Rimbalzi….

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Bounce extends Applet
         implements ActionListener {
   private Canvas canvas;
   private Button b1;
   private Button b2;

   public void init(){
      canvas = new Canvas();
      setLayout(new BorderLayout());
      add("Center", canvas);
      Panel p = new Panel();
      b1= new Button("Start");
      b1.addActionListener(this);
      b2= new Button("Close");
      b2.addActionListener(this);
      p.add(b1); p.add(b2);
      add("South", p);
   }
   public void actionPerformed(ActionEvent e){
       if (e.getSource()== b1){
          Ball b=new Ball(canvas);
          b.bounce();
        }
       else if (e.getSource()== b2)
          System.exit(0);

    }

}  // end def. Class Bounce
```

Con il pulsante Start si introducono palline che rimbalzano in un'area di disegno fino a che non è terminato il loro "ciclo di vita" (1000 movimenti).


```
class Ball{
   private Canvas 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;

   public Ball(Canvas c) {
      box = c;
   }

   public void draw(){
      Graphics g = box.getGraphics();
      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){}
      }
   }
```

```
   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();
   }

  }
```

… non funziona!

- La pallina rimbalza... ma l'applicazione si prende tutte le risorse fino a che le 1000 iterazioni non sono terminate
- Non è possibile interagire con il programma (ad es. premendo il pulsante "Close", o facendo partire una seconda pallina) in nessun modo

- Soluzione: usare threads separati, permettendo all'utente di interagire con il sistema durante l'esecuzione delle iterazioni, per sospenderla o per dare il via ad altre palline

Threads

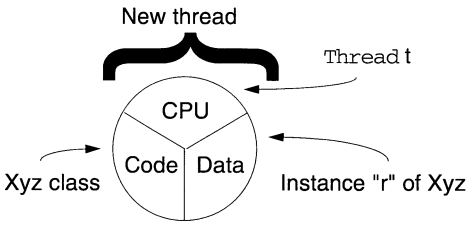
- ***Multitasking*** = più programmi/processi lavorano allo stesso tempo
- ***Multithreading*** = a livello più basso, più threads (che possono condividere dati) lavorano allo stesso tempo.
- Java, come Modula-3, offre primitive per il multithreading nel linguaggio (non in librerie!), a differenza di C e C++ che non supportano multithreading
- Java usa multithreading per eseguire garbage collection in background

Thread

- Una trama di esecuzione (thread) è costituita da tre componenti:
  - *La CPU virtuale*
  - *Il codice che la CPU esegue*
  - *I dati sui quali la CPU lavora*
- Lo stesso codice può essere condiviso da più threads, quando eseguono codice di istanze della stessa classe
- Gli stessi dati possono essere condivisi da più threads, quando accedono allo stesso oggetto
- In Java, la CPU virtuale è incapsulata in un'istanza della classe `Thread`. Quando si costruisce un thread, il codice e i dati che definiscono il suo contesto sono specificati dall'oggetto passato al suo costruttore

Creare un thread

```
public class ProvaThread{
  public static void main(String args[]){
     Xyz r = new Xyz();
     Thread t = new Thread(r);
     t.start();
  }

class Xyz implements Runnable{
  int i;
  public void run() {
    for (i=0; i<20; i++)
      System.out.println("Ciao");
  }
}
```



- Il costruttore di `Thread` prende come parametro un'istanza di `Runnable`
- L'istanza `r` di `Xyz` ha dei suoi dati (ad es. l'intero `i`)
- Poiché l'oggetto `r` è passato al costruttore di thread, il thread `t` "lavorerà" sui dati di `r`: mentre `t` viene eseguito, lavorerà sull'intero `i`
- Un thread non inizia a girare quando viene creato, ma deve essere fatto partire chiamando il metodo `start()`

Esempio (rivisto)

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Bounce extends Applet
        implements ActionListener {
   private Canvas canvas;
   private Button b1;
   private Button b2;

   public void init(){
      canvas = new Canvas();
      setLayout(new BorderLayout());
      add("Center", canvas);
      Panel p = new Panel();
      b1= new Button("Start");
      b1.addActionListener(this);
      b2= new Button("Close");
      b2.addActionListener(this);
      p.add(b1); p.add(b2);
      add("South", p);
   }
```

