        {   Integer i = new Integer(generator.nextInt(10));

              if (model.contains(i))
                   model.removeElement(i);
              else
                   model.addElement(i);

              yield();
         }
     }

    private DefaultListModel model;
    private Random generator;
}

class GoodWorkerThread extends Thread
{   public GoodWorkerThread(DefaultListModel aModel)
    {   model = aModel;
        generator = new Random();
     }

    public void run()
    {   while (true)
        {   final Integer i = new Integer(generator.nextInt(10));
              EventQueue.invokeLater(new Runnable()
                  {   public void run()
                       {   if (model.contains(i))
```

```
                        model.removeElement(i);
                    else
                        model.addElement(i);
                }
            });
        yield();
    }
  }

  private DefaultListModel model;
  private Random generator;
}
```

시도해 보라. “Bad”버튼을 두 번 클릭한다. 만일 여러분이 콘솔 윈도우에서 프로그램을 시작했다면 콘솔에 황당한 예외상황 메시지를 보게 될 것이다(그림 1-13 참조)

그림 1-13: 콘솔에서 예외상황 보고

무슨 일이 생긴 것일까? 항목이 리스트 모델에 삽입됐을 때, 모델은 디스플레이를 갱신하기 위한 이벤트를 발생한다. 그런 다음, 액션으로 들어가 모델의 현재 크기를 일고 모델 값을 출력하기 위한 준비를 한다. 그러나 작업 쓰레드는 계속 수행되고 있고 모델로부터 항목들을 제거한다. 그런 다음, 출력 코드는 실제보다 모델에 많은 값들이 있는 지를 생각하고 값들이 존재하지 않는 가를 묻고 ArrayIndexOutOfBounds 예외상황을 발생시킨다.

이 상황은 출력하는 동안 모델에 락을 거는 것에 의해 피할 수 있다. 그러나 스윙 디자이너들은 두가지 이유에서 스윙을 쓰레드에 안전에게 만드는 노력을 않기로 결정했다. 먼저, 동기화는 시간을 많이 소비하고 어느 누구도 스윙이 더 이상 느려지는 것을 원하지 않는 다는 것이다. 더욱 중요하게도, 스윙 팀이 어느 다른 팀이 쓰레드에 안전한 사용자 인터페이스 킷을 경험했는 지를 조사했다. 그들이 발견한 것은 이를 경험한 팀이 없다는 것이었다. 사용자 인터페이스 툴 킷을 만들 때, 여러분은 다른 프로그래머들이 그들 자신의 사용자 인터페이스 컴포넌트를 추가할 수 있도록 확장가능하길 원한다. 그러나 쓰레드에 안전한 툴킷을 사용하는 사용자 인터페이스 프로그래머들은 동기화에 대한 요구에 의해 혼돈스럽기 때문에 데드락 경향이 있는 컴포넌트를 생성하는 경향이 있다고 판명되었다.

그러므로, 여러분이 스윙에서 쓰레드를 사용할 때 다음의 몇가지 단순한 규칙을 따라야 한다. 그러나 먼저, 스윙 프로그램에서 무슨 쓰레드가 제공되는 지를 알아보자.

모든 자바 응용 프로그램은 *main 쓰레드*에서 수행되는 main 메소드를 시작한다. 스윙 프로그램에서 main 메소드는 기본적으로 다음을 수행한다:

- 프레임 윈도우에 컴포넌트들의 레이아웃을 위한 생성자를 호출한다.
- 윈도우상의 show나 setVisible 메소드를 호출한다.

