Types of Concurrent Tasks:

Threads

They share the same address space
They can communicate via shared memory

Processes

They do not share the same address space
They communicate via messages

Communication Model:

Shared Memory is the typical approach – all tasks share a common memory area where the variables are stored.

The *Message passing* approach refers to a de-centralized architecture where several tasks are executed on different machines of a network.

A concurrency model with communications based on the *shared memory* approach can be assumed also in a distributed architecture but it is more difficult to implement.

JAVA THREADS

Java provides two ways to create a new thread.
– Extend the Thread class (java.lang.Thread)
– Implement the Runnable interface (java.lang.*Runnable*)

When creating a new thread by extending the Thread class, you should override the run() method.

```
public class FooThread extends Thread
{
     public FooThread()
     {
          // Initialize parameters
     }
     public void run()
     {
          // do something
     }
}
```

To start a new thread, use the inherited method `start()`

```
FooThread ft = new FooThread();
ft.start();
```

`start()` is responsible for two things:

– Instructing the JVM to create a new thread
– Call your Thread object's `run()` method in the new thread

You might think of `run()` as being similar to `main()`
Like `main()`,`run()` defines a starting point for the JVM.
What happens when the `run()` method exits?

– The thread 'ends'.

RUNNABLE INTERFACE

A thread can also be created by implementing the `Runnable` interface instead of extending the `Thread` class.

```
public class FooRunnable implements Runnable
{
     public FooRunnable()
     {
     }
     public void run()
     {
          // do something
     }
}
```

When creating a new thread you need to pass an object that implements Runnable to the constructor of a Thread object, then start the thread as before.

```
FooRunnable fr = new FooRunnable();
new Thread(fr).start()
```

JOIN METHOD

The `join()` method is used to wait until a thread is done.

- The caller of `join()` blocks until the called thread finishes.
- Why might this be bad?
  - Blocking the main thread will make UI freeze.

Other versions of `join()` take in a timeout value: where if thread doesn't finish before timeout, `join()` returns.

Alternatively, you can check on a thread with "`isAlive()`" method

- Returns `true` if running, `false` if finished.

SLEEP METHOD

The `sleep()` method pauses execution of the current thread.

- Periodic actions: need to wait for some period of time before doing the action again.
- Allow other threads to run

Implemented as a class method, so you don't actually need a Thread object

```
Thread.sleep(1000); // Pause here for 1 second.
```

When you call `sleep( )`, it must be placed inside a `try` block because it's possible for `sleep( )` to be interrupted before it times out (`InterruptedException`). This happens if someone else has a reference to the thread and she calls `interrupt( )` on the thread (`interrupt( )` also affects the thread if `wait( )` or `join( )` has been called for it.)

Yielding

If you know that you've accomplished what you need to in your `run( )` method, you can give a hint to the thread scheduling mechanism that you've done enough and that some other thread might as well have the CPU. This hint (and it is a hint—there's no guarantee your implementation will listen to it) takes the form of the `yield( )` method.

Example 1

Write a Java program that runs two *threads* called Paolo and Francesca, respectively. The main *thread* has to call Francesca one second after UPaolo and then wait for Paolo.
Paolo and Francesca execute five iteration of a loop, sleeping for 0.5 sec each time.

// file : ./javaapplication/Filo.java

```
package javaapplication;
import java.io.*;

public class Filo extends Thread {
```

// L'interfaccia *java.lang.Runnable* è anche implementata dalla classe *Thread* che si sta qui utilizzando,
// infatti l'intestazione della classe *Thread* è: """*public class Thread extends Object implements Runnable*"""

// Il metodo *sleep(..)* in *Thread* è definito come: """ *public static void sleep() throws InterruptedException* """
// mentre il metodo *run()* di *Runnable* è definito come: """ *public void run( )* """
// Stiamo estendendo la classe Thread, quindi dovendo richiamare *sleep(..)* nel metodo *run();* si è obbligati
// a gestire l'eccezione sollevata da *sleep*
// perché si sta facendo *overriding* e nell'intestazione di *run()* non è previsto il sollevamento di eccezioni.

```
Filo(String nome){
     super(nome);
}

public void run() {

        for (int i = 1; i <= 5; i++) {
            try {
                 System.out.println(getName()+" : "+i);
                 Thread.sleep(500);
            } catch (InterruptedException e) {
               System.out.println(getName()+" interrotto!"+e.toString());
            }
        } // end for
    } // end run()

} // end class Filo
```

// file : ./javaapplication/demoFilo.java

```
package javaapplication;
import java.io.*;
import java.lang.*;
public class demoFilo {

    public static void main(String[] args) throws InterruptedException {
                                // NON GESTISCO L'ECCEZIONE SOLLEVATA DA Thread.sleep(..)
        Filo p = new Filo("Paolo");
        Filo f = new Filo("Francesca");

        p.start();
        Thread.sleep(1000);
        f.start();
         p.join();
    }
}
```

