SJC-P 並行性

スレッド

・各スレッドには優先順位
・新しい実行のスレッドを作成するには 2 通りの方法
　・クラスを Thread のサブクラスであると宣言
　・Runnable インタフェースを実装するクラスを宣言

・Runnable 実装クラスの、run() メソッドを直接呼ばない (単なるメソッドの呼び出しとなり、スレッドは開始されない)。Thread.start() を利用する。

関連メソッド

method	内容
getPriority()	優先順位を返す
setPriority(int newPriority)	優先順位を変更
getState()	状態を返す
interrupt()	このスレッドに割り込み
interrupted()	現在のスレッドが割り込まれているかどうか
isInterrupted()	このスレッドが割り込まれているどうか
isAlive()	このスレッドが生存しているかどうか
isDaemon()	このスレッドがデーモンスレッドであるかどうか
setDaemon(boolean on)	デーモンスレッドまたはユーザースレッドとしてマーク
join()	このスレッドが終了するのを待機
sleep(long millis)	現在実行中のスレッドを、指定されたミリ秒数の間、スリープ
start()	スレッドの実行を開始
yield()	現在実行中のスレッドオブジェクトを一時的に休止させ、ほかのスレッドが実行できるように
notify()	このオブジェクトのモニターで待機中のスレッドを 1 つ再開
notifyAll()	このオブジェクトのモニターで待機中のすべてのスレッドを再開
wait()	ほかのスレッドがこのオブジェクトの notify() メソッドまたは notifyAll() メソッドを呼び出すまで、現在のスレッドを待機

例


```
import java.awt.BorderLayout;

import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.SwingUtilities;

@SuppressWarnings("serial")
public class ThreadTest extends JFrame {
  private JTextField txt1;
  private JTextField txt2;
  public ThreadTest() {
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    txt1 = new JTextField();
    txt2 = new JTextField();
    this.getContentPane().add(txt1, BorderLayout.CENTER);
    this.getContentPane().add(txt2, BorderLayout.SOUTH);

    // 非同期でテキストフィールドの内容をカウントアップするスレッドを作成
    Thread t = new Thread(
        // Thread コンストラクタの引数に Runnable インターフェースの実装クラスを渡す
        new Runnable() {
          public void run() {
            int i = 0;
            while (true) {
              txt2.setText(String.valueOf(i++));
              try { Thread.sleep(500); }
              catch (InterruptedException e) {}
            }
          }
        }
    );
    // 開始 (Thred.start() を呼ぶ )
    t.start();

    this.pack();
    this.setVisible(true);
  }
  public static void main(String[] args) throws Exception {
    SwingUtilities.invokeAndWait(
      new Runnable() {
        public void run() {
          new ThreadTest();
        }
      }
    );
  }
}
```

標準入力を待つスレッドと、入力された値をエコーするスレッドを、wait,notifyAll で協調動作させる。

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.LinkedList;
import java.util.Queue;

public class ConcurrentTest4 {
    Queue<String> queu = new LinkedList<String>();

    public static void main(String[] args) {
        ConcurrentTest4 me = new ConcurrentTest4();
        new Thread(me.new InputProcess()).start();
        new Thread(me.new OutputProcess()).start();
    }

    private class InputProcess implements Runnable {
        public void run() {
            // 共有オブジェクト ( モニター ) を同期指定
```

```
            synchronized (queu) {
                try {
                    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
                    System.out.print("input message >");
                    String line = null;
                    while ((line = reader.readLine()) != null) {
                        queu.offer(line); // キューに挿入
                        // 準備が終わったので、OutputProcess に通知し、自身は wait する
                        queu.notifyAll();
                        System.out.println("[in]waiting...");
                        queu.wait(); // wait を呼び出すとき、現在のスレッドは、このオブジェクトモニター
のオーナでなければならない。
                    }
                } catch (IOException ioe) { ioe.printStackTrace(); }
                  catch (InterruptedException ipe) { ipe.printStackTrace(); }
            }
        }
    }
    private class OutputProcess implements Runnable {
        public void run() {
            synchronized (queu) {
                while (true) {
                    try {
                        if (queu.size() > 0) {
                            System.out.println("echo >" + queu.poll());
                            // 処理が終わったので、InputProcess に通知し、自身は wait
                            queu.notifyAll();
                        }
                        System.out.println("[out]waiting...");
                        queu.wait();
                    } catch (InterruptedException e) { e.printStackTrace(); }
                }
            }
        }
    }
}
```

```
public class ConcurrentTest1 {
   public static void main(String[] args) {
        ConcurrentTest1 ct1 = new ConcurrentTest1();
        new Thread(ct1.new T1()).start();
        new Thread(ct1.new T2()).start();
    }
    private class T1 implements Runnable {
        public void run() {
            for (int i=0;i<5;i++) {
                // こういうロックも取得できる
                // 標準出力への割り込みをブロックし、START-END が対になるように制御
                synchronized (System.out) {
                    System.out.println("T1(" + i + ") START");
                    try { Thread.sleep(1000); }
                    catch (InterruptedException e){ e.printStackTrace(); }
                    System.out.println("T1(" + i + ") END");
                }
            }

        }
    }
    private class T2 implements Runnable {
        public void run() {
            for (int i=0;i<5;i++) {
                synchronized (System.out) {
                    System.out.println("T2(" + i + ") START");
                    try { Thread.sleep(500); }
                    catch (InterruptedException e){ e.printStackTrace(); }
                    System.out.println("T2(" + i + ") END");
                }
            }
        }
    }
}
```

結果

```
T1(0) START
```

```
T1(0) END
T2(0) START
T2(0) END
T1(1) START
T1(1) END
T2(1) START
T2(1) END
T1(2) START
T1(2) END
T2(2) START
T2(2) END
T1(3) START
T1(3) END
  :
```

synchronized がないと、T1、T2 の START-END の間に割り込みが起こる。