첫번째 윈도우가 보여질 때, 두번째 쓰레드인 *이벤트 디스패키 쓰레드*가 생성된다. actionPerformed 나 paintComponent 같은 모든 이벤트 통지는 이벤트 디스패키 쓰레드에서 수행된다. 메인 쓰레드는 main 메소드가 종료할 때까지 계속 수행된다. 물론, 일반적으로 main 메소드는 프레임 윈도우를 출력한 다음에 즉시 종료된다(그림 1-14). 다른 쓰레드들은 이벤트 큐에 이벤트를 보내는 쓰레드처럼 장면뒤에서 수행되고 있다. 그러나 이들 쓰레드들은 응용 프로그래머들에게는 보이지않는다.

그림 1-14: 스윙 프로그램에서의 쓰레드

스윙 응용 프로그램에서, 필수적으로 이벤트 핸들러내에 포함되어 있는 모든 코드는 사용자 인터페이스와 다시 그리기 요청에 반응한다. 모든 코드는 이벤트 디스패칭 쓰레드상에서 작동된다. 다음은 여러분이 따라야 할 필요가 있는 규칙들이다.

1. 만일 액션이 시간을 많이 소비하면, 해당 작업을 수행하는 새로운 쓰레드를 생성한다. 만일 여러분이 이벤트 디스패키 쓰레드에서 오랜 시간이 걸리면 응용 프로그램은 다른 이벤트들에 반응을 할 수 없기 때문에 죽은 것처럼 보인다.
2. 만일 액션이 입출력상에 블록이 걸리면, 이 작업을 수행할 새로운 쓰레드를 생성한다. 여러분은 네트웍 연결에 반응이 없는 것처럼 잠재적인 인식 시간동안 사용자 인터페이스가 멈춰있길 원하지 않는다.
3. 만일 여러분이 특정한 시간동안 대기하고 있을 필요가 있다면 이벤트 디스패치 쓰레드에서 잠들면 안된다. 대신에 타이머를 사용하라.
4. 여러분의 쓰레드들에서 여러분이 수행하는 작업은 사용자 인터페이스에 접촉할 수없다. 여러분이 여러분의 쓰레드들을 올리기 전에 UI로부터 모든 정보를 읽어라. 그런 다음, 쓰레드들을 올리고 이벤트 디스패칭 쓰레드로부터 사용자 인터페이스를 갱신하면 일단 쓰레드는 완결되는 것이다.

마지막 규칙은 일반적으로 스윙 프로그래밍에서 *단일 쓰레드 규칙* 이라고 불린다. 다음

은 단일 쓰레드 규칙에 대한 몇가지 예외이다.

1. 소수의 스윙 메소드들은 쓰레드에 안전하다. 그들은 특히 API 문서에서 다음과 같은 문장으로 표기되어 있다. " This method is thread safe, although most Swing method are not." 이들 쓰레드 안전한 메소드들중에서 가장 유용한 것들은 다음과 같다.
   JTextComponent.sexText
   JTextArea.insert
   JTextArea.append
   JTextArea.replaceRange
2. JComponent 클래스의 다음 메소드들은 모든 쓰레드로부터 호출될 수 있다.
   repaint
   revalidate
   repaint 메소드는 repaint 이벤트를 스케쥴한다. 만일 컴포넌트의 컨텐트가 변하거나 컴포넌트의 크기와 위치가 갱시된다면 revalidate 메소드는 컴포넌트의 레이아웃을 마크하고 레이아웃 이벤트를 스케줄한다.(페인트 이벤트처럼 레이아웃 이벤트들은 융합된다. 만일 이벤트 큐에 다중 레이아웃 이벤트들이 있다면 레이아웃은 단지 다시 계산된다.)

노트: 기존의 AWT 는 컴포넌트의 레이아웃을 무효화하고 컴포넌트의 레이아웃을 강제하기위해 invalidate 와 validate 메소드를 가지고 있다. 스윙에서, 여러분은 이 대신에 revalidate 를 호출할 수 있다.

---

3. 여러분은 안전하게 모든 쓰레드에서 이벤트 리스너를 추가하고 제거할 수 있다. 물론 리스너 모델들은 이벤트 디스패팅 쓰레드에서 호출될 것이다.
4. 여러분은 컴포넌트들을 생성하고 이들의 속성을 설정하고 컨텐이너에 추가할 수 있다. 단, 컴포넌트들이 실제화(realize)되지 않아야 한다. 컴포넌트들이 페인트나 밸리리데이션(validation) 이벤트를 받는 다면 실제화된다. 이것은 show,setVisible 또는 pack 메소드가 컴포넌트상에서 호출되자 마자 실제화되거나 컨테이너에 컴포넌트들이 추가될 때 실제화된다. 일단 컴포넌트들이 실제화되면 더 이상 다른 쓰레드에서 이들을 조작할 수 없다.
   특히, show 를 호출하기 전에 main 메소드내에서 응용 프로그램의 GUI 를 생성할 수 있다. 그리고 애플릿 생성자나 init 메소드에서 애플릿의 GUI 를 생성할 수 있다.

이러한 규칙은 복잡해 보니다. 그러나 실제 따르기가 어렵지는 않다. 시간을 소요하는 프로세스를 시작하는 새로운 쓰레드를 시작하는 것은 윕다. 사용자의 요청에 따라 GUI 로부터 필요한 정보를 수집하고 이들을 쓰레드에 전달하고 쓰레드를 시작한다.