Example 2

```
package javaapplication3;
public class Main {
   public static void main(String[] args) {
      PrintThread thread1, thread2, thread3, thread4;
      thread1 = new PrintThread("thread1");
      thread2 = new PrintThread("thread2");
      thread3 = new PrintThread("thread3");
      thread4 = new PrintThread("thread4");

      System.err.println("\nStarting Threads");

      thread1.start(); thread2.start();  thread3.start(); thread4.start();

      System.err.println("Threads started");
   }
}
class PrintThread extends Thread {
   private int sleepTime;
   public PrintThread(String nome) {
      super(nome);
      sleepTime = (int) (Math.random()*5000);
      System.err.println("Name: "+this.getName()+"; sleep: "+sleepTime);
   }
   public void run() {
      try {
         System.err.println(this.getName()+" going to sleep");
         Thread.sleep(sleepTime);
      }
      catch (InterruptedException exception) {
         System.err.println(exception.toString());
      }
      System.err.println(this.getName()+" done sleeping");
   }
}
```

```
Name: thread1; sleep: 453
Name: thread2; sleep: 1385
Name: thread3; sleep: 1505
Name: thread4; sleep: 2594

Starting Threads
thread1 going to sleep
Threads started
thread3 going to sleep
thread2 going to sleep
thread4 going to sleep
thread1 done sleeping
thread2 done sleeping
thread3 done sleeping
thread4 done sleeping
```

```
Name: thread1; sleep: 4493
Name: thread2; sleep: 1079
Name: thread3; sleep: 460
Name: thread4; sleep: 101

Starting Threads
thread1 going to sleep
Threads started
thread3 going to sleep
thread2 going to sleep
thread4 going to sleep
thread4 done sleeping
thread3 done sleeping
thread2 done sleeping
thread1 done sleeping
```

ACCESS SYNCHRONIZATION

A **synchronized** method in Java guarantees access in mutual exclusion to a shared resources

Every object **has a lock** associated with it.
If you declare a method to be "`synchronized`", then a **lock** on the referred object will be obtained before executing the method.
Example of synchronized method declaration:

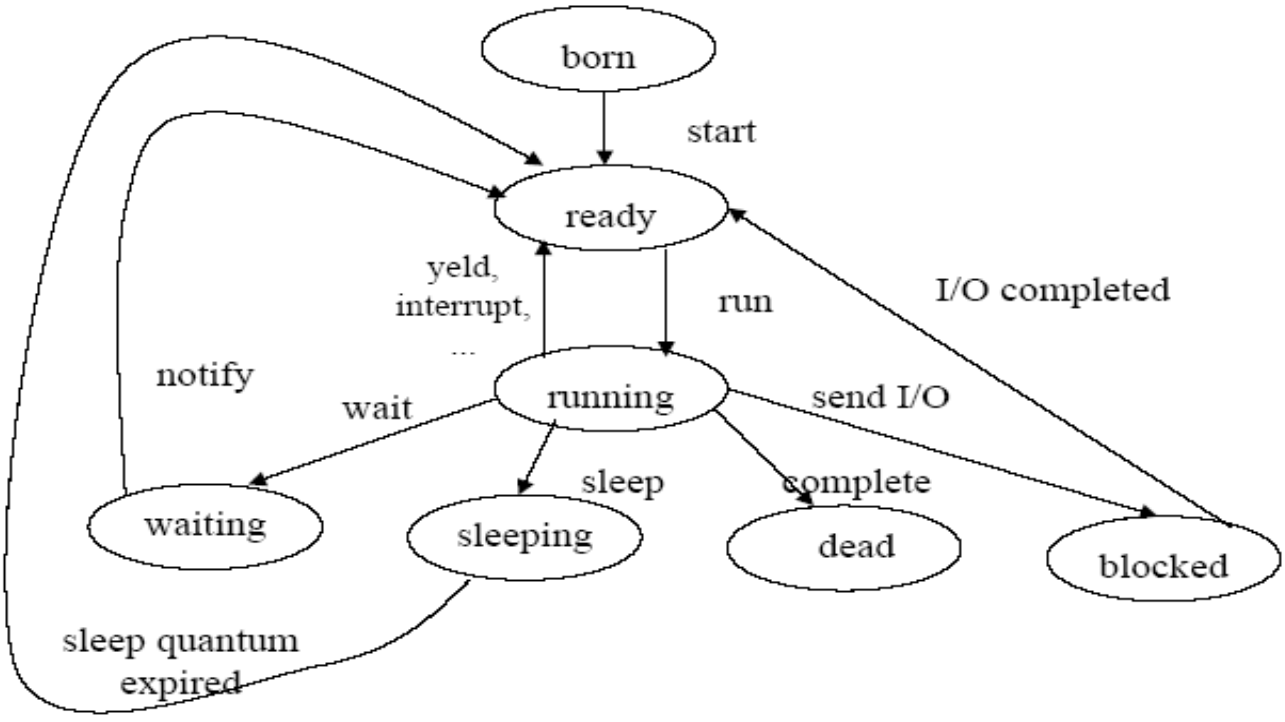
```
synchronized int some_method (int param)
```

Threads' (possibly synchronized) methods are suspended and resumed through **wait** and **notify** methods

A shared object that exploits such synchronization mechanisms implements the concept of MONITOR (shared, passive object)

- Monitor first introduced in concurrent Pascal (by Hoare and Brinch-Hansen)

There is a single lock that is shared by **all** the **synchronized** methods of a particular object, and this lock prevents common memory from being written by more than one thread at a time.