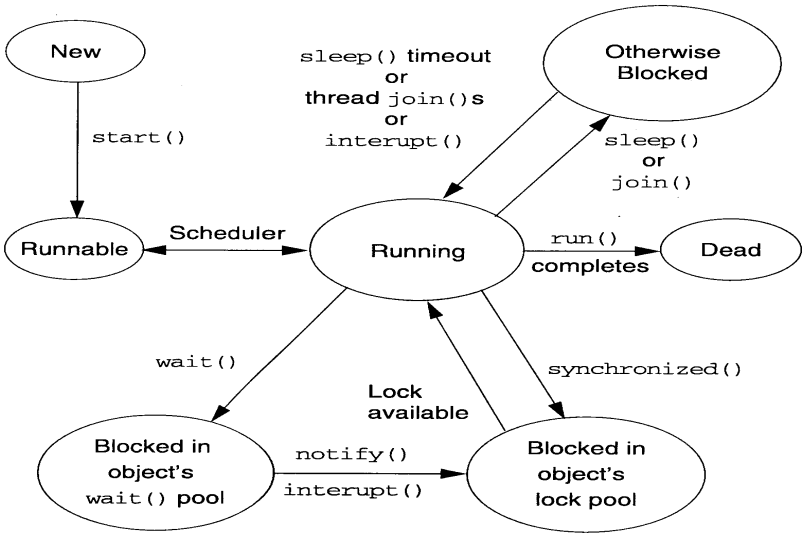
```
public void actionPerformed(ActionEvent e){
     if (e.getSource()== b1){
         Ball b=new Ball(canvas);
         Thread t= new Thread(b);
         t.start();
     }
     else if (e.getSource()== b2)
         System.exit(0);

}
}  // end of Bounce
```



```
class Ball implements Runnable {
   private Canvas 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;

   public Ball(Canvas c) {
      box = c;
   }

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

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

```
   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();
   }

}
```

Gli stati di un thread



Gli stati di un thread

- new
  - quando il thread viene creato con una new, il thread non gira ancora. Ad es. deve essere allocata della memoria.
- runnable
  - quando viene invocato il metodo `start()`. Un thread runnable non è detto che giri effettivamente: questo dipende da quando il sistema operativo glielo consente
- blocked
  - il thread è bloccato in tre modi:
    - nella wait-pool di un oggetto
    - nella lock-pool di un oggetto
    - a causa di uno `sleep()` o `join()`.
- dead
  - quando il metodo run termina

Priorità

- Ogni thread ha una priorità
  - MIN_PRIORITY (posta a 1 nella classe Thread)
  - MAX_PRIORITY(posta a 10 nella classe Thread)
  - NORM_PRIORITY (posta a 5 nella classe Thread)
- Scheduling:
  - Quando lo scheduler decide quale sarà il nuovo thread a girare, selezionerà il thread con massima priorità, e continuerà ad eseguirlo fino a che esso
    - "lascia la precedenza" col metodo `yield()`
    - oppure cessa di essere in stato "runnable"
    - oppure è rimpiazzato da un altro thread la cui priorità è divenuta maggiore
    - oppure è rimpiazzato da un altro thread con la stessa priorità se il sistema supporta time-slicing

Esempio

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Bounce extends Applet
           implements ActionListener {
   private Canvas canvas;
   private Button b1;
   private Button b2;
   private Button b3;

   public void init(){
      canvas = new Canvas();
      setLayout(new BorderLayout());
      add("Center", canvas);
      Panel p = new Panel();
      b1= new Button("Start");
      b1.addActionListener(this);
      b2= new Button("Quick");
      b2.addActionListener(this);
      b3= new Button("Close");
      b3.addActionListener(this);
      p.add(b1); p.add(b2); p.add(b3);
      add("South", p);
   }

   public void actionPerformed(ActionEvent e){
     if (e.getSource()== b1){
         Ball b=new Ball(canvas);
         Thread t = new Thread(b);
         t.setPriority(Thread.MIN_PRIORITY);
         t.start();
     }
     else if (e.getSource()== b2){
         Ball b=new Ball(canvas);
         Thread t = new Thread(b)
         t.setPriority(Thread.MAX_PRIORITY);
         t.start();
     }
     else if (e.getSource()== b3)
         System.exit(0);
   }
}
// la classe Ball è uguale a prima
```

La pallina "veloce" corre di più, e se ce ne sono molte, di veloci, quelle normali si fermano finché queste non sono in "sleep".

sleep() - yield()

- Il codice di un thread può contenere una chiamata al comando `Thread.sleep(k)`, che forza il thread a sospendere la propria esecuzione per almeno `k` millisecondi
- Poiché non è detto che i threads siano time-sliced, è necessario assicurarsi che gli altri threads abbiano la possibilità di continuare l'esecuzione. Questo può essere ottenuto chiamando `sleep()` a intervalli regolari
- Il metodo `yield()` pone il thread corrente nella lista dei thread "runnable" e dà la possibilità ad un altro thread della stessa priorità di essere eseguito. Se nessun altro thread della stessa priorità è runnable, yield() non ha nessun effetto.