```
public void actionPerformed(ActionEvent e)
{       // gather data needed nt thread
        MyThread t = new MyThread(data);
        t.start();
}
```

어려운 부분은 여러분의 쓰레드내에서 진행상황을 알려주는 GUI 를 갱신하는 것과 쓰레드가 종료했을 때 작업의 결과를 표현하는 것이다. 여러분이 여러분 자신의 쓰레드내에서 어떠한 스윙 컴포넌트들도 건드릴 수 없다는 것을 기억하라. 예를 들면, 여러분이 진행 바나 라벨 텍스트를 갱신하길 원한다면 여러분의 쓰레드에서 이들의 값을 설정할 수 없다.

이 문제를 풀기위해, 모든 쓰레드내에서 이벤트 큐에 임의의 액션들을 추가할 수 있게 해주는 두개의 편리한 유틸리티 메소드들이 있다. 예를 들면, 한 쓰레드에서 진행상태를 알려주는 "x% complete"라는 라벨을 주기적으로 갱신하길 원한다고 가정해보자. 여러분은 여러분의 쓰레드로부터 label.setText 를 호출할 수 없다. 그러나 이벤트 디스패팅 쓰레드에서 실행되는 EventQueue 클래스의 invokeLater 와 invokeAndWait 메소드를 사용할 수 있다.

노트: 이들 메소드는 도한 javax.swing.SwingUtilities 클래스에서 이용할 수 있다. 만일 여러분이 JDK1.1 에서 스윙을 사용한다면 해당 클래스를 사용할 필요가 있다.- 메소드들은 JDK1.2 의 EventQueue 에 추가되어 있다.

다음은 여러분이 해야 할 필요가 있는 것이다. 여러분은 스윙 코드를 Runnable 인터페이스를 구현한 클래스의 run 메소드내에 위치해야 한다. 그러면 여러분은 해당 클래스의 객체를 생성하고 이를 invokeLater 나 invokeAndWait 정적 메소드에 전달한다. 예를 들면, 다음은 라벨 텍스를 갱신하는 방법이다. 먼저, run 메소드에서 클래스를 생성한다.

```
public class LabelUpdater implements Runnable
{       public LabelUpdater(JLabel aLabel , int aPercentage)
```

```
        {       label = aLabel;
                percentage = aPercentage;
        }
        public void run()
        {       label.setText(percentage + "complete");
        }
}
```

다음으로 , 객체를 생성하고 이를 invokeLater 메소드에 전달한다.

```
Runnable updater = new LabelUpdater(label , percentage)
EventQueue.invokeLater(updater);
```

invokeLater 메소드는 이벤트가 이벤트 큐에 도달했을 때 즉시 반환한다. run 메소드는 비동기적으로 실행된다. InvokeAndWait 메소드은 run 메소드가 실제 실행될 때까지 대기하고 있다. EventQueue 클래스는 동기화의 자세한 상황을 다룬다. 진행 레벨을 갱신하는 상황에서 invokeLater 메소드가 더 적합하다. 사용자는 완전한 진행상황을 표시하는 것보다 진행 상황을 만드는 작업 쓰레드를 갖는 편이 좋다.

이벤트 디스패치 쓰레드내의 코드를 호출하기 위해, 익명 내부 클래스들은 유용한 지름길을 제공한다. 가령, 앞서 제공한 예제 코드는 다음처럼 단순화될 수 있다:

```
EventQueue.invokeLater(new Runnable()
        {       public void run()
                {       label.setText(percentage + "% complete");
                }
        }
```

노트: invokeLater와 invokeAndWait 메소드는 Runnable 인터페이스를 구현한 객체를 사용한다. 여러분은 이미 Runnable 객체로 부터새로운 쓰레드를 생성하는 방법을 보았다. 그러나 이 경우 run 메소드의 코드는 새로운 쓰레드가 아니라 이벤트 디스패칭 쓰레드내에서 실행된다.

예제 1-9 는 리스트 박스의 내용을 안전하게 변경하기 위해 invokeLater 메소드를 사용하

는 방법을 보여준다. 만일 여러분이 “Good” 버튼을 누르면 쓰레드는 숫자들을 삽입하고 제거한다. 그러나 실제 변경은 이벤트 디스패칭 쓰레드내에서 발생한다.

예제 1-9 : SwingThreadTest.java

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

public class SwingThreadTest
{  public static void main(String[] args)
   {  JFrame frame = new SwingThreadFrame();
      frame.show();
   }
}

class SwingThreadFrame extends JFrame
{   public SwingThreadFrame()
    {   setTitle("SwingThread");
        setSize(400,300);
        addWindowListener(new WindowAdapter()
           {   public void windowClosing(WindowEvent e)
               {   System.exit(0);
               }
           } );
       model = new DefaultListModel();

       JList list = new JList(model);
       JScrollPane scrollPane = new JScrollPane(list);

       JPanel p = new JPanel();
         p.add(scrollPane);
        getContentPane().add(p, "South");

       JButton b = new JButton("Good");
```

```
        b.addActionListener(new ActionListener()
           {  public void actionPerformed(ActionEvent event)
              {  new GoodWorkerThread(model).start();
              }
           });
        p = new JPanel();
        p.add(b);
        b = new JButton("Bad");
        b.addActionListener(new ActionListener()
           {  public void actionPerformed(ActionEvent event)
              {  new BadWorkerThread(model).start();
              }
           });
        p.add(b);

        getContentPane().add(p, "North");
    }

    private DefaultListModel model;
}

class BadWorkerThread extends Thread
{  public BadWorkerThread(DefaultListModel aModel)
   {  model = aModel;
      generator = new Random();
    }

   public void run()
   {  while (true)
      {  Integer i = new Integer(generator.nextInt(10));

         if (model.contains(i))
            model.removeElement(i);
         else
            model.addElement(i);
```

```
            yield();
        }
    }

    private DefaultListModel model;
    private Random generator;
}

class GoodWorkerThread extends Thread
{   public GoodWorkerThread(DefaultListModel aModel)
    {   model = aModel;
        generator = new Random();
    }

    public void run()
    {   while (true)
        {   final Integer i = new Integer(generator.nextInt(10));
            EventQueue.invokeLater(new Runnable()
                {   public void run()
                    {   if (model.contains(i))
                            model.removeElement(i);
                        else
                            model.addElement(i);
                    }
                });
            yield();
        }
    }

    private DefaultListModel model;
    private Random generator;
}
```

java.awt.EventQueue

- static void invokeLater(Runnable runnable)
  지연중인 이벤트들을 처리한 다음에 runnable **객체의** run **메소드가 이벤트 디스패**

치 쓰레드내에서 실행되도록 한다.

- static void invokeAndWait(Runnable runnable)
  지연중인 이벤트들을 처리한 다음에 runnable 객체의 run 메소드가 이벤트 디스패치 쓰레드내에서 실행되도록 한다. 이 콜백은 run 메소드가 중단될 때까지 블록한다.

쓰레드간 통신을 위한 파이프 사용

때때로, 쓰레드사이의 통신 패턴은 아주 단순하다. 생산자(Procedure)라 불리는 하나의 쓰레드는 바이트의 스트림을 생성해 낸다. 소비자(Consumer)라 불리는 다른 쓰레드는 해당 바이트 스트림을 읽고 처리한다. 만일 읽기 위한 바이트 스트림이 없을 때 소비자 쓰레드는 블록된다. 만일 생산자가 소비자가 처리하는 것보다 빨리 많은 데이터를 생성해내면 소비자쓰레드의 작성(Write) 연산은 블록된다. 자바 프로그래밍 언어는 PipedInputStream 과 PipedOutputStream 이라는 통신 패턴을 구현한 일련의 편리한 클래스들을 가지고 있다. (만일 생산자 쓰레드가 바이트 대신에 유니코드 문자 스트림을 생성해 낸다면 일련의 다른 클래스를 사용한다: PipedReader 와 PipedWriter).

파이프를 사용하는 기본적인 이유는 각 쓰레드를 단순하게 유지하기 위해서이다. 생산자 쓰레드는 단순히 스크림에 결과를 보내고 이를 잊어버린다. 소비자는 어디서오는 지 상관할 필요없이 스트림으로부터 데이터를 읽는다. 여러분은 파이프를 사용하여 쓰레드 동기화에 대한 걱정없이 다중 쓰레드들을 서로 연결할 수 있다.