WAIT & NOTIFY METHODS

It's important to understand that `sleep()` (class Thread's method) *does not* release the lock when it is called.
On the other hand, the method `wait()`(class Object's method) *does* release the lock, which means that other `synchronized` methods of the object can be called during a `wait( )`.

Every object maintains a list of "waiting" threads.
A thread that is waiting at/on a particular object is suspended until some other thread wakes it.
`Notify()` or `NotifyAll()` awakens a thread that is waiting.
`wait(), notify(),notifyAll()` are methods associated with ANY object.

Deadlock: Because threads can become blocked and because objects can have synchronized methods that prevent threads from accessing that object until the synchronization lock is released, it's possible for one thread to get stuck waiting for another thread, which in turn waits for another thread, etc. You get a continuous loop of threads waiting on each other, and no one can move.

Starvation: Starvation occurs when a process waits for a resource that continually becomes available, BUT is NEVER assigned to that process because of priority or a flaw in the design of the scheduler.

1 produttore – 1 consumatore;
risorsa : 1 valore intero incapsulato in un oggetto con metodi di accesso NON Sincronizzati.
(il produttore assegnerà in sequenza i valori da 1 a 10)

```
package javaapplication4;
public class SharedCell {
   public static void main(String[] args) {
      HoldIntUnSync h = new HoldIntUnSync();
      ProducerInt p = new ProducerInt(h);
      ConsumerInt c = new ConsumerInt(h);
      p.start();
      c.start();
   }
}
// Thread ProducerInt ... non c'e la keyword public cosi' posso dichiaranlo in questo stesso file
class ProducerInt extends Thread {
   private HoldIntUnSync pHold;
   public ProducerInt(HoldIntUnSync h) {
      super(" ProducerInt ");
      pHold = h;
   }
   public void run() {
      for (int i = 0; i <= 10 ; i++) {
         try {
            Thread.sleep( (int)(Math.random()*3000));
         } catch (InterruptedException e) {
            System.err.println(e.toString());
         }
         pHold.setSharedInt(i);
      }
      System.err.println(getName() + " finished producing values "+ "\n Terminating "+getName());
   }
}
// Thread ConsumerInt
class ConsumerInt extends Thread {
   private HoldIntUnSync cHold;
   public ConsumerInt(HoldIntUnSync h) {
      super(" ConsumerInt ");
      cHold = h;
   }
   public void run() {
      int val, sum = 0;
      do {
         try {
           Thread.sleep( (int)(Math.random()*3000));
         } catch (InterruptedException e) {
            System.err.println(e.toString());
         }
         val = cHold.getSharedInt();
         sum += val;
      } while (val != 10);
      System.err.println(getName() + "retrieved values totaling: "+ sum +"\n Terminating "+getName());
   }
}
```

```
// falso Monitor, non ci sono metodi
// synchronized la JVM non associa nessun lock agli oggetti di questa classe
class HoldIntUnSync {
   private int sharedInt = -1;

   public void setSharedInt(int val) {
      System.err.println(Thread.currentThread().getName()+ " setting sharedInt to "+val);
      sharedInt = val;
   }

   public int getSharedInt() {
      System.err.println(Thread.currentThread().getName()+ " retrieving sharedInt value "+sharedInt);
      return sharedInt;
   }
}
```

```
ProducerInt  setting sharedInt to 0
 ConsumerInt  retrieving sharedInt value 0
 ConsumerInt  retrieving sharedInt value 0
 ProducerInt  setting sharedInt to 1
 ProducerInt  setting sharedInt to 2
 ConsumerInt  retrieving sharedInt value 2
 ConsumerInt  retrieving sharedInt value 2
 ProducerInt  setting sharedInt to 3
 ConsumerInt  retrieving sharedInt value 3
 ProducerInt  setting sharedInt to 4
 ProducerInt  setting sharedInt to 5
 ConsumerInt  retrieving sharedInt value 5
 ConsumerInt  retrieving sharedInt value 5
 ProducerInt  setting sharedInt to 6
 ConsumerInt  retrieving sharedInt value 6
 ProducerInt  setting sharedInt to 7
 ConsumerInt  retrieving sharedInt value 7
 ProducerInt  setting sharedInt to 8
 ConsumerInt  retrieving sharedInt value 8
 ProducerInt  setting sharedInt to 9
 ProducerInt  setting sharedInt to 10
 ProducerInt  finished producing values
 Terminating  ProducerInt
 ConsumerInt  retrieving sharedInt value 10
 ConsumerInt retrieved values totaling: 48
 Terminating  ConsumerInt
```

