Java Threads

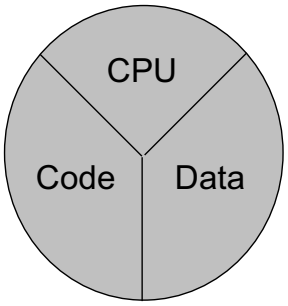
Rui Moreira

Link:

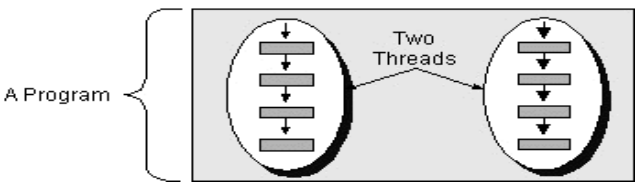
Thread

- Leightweight process
- Single and sequential flow of control within a program
- Runs independently of other threads
- Isolate tasks that will be performed concurrently

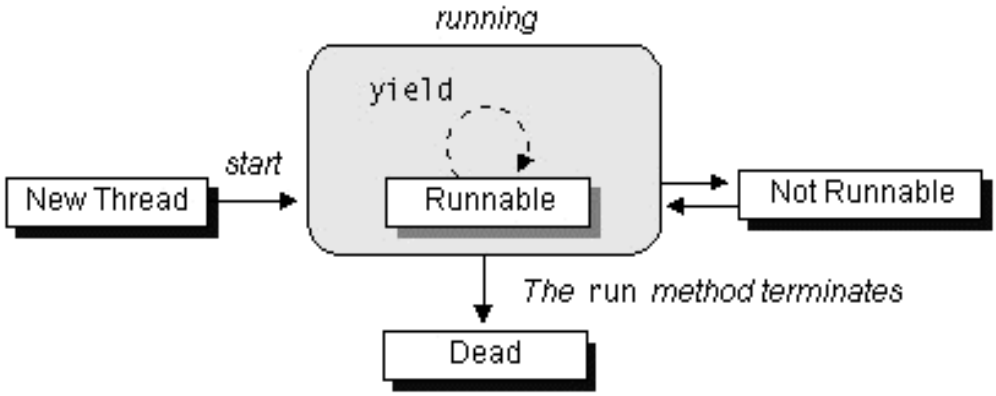
Thread comprises 3 parts:



- Virtual CPU;
- Code executed;
- Data processed.



Thread life cycle



Some of the available methods

- `t.sleep()`
  Put t thread on hold
- `yield()`
  Temporarily pause current thread to allow others to execute
- `t.join()`
  Current thread will wait until t thread terminates
- `t.setPriority(int newPriority)`
  Changes the thread priority (MAX/MIN/NORM_PRIORITY)
- `t.destroy()`
  Destroyu t thread without cleanup
- `t.suspend()`
  Suspend t thread execution
- `t.resume()`
  Resume t thread execution

2 mechanisms to create Threads

- Implement Runnable

```
public class SleepRunnable implements Runnable {}
```

When not possible to extend Thread and another class - Java single inheritance constraint

- Extend Thread & override run()

```
public class SleepThread extends Thread {}
```

Possibility to use “this” operator on the thread class

Implements Runnable

```
public class SleepRunnable implements Runnable {

  public void run() {
    for (int i=0; i<10; i++) {
      Thread current = Thread.currentThread();
      System.out.println("SleepRunnable - run(): +"+i+" "+current.getName());
      try {
        // Put thread to sleep
        current.sleep((long)(Math.random()*1000));
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
    }
    System.out.println("SleepRunnable - run(): done "+current.getName());
  }
}
```

Test Runnable

```
public class TestThreadsApp {
  public static void main (String[] args) {
    // Create threads
    Runnable sr2 = new SleepRunnable();
    Thread spip = new Thread(sr1, "Spip");
    Runnable sr2 = new SleepRunnable();
    Thread spirou = new Thread(sr2, "Spirou");
    // Start threads
    spip.start();
    spirou.start();
  }
}
```


Extends Thread

```
public class SleepThread extends Thread {

  public SimpleThread(String str) { super(str); }

  public void run() {
    for (int i=0; i<10; i++) {
      System.out.println("SleepThread - run(): +"+i+" "+getName());
      try {
        // Put thread to sleep
        sleep((long)(Math.random()*1000));
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    System.out.println("SleepThread - run(): done "+getName());
  }
}
```

Test Thread

```
public class TestThreadsApp {

  public static void main (String[] args) {
    SleepThread spip = new SleepThread("Spip");
    SleepThread spirou = new SleepThread("Spirou");
    spip.start();
    spirou.start();
  }
}
```


Rule of Thumb

Java provides only single inheritance, therefore, if your class must subclass some other class (e.g., Applet) you should use Runnable.