Esempio

```
import java.applet.Applet;
import java.awt.*;

public class ProvaYield extends Applet{
  private Trama t1,t2;
  private TextArea output;
  private String [] memo;

  public void init(){
     output=new TextArea(10,30);
     add(output);
}

  public void start(){
     memo = new String[80];
     t1 = new Trama(1,memo);
     t2 = new Trama(2,memo);
     t1.start(); t2.start();
     try {t1.join(); t2.join(); }
          catch(InterruptedException e) {}
     for (int i=0; i<80; i++){
          output.append(memo[i]);
          }
     }
}
```

```
class Trama extends Thread{
  private int id;
  static int index;
  String [] memo;

  public Trama(int n, String [] m){
       id = n;
       memo = m;
       setPriority(Thread.MAX_PRIORITY);
}
  public void run(){
       for (int i=0; i<20; i++){
            memo[index++] = "Thread " +
             id +" prima di yield() \n";
            yield();
            memo[index++] = "Thread " +
             id + " dopo yield() \n";
        }
       }
}
```

```
Thread 1 prima di yield()
Thread 2 prima di yield()
Thread 1 dopo yield()
Thread 1 prima di yield()
Thread 2 dopo yield()
Thread 2 prima di yield()
Thread 1 dopo yield()
Thread 1 prima di yield()
Thread 2 dopo yield()
Thread 2 prima di yield()
.....
```

isAlive() - join()

- Il metodo `isAlive()` è usato per determinare se un thread è ancora disponibile. “Alive” non significa che il thread è “runnable”: restituisce *true* se il thread è iniziato ma non è ancora nello stato “dead”.
- Il metodo `join()` forza il metodo corrente ad aspettare finché il thread sul quale il `join()` è chiamato non termina
- join può essere anche chiamato con un valore di timeout in millisecondi: `t.join(k)` sospende il thread corrente per `k` millisecondi o finché il thread `t` non termina

Sincronizzazione: esempio

```
class BankTest{
   public static void main(String[] args){
      Bank b = new Bank();
      int i;
      for (i = 1; i <= Bank.NACCOUNTS; i++){
        Transaz r = new Transaz(b, i);
        Thread t  = new Thread(r);
        t.start();
      }
   }
}

class Bank{
   public static final int IN_BALANCE = 10000;
   public static final int NACCOUNTS = 80;
   private long[] accounts;
   private int ntransacts;

   public Bank(){
      accounts = new long[NACCOUNTS];
      int i;
      for (i = 0; i < NACCOUNTS; i++)
         accounts[i] = IN_BALANCE;
      ntransacts = 0;
      test();
   }

   public void transfer
        (int from,int to,int amount){
      while (accounts[from] < amount){
         try {  Thread.sleep(5); }
         catch(InterruptedException e) {}
      }
      accounts[from] -= amount;
      accounts[to] += amount;
      ntransacts++;
      if (ntransacts % 5000 == 0) test();
   }

   public void test(){
      int i;
      long sum = 0;

      for (i = 0; i < NACCOUNTS; i++)
         sum += accounts[i];
      System.out.println("Transactions:"
         + ntransacts + " Sum: " + sum);
   }
}
```

```
class Transaz implements Runnable {

   private Bank bank;
   private int from;

   public Transaz (Bank b, int i){
      from = i - 1;
      bank = b;
   }

   public void run(){
      while (true){
         int to = (int)((Bank.NACCOUNTS - 1) * Math.random());
         if (to == from) to = (to + 1) % Bank.NACCOUNTS;
         int amount = 1 + (int)(Bank.IN_BALANCE * Math.random()) / 2;
         bank.transfer(from, to, amount);
         try { Thread.sleep(1); }
         catch(InterruptedException e) {}
      }
   }
}
```

output

```
iniziale  800000
  20000   798538
 700000   799926
 900000   790436
1000000   788144
1500000   786120
2000000   788577
2500000   785879
4000000   783584
```


`synchronized`

- Quando due o più threads accedono alle stesse risorse, si possono creare problemi di sincronizzazione
- Il qualificatore di metodo **`synchronized`** garantisce che il metodo termini prima che un altro thread possa utilizzare lo stesso oggetto

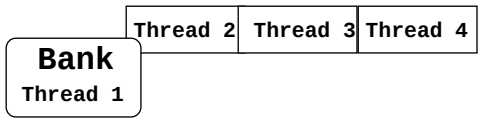
```
public synchronized void transfer (int from,int to,int amount){
   while (accounts[from] < amount){
      try {  Thread.sleep(5); }
      catch(InterruptedException e) {}
   }
   accounts[from] -= amount;
   accounts[to] += amount;
   ntransacts++;
   if (ntransacts % 5000 == 0) test();
}
```