1 produttore – 1 consumatore;
risorsa : 1 valore intero incapsulato in un oggetto con metodi di accesso Sincronizzati.
(il produttore assegnerà in sequenza i valori da 1 a 10)

```
package javaapplication5;
public class SharedCell2 {
   public static void main(String[] args) {
      HoldIntSync h = new HoldIntSync();
      ProducerInt p = new ProducerInt(h);
      ConsumerInt c = new ConsumerInt(h);
      p.start();
      c.start();
   }
}
// Thread ProducerInt ... non c'e la keyword public cosi' posso dichiaranlo in questo stesso file
class ProducerInt extends Thread {
   private HoldIntSync pHold;
   public ProducerInt(HoldIntSync h) {
      super(" ProducerInt ");
      pHold = h;
   }
   public void run() {
      for (int i = 0; i <= 10 ; i++) {
         try {
            Thread.sleep( (int)(Math.random()*3000));
         } catch (InterruptedException e) {
            System.err.println(e.toString());
         }
         pHold.setSharedInt(i);
      }
      System.err.println(getName() + " finished producing values "+ "\n Terminating "+getName());
   }
}
class ConsumerInt extends Thread { // Thread ConsumerInt
   private HoldIntSync cHold;
   public ConsumerInt(HoldIntSync h) {
      super(" ConsumerInt ");
      cHold = h;
   }
   public void run() {
      int val, sum = 0;
      do {
         try {
           Thread.sleep( (int)(Math.random()*3000));
         } catch (InterruptedException e) {
            System.err.println(e.toString());
         }
         val = cHold.getSharedInt();
         sum += val;
      } while (val != 10);
      System.err.println(getName() + "retrieved values totaling: "+ sum +"\n Terminating "+getName());
   }
}
```

```
// Monitor
class HoldIntSync {
   private int sharedInt = -1;
   private boolean writeable = true ; // condition variable

   public synchronized void setSharedInt(int val) {
      while (!writeable) {
         try {
               wait();
         } catch (InterruptedException e) {
            System.err.println(e.toString());
            e.printStackTrace();
         }
      } // end while

      System.err.println(Thread.currentThread().getName()+ " setting sharedInt to "+val);

      sharedInt = val;


      writeable = false;
      notify();   // wake up any blocked threads
   }

   public synchronized int getSharedInt() {
      while (writeable) {
         try {
               wait();
         } catch (InterruptedException e) {
            System.err.println(e.toString());
            e.printStackTrace();
         }
      } // end while
      writeable = true;
      notify();   // wake up any blocked threads

      System.err.println(Thread.currentThread().getName()+ " retrieving sharedInt value "+sharedInt);
      return sharedInt;
   }
}
```

```
ProducerInt  setting sharedInt to 0
 ConsumerInt  retrieving sharedInt value 0
 ProducerInt  setting sharedInt to 1
 ConsumerInt  retrieving sharedInt value 1
 ProducerInt  setting sharedInt to 2
 ConsumerInt  retrieving sharedInt value 2
 ProducerInt  setting sharedInt to 3
 ConsumerInt  retrieving sharedInt value 3
 ProducerInt  setting sharedInt to 4
 ConsumerInt  retrieving sharedInt value 4
 ProducerInt  setting sharedInt to 5
 ConsumerInt  retrieving sharedInt value 5
 ProducerInt  setting sharedInt to 6
 ConsumerInt  retrieving sharedInt value 6
 ProducerInt  setting sharedInt to 7
 ConsumerInt  retrieving sharedInt value 7
 ProducerInt  setting sharedInt to 8
 ConsumerInt  retrieving sharedInt value 8
 ProducerInt  setting sharedInt to 9
 ConsumerInt  retrieving sharedInt value 9
 ProducerInt  setting sharedInt to 10
 ProducerInt  finished producing values
 Terminating  ProducerInt
 ConsumerInt  retrieving sharedInt value 10
 ConsumerInt retrieved values totaling: 55
 Terminating  ConsumerInt
```