Implement Runnable

```
public class ClockRunnable extends Applet implements Runnable {
  private Thread clockThread = null;
  public void start(){
    if (clockThread==null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
  }
  public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread==myThread) {
      repaint()
      try {Thread.sleep(1000);} catch (InterruptedException e){ /* go to work */ }
    }
  }
  public void stop() { clockThread = null; }

  public void paint(Graphics g) {
    Calendar cal = Calendar.getInstance();
    Date date = cal.getTime();
    DateFormat  dateFormatter = DateFormat.getTimeInstance();
    g.drawString(dateFormatter.format(date), 5, 10);
  }
}
```


Waiting for some thread

```
import java.util.*;

public class ScheduleTaskApp {
  Timer timer = null;

  public ScheduleTaskApp(int seconds) {
     timer = new Timer(seconds);
     System.out.println("ScheduleTaskApp - constructor(): going to schedule task...");
     timer.schedule(new AlertTask(), seconds*1000);
  }

  // Declare a inner class
  class AlertTask extends TimerTask {
    public void run(){
      System.out.println("AlertTask - run(): alert, I am alive...");
      timer.calcel();
    }
    System.out.println("SleepThread - run(): done!");
  }

  public static void main(String[90 args){
    try {
      new ScheduleTaskApp(4);
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Synchronization

- Mechanism to control threads that share data
  e.g. Stack

```
public class Stack {
   private char[] char_stack = new char[10];
   private int top_stack = 0; // next available position

   public void push(char c){
        char_stack[top_stack]=c;
        top_stack++;
   }

   public char pop(){
        top_stack--;
        return char_stack[top_stack];
   }

}
```

Problem:

2 threads concurrently accessing the same Stack object can interrupt each other (in the middle of push/pop calls) and cause stack malfunctions!!

Object lock flag

- Every Java object (instance) has an associated flag – lock flag
- The keyword `synchronized` allows to interact with this flag

```
// When a thread executes the push method and reaches the
// synchronized scope tries to obtain the lock flag for
// executing the protected code otherwise waits for the lock flag.
public void push(char c){
      synchronized(this){
        char_stack[top_stack]=c;
        top_stack++;
      }
}
```

Object lock flag (cont.)

The keyword `synchronized` may be used as

- *variable* modifier

```
private synchronized int top_stack = 0;
```

- *method* modifier

```
// Used for the whole method:
// - may be inefficient for extensive methods - holding lock + time!
// - is explicit - javadoc generates sync info for users of the method!
public synchronized void push(char c){
     char_stack[top_stack]=c;
     top_stack++;
}
```


Thread interaction – wait() & notify()

- Threads may perform concurent tasks or even related/cooperative tasks which implies some interaction between threads
- Every Java object has 2 associated thread queues:
  - One for the threads trying to obtain the lock flag
  - Another for implementing the communication mechanisms of wait() & notify()

Rules for synchronization

- Before calling **wait**(), **notify**() ot **notifyAll**() on a object we must hold the lock flag for that object, i.e., these calls must be done inside **synchronized** blocks

Producer Thread

```
public class ProducerThread extends Thread {
   SyncStack theStack = null;
   public ProducerThread(String n, SyncStack ss){
     super(n);
     this.theStack=ss;
   }
   public void run(){
     char c;
     for(int i=0; i<20; i++){
       c = (char)(Math.random()*26+'A');
       theStack.push(c);
       try {
         // Wait between 0 and 100 millisecs
         Thread.sleep((int)(Math.random()*100));
       } catch (InterruptedException ie) { /* ignore */ }
     }
   }
}
```

Consumer Thread

```
public class ConsumerThread extends Thread {
    SyncStack theStack = null;
    public ConsumerThread(String n, SyncStack ss){
      super(n);
      this.theStack=ss;
    }
    public void run(){
      char c;
      for(int i=0; i<20; i++){
        c = theStack.pop();
        try {
          // Wait between 0 and 1000 millisecs
          Thread.sleep((int)(Math.random()*1000));
        } catch (InterruptedException ie) { /* ignore */ }
      }
    }
}
```



```
public class SyncStack implements SyncStackI {
    private char[] buffer = new char[6];
    private int index = 0;

    public synchronized void push(char c){
           while(index==buffer.length){
             try {
               this.wait();
             } catch (InterruptedException ie) { /* ignore */ }
           }
           this.notify();
           buffer[index]=c;
           index++;
           printStack("push");
    }
    public synchronized char pop(){
           while(index==0){
             try {
               this.wait();
             } catch (InterruptedException ie) { /* ignore */ }
           }
           this.notify();
           index--;
           printStack("pop");
           return buffer[index];
    }
    private void printStack(String method){
           System.out.print("SyncStack - "+method+"() = ");
           for(int c=0; c<index; c++) System.out.print(buffer[c]);
           System.out.println();
    }
}
```