Lock flag - Synchronized

- Ogni oggetto ha un “lock flag” associato. Il modificatore `synchronized` permette di interagire con questo flag, che consente *accesso esclusivo* all’oggetto.
- Quando un thread deve eseguire un metodo synchronized, esamina l’oggetto passato come argomento e cerca di ottenere l’accesso esclusivo dell’oggetto prima di continuare
- Quando il thread ha terminato l’esecuzione del metodo, il lock flag viene rilasciato automaticamente

Monitors

- Ma cosa succede nell’esempio precedente se il conto non contiene abbastanza denaro? Quel thread ha uso esclusivo dell’oggetto di classe Bank! Nessun altro thread può fare depositi...
- Quando si crea un oggetto con metodi sinchronyzed, Java associa a quell’oggetto (detto monitor) una coda dei threads che aspettano di poter accedere all’oggetto.



wait - notify

- Se un thread “runnable” chiama **`wait()`**, il thread entra nella coda di attesa associata all’oggetto particolare sul quale è stato chiamato **`wait()`**.
- Il primo thread nella coda di attesa di un oggetto diventa “runnable” appena un altro thread associato allo stesso oggetto chiama **`notify()`** o **`notifyAll()`**.

Uso di wait e notify

```
public synchronized void transfer (int from,int to,int amount){
   while (accounts[from] < amount){
      try {  wait(); }
      catch(InterruptedException e) {}
   }
   accounts[from] -= amount;
   accounts[to] += amount;
   ntransacts++;
   if (ntransacts % 5000 == 0) test();
   notify();
}
```

Tipicamente…

- Se due o più threads modificano un oggetto, dichiarare synchronized il metodo che realizza le modifiche
- Se un thread deve aspettare che lo stato di un oggetto cambi, deve farlo dentro l'oggetto, non fuori, chiamando `wait()`
- Ogniqualvolta un metodo modifichi lo stato dell'oggetto, deve chiamare `notify()`, per dare una chance ai threads in lista di attesa di vedere se le condizioni dell'oggetto sono cambiate.

La classe java.lang.Thread

- **`Thread()`**: costruisce un nuovo thread
- **`void run()`**: questo metodo deve essere overridden e deve contenere il codice che deve essere eseguito nel thread
- **`void start()`**: inizia il thread, e invoca il metodo `run()`
- **`static void sleep(long millisec)`**: mette "a dormire" il thread per un numero di millisecondi fissato. Osservare che questo metodo è statico
- **`void interrupt()`**: manda al thread una richiesta di interruzione
- **`static boolean interrupted()`**: domanda al thread se è stato interrotto o no

`extends Thread`

- Una alternativa all'implementazione dell'interfaccia `Runnable` è utilizzare direttamente un oggetto di una classe che estende `Threads`
- Anche in questo caso è essenziale fare l'overriding del metodo `run()`

Démoni

- Un démone è un thread che non ha altro ruolo che servire altri threads
- Quando non rimane vivo nessun thread (che non sia un démone), anche i démoni muoiono. Questo accade, ad esempio, quando l'applicazione termina: in tale caso tutti i threads demoni terminano automaticamente;
- **`void setDaemon(boolean on)`**
  questo metodo deve essere chiamato prima dello start del thread

Costruire timers usando threads

```
import java.awt.*;
import java.util.*;
import java.applet.Applet;

public class TimerTest extends Applet {
   public void init(){
      add(new ClockCanvas("Venezia", 0));
      add(new ClockCanvas("New York", 6));
   }
}

interface Timed{
   public void tick(Timer t);
}
```

```
class Timer extends Thread {

   private Timed target;
   private int interval;

   public Timer(Timed t, int i) {
      target  = t;
      interval = i;
      setDaemon(true);
   }

   public void run(){
      while (true){
         try { sleep(interval); }
         catch(InterruptedException e){}
         target.tick(this);
      }
   }
}
```



```
class ClockCanvas extends Canvas implements Timed{
   private int seconds = 0;
   private String city;
   private int offset;
   private final int LOCAL = 0;

   public ClockCanvas(String c, int off){
      city = c; offset = off;
      new Timer(this, 1000).start();
      resize(125, 125);
   }

   public void paint(Graphics 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 tick(Timer t){
      Date d = new Date();
      seconds = (d.getHours() - LOCAL + offset)* 60 * 60 + d.getMinutes() * 60 + d.getSeconds();
      repaint();
   }
}
```