1 produttore – 1 consumatore;
risorsa : 1 vettore di 5 valori interi incapsulati in un oggetto con metodi di accesso Sincronizzati.
(il produttore assegnerà in sequenza i valori da 1 a 10)

```
package javaapplication6;
public class SharedCell3 {
   public static void main(String[] args) {
      HoldIntSync h = new HoldIntSync();
      ProducerInt p = new ProducerInt(h);
      ConsumerInt c = new ConsumerInt(h);
      p.start();
      c.start();
   }
}
// Thread ProducerInt ... non c'e la keyword public cosi' posso dichiaranlo in questo stesso file
class ProducerInt extends Thread {
   private HoldIntSync pHold;
   public ProducerInt(HoldIntSync h) {
      super(" ProducerInt ");
      pHold = h;
   }
   public void run() {
      for (int i = 0; i <= 10 ; i++) {
         try {
            Thread.sleep( (int)(Math.random()*3000));
         } catch (InterruptedException e) {
            System.err.println(e.toString());
         }
         pHold.setSharedInt(i);
      }
      System.err.println(getName() + " finished producing values "+ "\n Terminating "+getName());
   }
}
class ConsumerInt extends Thread { // Thread ConsumerInt
   private HoldIntSync cHold;
   public ConsumerInt(HoldIntSync h) {
      super(" ConsumerInt ");
      cHold = h;
   }
   public void run() {
      int val, sum = 0;
      do {
         try {
           Thread.sleep( (int)(Math.random()*5000));
         } catch (InterruptedException e) {
            System.err.println(e.toString());
         }
         val = cHold.getSharedInt();
         sum += val;
      } while (val != 10);
      System.err.println(getName() + "retrieved values totaling: "+ sum +"\n Terminating "+getName());
   }
}
```

```
// Monitor
class HoldIntSync {
   private int sharedInt[] = {-1, -1, -1, -1, -1};
   private boolean writeable = true ; // condition variable
   private boolean readable = false ; // condition variable
   private int readIndex = 0; // index of the next element to be read
   private int writeIndex = 0; // index of the next cell to be written

   private void displayBuffer() {
      System.err.print("\t\t buffer: [ ");
      for (int i = 0; i < 4; i++)
         System.err.print(sharedInt[i]+", ");

      System.err.println(sharedInt[4]+" ] ");
   }

   public synchronized void setSharedInt(int val) {
      while (!writeable) {
         try {
               wait();
         } catch (InterruptedException e) {
            System.err.println(e.toString());
            e.printStackTrace();
         }
      } // end while

      sharedInt[writeIndex] = val;
      System.err.print(Thread.currentThread().getName()+ " --- Produced "+val+" into cell "+writeIndex);
      readable = true;
      writeIndex = (writeIndex + 1) % 5;
      System.err.print(" writeIndex: "+writeIndex+" ; readIndex: "+readIndex);
      displayBuffer();

      if (writeIndex == readIndex) {
         writeable = false;
         System.err.println(Thread.currentThread().getName()+ " Buffer Full!");
      }
      notify();   // wake up any blocked threads
   }

   public synchronized int getSharedInt() {
      int temp;
      while (!readable) {
         try {
               wait();
         } catch (InterruptedException e) {
            System.err.println(e.toString());
            e.printStackTrace();
         }
      } // end while
      temp = sharedInt[readIndex];
      sharedInt[readIndex] = -1;
      System.err.print(Thread.currentThread().getName()+" --- Consumed "+temp+" into cell "+writeIndex);
      writeable = true;
      readIndex = (readIndex + 1) % 5;
```

```
        System.err.print(" writeIndex: "+writeIndex+" ; readIndex: "+readIndex);
        displayBuffer();
        if (readIndex ==  writeIndex) {
           readable = false;
           System.err.println(Thread.currentThread().getName()+ " Buffer Empty!");
        }
        notify();   // wake up any blocked threads
        return temp;
    }
}
```

```
ProducerInt  --- Produced 0 into cell 0 writeIndex: 1 ; readIndex: 0          buffer: [ 0, -1, -1, -1, -1 ]
 ProducerInt  --- Produced 1 into cell 1 writeIndex: 2 ; readIndex: 0         buffer: [ 0, 1, -1, -1, -1 ]
 ConsumerInt  --- Consumed 0 into cell 2 writeIndex: 2 ; readIndex: 1         buffer: [ -1, 1, -1, -1, -1 ]
 ProducerInt  --- Produced 2 into cell 2 writeIndex: 3 ; readIndex: 1         buffer: [ -1, 1, 2, -1, -1 ]
 ConsumerInt  --- Consumed 1 into cell 3 writeIndex: 3 ; readIndex: 2         buffer: [ -1, -1, 2, -1, -1 ]
 ConsumerInt  --- Consumed 2 into cell 3 writeIndex: 3 ; readIndex: 3         buffer: [ -1, -1, -1, -1, -1 ]
 ConsumerInt  Buffer Empty!
 ProducerInt  --- Produced 3 into cell 3 writeIndex: 4 ; readIndex: 3         buffer: [ -1, -1, -1, 3, -1 ]
 ConsumerInt  --- Consumed 3 into cell 4 writeIndex: 4 ; readIndex: 4         buffer: [ -1, -1, -1, -1, -1 ]
 ConsumerInt  Buffer Empty!
 ProducerInt  --- Produced 4 into cell 4 writeIndex: 0 ; readIndex: 4         buffer: [ -1, -1, -1, -1, 4 ]
 ConsumerInt  --- Consumed 4 into cell 0 writeIndex: 0 ; readIndex: 0         buffer: [ -1, -1, -1, -1, -1 ]
 ConsumerInt  Buffer Empty!
 ProducerInt  --- Produced 5 into cell 0 writeIndex: 1 ; readIndex: 0         buffer: [ 5, -1, -1, -1, -1 ]
 ConsumerInt  --- Consumed 5 into cell 1 writeIndex: 1 ; readIndex: 1         buffer: [ -1, -1, -1, -1, -1 ]
 ConsumerInt  Buffer Empty!
 ProducerInt  --- Produced 6 into cell 1 writeIndex: 2 ; readIndex: 1         buffer: [ -1, 6, -1, -1, -1 ]
 ProducerInt  --- Produced 7 into cell 2 writeIndex: 3 ; readIndex: 1         buffer: [ -1, 6, 7, -1, -1 ]
 ProducerInt  --- Produced 8 into cell 3 writeIndex: 4 ; readIndex: 1         buffer: [ -1, 6, 7, 8, -1 ]
 ConsumerInt  --- Consumed 6 into cell 4 writeIndex: 4 ; readIndex: 2         buffer: [ -1, -1, 7, 8, -1 ]
 ProducerInt  --- Produced 9 into cell 4 writeIndex: 0 ; readIndex: 2         buffer: [ -1, -1, 7, 8, 9 ]
 ProducerInt  --- Produced 10 into cell 0 writeIndex: 1 ; readIndex: 2        buffer: [ 10, -1, 7, 8, 9 ]
 ProducerInt  finished producing values
 Terminating  ProducerInt
 ConsumerInt  --- Consumed 7 into cell 1 writeIndex: 1 ; readIndex: 3         buffer: [ 10, -1, -1, 8, 9 ]
 ConsumerInt  --- Consumed 8 into cell 1 writeIndex: 1 ; readIndex: 4         buffer: [ 10, -1, -1, -1, 9 ]
 ConsumerInt  --- Consumed 9 into cell 1 writeIndex: 1 ; readIndex: 0         buffer: [ 10, -1, -1, -1, -1 ]
 ConsumerInt  --- Consumed 10 into cell 1 writeIndex: 1 ; readIndex: 1        buffer: [ -1, -1, -1, -1, -1 ]
 ConsumerInt  Buffer Empty!
 ConsumerInt retrieved values totaling: 55
 Terminating  ConsumerInt
```