```
// NB: the interface SyncStackI should be
// defined in the file SyncStackI.java
public interface SyncStackI {
    public void push(char c);
    public char pop();
}
```

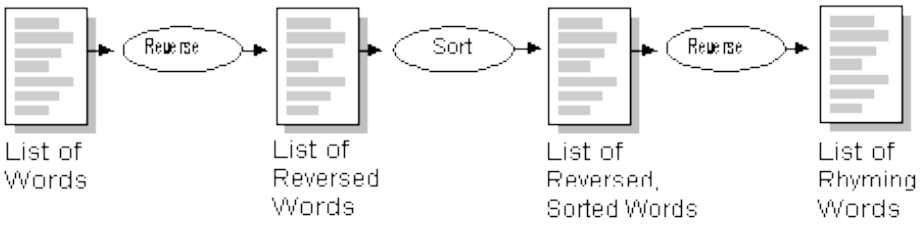
Running Producer-Consumer

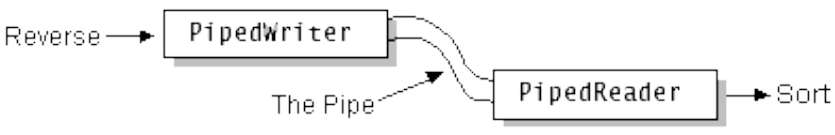
```
public class SyncProdConsApp {

    public static void main(String[] args){
          SyncStack sstack = new SyncStack();
          ProducerThread p = new ProducerThread("Producer", sstack);
          ConsumerThread c = new ConsumerThread("Consumer", sstack);
          p.start();
          c.start();
    }
}
```


Example

http://java.sun.com/docs/books/tutorial/essential/io/pipedstreams.html





```
import java.io.*;
public class RhymeWordsApp {
    public static void main(String[] args) throws IOException {
          FileReader words = new FileReader("words.txt");
          // do the reversing and sorting
          Reader rhymedWords = reverse(sort(reverse(words)));
          // write new list to standard out
          BufferedReader in = new BufferedReader(rhymedWords);
          String input;
          while ((input = in.readLine()) != null) System.out.println(input);
          in.close();
    }
    public static Reader reverse(Reader source) throws IOException {
          BufferedReader in = new BufferedReader(source);
          PipedWriter pipeOut = new PipedWriter();
          PipedReader pipeIn = new PipedReader(pipeOut);
          PrintWriter out = new PrintWriter(pipeOut);
          new ReverseThread(out, in).start();
          return pipeIn;
    }
    public static Reader sort(Reader source) throws IOException {
          BufferedReader in = new BufferedReader(source);
          PipedWriter pipeOut = new PipedWriter();
          PipedReader pipeIn = new PipedReader(pipeOut);
          PrintWriter out = new PrintWriter(pipeOut);
          new SortThread(out, in).start();
          return pipeIn;
    }
}
```



```
import java.io.*;

public class ReverseThread extends Thread {
    private PrintWriter out = null;
    private BufferedReader in = null;
    public ReverseThread(PrintWriter out, BufferedReader in) {
          this.out = out; this.in = in;
    }
    public void run() {
          if (out != null && in != null) {
            try {
              String input;
              while ((input = in.readLine()) != null) {
                out.println(reverseIt(input));
                out.flush();
              }
              out.close();
            } catch (IOException e) {
              System.err.println("ReverseThread run: " + e);
            }
          }
    }
    private String reverseIt(String source) {
          int i, len = source.length();
          StringBuffer dest = new StringBuffer(len);
          for (i = (len - 1); i >= 0; i--) dest.append(source.charAt(i));
          return dest.toString();
    }
}
```

```
import java.io.*;

public class SortThread extends Thread {
    private PrintWriter out = null;
    private BufferedReader in = null;
    public SortThread(PrintWriter out, BufferedReader in) {
          this.out = out; this.in = in;
    }
    public void run() {
          int MAXWORDS = 50;
          if (out != null && in != null) {
            try {
              String[] listOfWords = new String[MAXWORDS];
              int numwords = 0;
              while ((listOfWords[numwords]=in.readLine())!=null) numwords++;
              quicksort(listOfWords, 0, numwords-1);
              for (int i = 0; i < numwords; i++) out.println(listOfWords[i]);
              out.close();
            } catch (IOException e) {
              System.err.println("SortThread run: " + e);
            }
          }
    }

    private static void quicksort(String[] a, int lo0, int hi0) {
          // ...
    }
}
```