예제 1-10 은 파이프 스트림들을 보여주는 프로그램이다. 우리는 임의 시점에 임의 숫자를 만들어내는 생산자 쓰레드를 가지고 있다. 필터 쓰레드는 입력 숫자를 읽고 연속해서 데이터의 평균을 계산해낸다. 그리고 소비자 쓰레드는 대답을 출력한다.(여러분은 이 프로그램을 멈추기위해 Ctrl+C 를 사용해야 한다.) 그림 1-15 는 쓰레드간에 연결된 파이트와 쓰레드들을 보여준다. 유닉스 사용자는 이러한 파이프 스트림을 유닉스에서 프로세스들을 연결하기위한 파이트들과 동일한 것으로 인식할 것이다.

그림 1-15 : 일련의 연속화된 파이프

예제 1-10 : PipeTest.java

```
import java.util.*;
import java.io.*;
```

```
public class PipeTest
{  public static void main(String args[])
   {   try
       {  /* set up pipes */
            PipedOutputStream pout1 = new PipedOutputStream();
          PipedInputStream pin1 = new PipedInputStream(pout1);

          PipedOutputStream pout2 = new PipedOutputStream();
          PipedInputStream pin2 = new PipedInputStream(pout2);

           /* construct threads */

          Producer prod = new Producer(pout1);
          Filter filt = new Filter(pin1, pout2);
          Consumer cons = new Consumer(pin2);

           /* start threads */

            prod.start();
            filt.start();
            cons.start();
        }
       catch (IOException e){}
   }
}

class Producer extends Thread
{  public Producer(OutputStream os)
   {  out = new DataOutputStream(os);
    }

   public void run()
   {   while (true)
       {   try
           {   double num = rand.nextDouble();
```

```
                out.writeDouble(num);
                out.flush();
               sleep(Math.abs(rand.nextInt() % 1000));
            }
            catch(Exception e)
            {   System.out.println("Error: " + e);
            }
        }
    }

    private DataOutputStream out;
    private Random rand = new Random();
}

class Filter extends Thread
{   public Filter(InputStream is, OutputStream os)
    {   in = new DataInputStream(is);
        out = new DataOutputStream(os);
    }

    public void run()
    {   for (;;)
        {   try
            {   double x = in.readDouble();
                total += x;
                count++;
              if (count != 0) out.writeDouble(total / count);
            }
            catch(IOException e)
            {   System.out.println("Error: " + e);
            }
        }
    }

    private DataInputStream in;
    private DataOutputStream out;
```

```
    private double total = 0;
    private int count = 0;
}

class Consumer extends Thread
{   public Consumer(InputStream is)
    {     in = new DataInputStream(is);
     }

    public void run()
     {    for(;;)
         {     try
             {    double avg = in.readDouble();
                  if  (Math.abs(avg - old_avg) > 0.01)
                 {    System.out.println("Current average is " + avg);
                       old_avg = avg;
                   }
              }
              catch(IOException e)
              {    System.out.println("Error: " + e);
              }
         }
     }

    private double old_avg = 0;
    private DataInputStream in;
}
```

java.io.PipedInputStream

- PipedInputStream()
  파이프 출력 스트림에 아직 연결되지 않은 새로운 파이프 입력 스트림을 생성해낸다.
- PipedInputStream(PipedOutputStream out)
  파이프 출력 스트림으로부터 데이터를 읽는 새로운 파이트 입력 스트림을 생성한다.

전달인자: out **데이터 소스**

- void connect(PipedOutputStream out)
  읽을 데이터를 제공하는 파이프 출력 스트림을 부착한다.
  전달인자: out **데이터 소스**

java.io.PipedOutputStream

- PipedOutputStream()
  파이프 입력 스트림에 아직 연결되지 않은 새로운 파이트 출력 스트림을 생성해낸다.

- PipedOutputStream(PipedInputStream in)
  파이트 입력 스트림으로부터 데이터를 읽는 새로운 파이프 출력 스트림을 생성한다.
  전달인자: in **데이터 소스**

- void connect(PipedInputStream in)
  작성할 데이터를 제공하는 파이프 입력 스트림을 부착한다.
  전달인자: in **데이터 소스**