N produttori – N consumatori; N = 3
risorsa : 1 lista di prodotti con numero minimo e numero massimo di elementi nella lista (minProducts, maxProducts), incapsulati in un oggetto con metodi di accesso Sincronizzati.
I produttori devono ritardare l'aggiunta di elementi se la lunghezza della lista è uguale a maxProducts.
I consumatori devono ritardare il prelievo di elementi se la lunghezza della lista è minore o uguale a minProducts.

```
// . /common/Main.java
package common;

import producer.Producer;
import consumer.Consumer;

public class Main {
	public static void main(String [] args) {
		IFactory factory = new Factory();    //Factory is an implementation of the interface IFactory

		Thread producer0 = new Thread(new Producer(factory, 0));
		Thread producer1 = new Thread(new Producer(factory, 1));
		Thread producer2 = new Thread(new Producer(factory, 2));
		Thread consumer3 =new Thread(new Consumer(factory, 3));
		Thread consumer4 =new Thread(new Consumer(factory, 4));
		Thread consumer5 =new Thread(new Consumer(factory, 5));

		producer0.start();	consumer3.start();
		producer1.start();	consumer4.start();
		producer2.start();	consumer5.start();
	}
}

// . /common/Product.java
package common;
public enum Product {
	ITEM_A
	, ITEM_B
	, ITEM_C
}

// . /common/IFactory.java
package common;
public interface IFactory {
	public Product removeProduct();
	public void addProduct(Product p);
}
```

```
// . /common/Factory.java
package common;
import java.util.LinkedList;
import java.util.List;
public class Factory implements IFactory { // Monitor
	public List<Product> products;
	private final int minProducts;
	private final int maxProducts;

	public Factory() {  this(0, 10);  }
	public Factory(int maxProducts) { this(0, maxProducts);    }
	private int getNumElems() { return products.size(); }

	public Factory(int minProducts, int maxProducts) { // Costruttore
		if (minProducts >= maxProducts) {
			minProducts = 0; maxProducts = 10;
		}
		this.minProducts = minProducts;
		this.maxProducts = maxProducts;
		this.products = new LinkedList<Product>();
	}
	public  synchronized Product removeProduct() {
		Product prod;
		while (getNumElems() <= minProducts) {
			try {
				wait();
			} catch (InterruptedException e) {
				 /* does nothing */
			}
		}
		prod = products.remove(getNumElems() - 1);
		notify();    // wake up any blocked threads
		return prod;
	}
	public synchronized void addProduct(Product prod) {
	    while (getNumElems() == maxProducts) {
			try {
				 wait();
			} catch (InterruptedException e) {
				/* does nothing */
			}
		}
		products.add(prod);
		notify();
	}
}
```

```
// . /consumer/IConsumer.java
package consumer;
import common.Product;

public interface IConsumer extends Runnable {
	public Product consume ();
}
// . /consumer/Consumer.java
package consumer;
import common.IFactory;
import common.Product;

public class Consumer implements IConsumer {
	private IFactory factory;
	private int threadID;

	public Consumer(IFactory factory, int threadID) {
		this.factory = factory;
		this.threadID = threadID;
	}

	public void run() {
		Product consumed;

		while(true) {
			try {
				Thread.sleep((long)(Math.random() * 5000));
			} catch (InterruptedException e) {
				continue;
			}

			consumed = consume();
			System.out.println("[THREAD  "+this.threadID+",  CONSUMER] : " +
					"removed a "+consumed.name());
		}  // end while
	}  // end run

	public Product consume() {
		return factory.removeProduct();
	}
}  // end Consumer
```

```
// . /producer/IProducer.java
package producer;
public interface IProducer extends Runnable {
	public void produce ();
}
// . /producer/Producer.java
package producer;
import java.util.Random;
import common.IFactory;
import common.Product;

public class Producer implements IProducer {
	private IFactory factory;
	private Random rand;
	private int threadID;
	public Producer(IFactory factory, int id) {
		this.factory = factory;
		this.rand = new Random(id);
		this.threadID = id;
	}
	public void produce() {
		Product prod = getRandomElement();

		factory.addProduct(prod);
		System.out.println(("[THREAD  "+this.threadID+",  PRODUCER]:  "+
				"added a "+prod.name()));
	}
	public void run() {
		while (true) {
			try {
				Thread.sleep((long)(Math.random() * 5000));
			} catch (InterruptedException e) {
				continue;
			}
			produce();
		}  // end while
	} // end run
	private Product getRandomElement() {
		switch(rand.nextInt(3)) {
			case 0 : return Product.ITEM_A;
			case 1 :	return Product.ITEM_B;
			default : return Product.ITEM_C;
		}
	}
}  // end  Producer
```

[PRODUCER 2] :added a ITEM_B
[CONSUMER 3] :removed a ITEM_B
[THREAD 2, PRODUCER]: added a ITEM_B
[THREAD 5, CONSUMER] : removed a ITEM_B
[THREAD 1, PRODUCER]: added a ITEM_A
[THREAD 5, CONSUMER] : removed a ITEM_A
[THREAD 0, PRODUCER]: added a ITEM_A
[THREAD 3, CONSUMER] : removed a ITEM_A
[THREAD 0, PRODUCER]: added a ITEM_B
[THREAD 4, CONSUMER] : removed a ITEM_B
[THREAD 2, PRODUCER]: added a ITEM_A
[THREAD 3, CONSUMER] : removed a ITEM_A
[THREAD 1, PRODUCER]: added a ITEM_B
[THREAD 4, CONSUMER] : removed a ITEM_B
[THREAD 1, PRODUCER]: added a ITEM_B
[THREAD 5, CONSUMER] : removed a ITEM_B
[THREAD 0, PRODUCER]: added a ITEM_B
[THREAD 4, CONSUMER] : removed a ITEM_B
[THREAD 2, PRODUCER]: added a ITEM_C
[THREAD 3, CONSUMER] : removed a ITEM_C
[THREAD 1, PRODUCER]: added a ITEM_A
[THREAD 4, CONSUMER] : removed a ITEM_A
[THREAD 0, PRODUCER]: added a ITEM_C
[THREAD 3, CONSUMER] : removed a ITEM_C
BUILD STOPPED (total time: 13 seconds)

Dining philosophers problem

The dining philosophers problem is summarized as five philosophers sitting at a table doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating.

The five philosophers sit at a circular table with a large bowl of spaghetti in the center.

A fork is placed in between each philosopher, and as such, each philosopher has one fork to his or her left and one fork to his or her right.

- As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks.
- The philosopher can only use the fork on his or her immediate left or right.

The philosophers never speak to each other, which creates a dangerous possibility of **deadlock** when every philosopher holds a left fork and waits perpetually for a right fork (or viceversa).

Originally used as a means of illustrating the problem of deadlock, this system reaches deadlock when there is a 'cycle of unwarranted requests'. In this case philosopher P_1 waits for the fork grabbed by philosopher P_2 who is waiting for the fork of philosopher P_3 and so forth, making a circular chain.

```
// ./philosophers/Main.java
package philosophers;

public class Main {
	public static void main (String [] args) {
		Table t = new Table();

		Philosopher Plato = new Philosopher(t, Name.Plato);
		Philosopher Aristotles = new Philosopher(t, Name.Aristotles);
		Philosopher Socrates  = new Philosopher(t, Name.Socrates);
		Philosopher Kan = new Philosopher(t, Name.Kant);
		Philosopher Leibniz = new Philosopher(t, Name.Leibniz);

		Plato.start();
		Aristotles.start();
		Socrates.start();
		Kant.start();
		Leibniz.start();
	}
}

// ./philosophers/Name.java
package philosophers;

public enum Name {
	Plato,
	Aristotles,
	Socrates,
	Kant,
	Leibniz
}
```

```
// ./philosophers/Fork.java
package philosophers;

public class Fork { // Monitor
	private boolean state;

	public Fork() {
		this.state = false;
	}

	public synchronized boolean isTaken() {
		return this.state;
	}

	public synchronized void take() {
		while  (isTaken()) {
			try {
				wait();
			} catch (InterruptedException e) {
				return;
			}
		}
		this.state = true;
	}

	public synchronized void release() {
		this.state = false;
		notify();
	}
}

// ./philosophers/Table,java
package philosophers;

public class Table {
	private Fork forks[];

	public Table() {
		this.forks = new Fork[5];
		for (int i = 0; i < this.forks.length; i++) {
			this.forks[i] = new Fork();
		}
	}

	public void acquireFork(int fork) {
		forks[fork].take();
	}

	public void releaseFork(int fork) {
		forks[fork].release();
	}
}
```

```
// ./philosophers/Philosopher.java
package philosophers;

public class Philosopher extends Thread {
	private Name name;
	private Table table;

	public Name getPhilosopherName() {
		return this.name;
	}
	public Philosopher (Table table, Name name) {
		this.name = name;
		this.table = table;
	}
	private void eat(Table t) {
		t.acquireFork(this.getPhilosopherName().ordinal());

		System.out.println("" + this.getPhilosopherName().name()
				+" [ " + getPhilosopherName().ordinal()+ " ] "
				+ " has taken the fork on his LEFT [fork "
				+ this.getPhilosopherName().ordinal() + "]");

		t.acquireFork((this.getPhilosopherName().ordinal() + 1) % 5);

		System.out.println("" + this.getPhilosopherName().name()
				+" [ " + getPhilosopherName().ordinal()+ " ] "
				+" has taken the fork on his RIGHT [fork "
				+ (this.getPhilosopherName().ordinal() + 1) % 5 + "]");

		System.out.println("" + this.getPhilosopherName().name()
				+" [ " + getPhilosopherName().ordinal()+ " ] "
				+" is eating ...");
		try {
			Thread.sleep((long) (Math.random() * 50000));
		} catch (InterruptedException e) { return; }

		System.out.println("" + this.getPhilosopherName().name()+ " releases the forks.");

		t.releaseFork(this.getPhilosopherName().ordinal());
		t.releaseFork((this.getPhilosopherName().ordinal() + 1) % 5);
	}
	public void run () {
		while (true) {
			System.out.println(""+ name.name()+" [ " + getPhilosopherName().ordinal()+ " ] "
					+" is thinking...");
			try {
				Thread.sleep((long) (Math.random() * 10000));
			} catch (InterruptedException e) {return;}

			System.out.println(""+ name.name()+" [ " + getPhilosopherName().ordinal()+ " ] "
					+" is hungry...");
			eat(this.table);
		}
	}
}
```

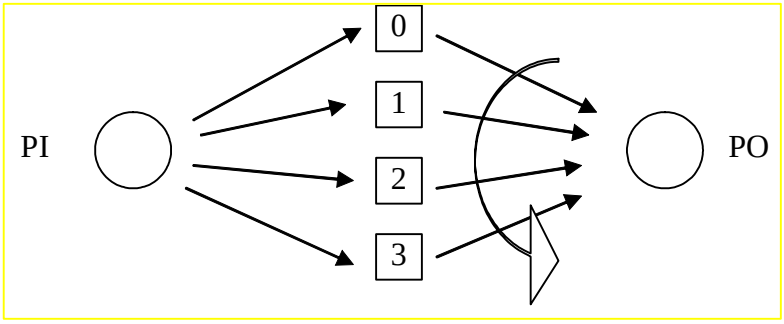
Aristotles [1] is thinking...
Socrates [2] is thinking...
Leibniz [4] is thinking...
Plato [0] is thinking...
Kant [3] is thinking...
Kant [3] is hungry...
Kant [3] has taken the fork on his LEFT [fork 3]
Kant [3] has taken the fork on his RIGHT [fork 4]
Kant is eating ...
Plato [0] is hungry...
Plato [0] has taken the fork on his LEFT [fork 0]
Plato [0] has taken the fork on his RIGHT [fork 1]
Plato is eating ...
Socrates [2] is hungry...
Socrates [2] has taken the fork on his LEFT [fork 2]
Aristotles [1] is hungry...
Leibniz [4] is hungry...
Plato [0] releases the forks.
Plato [0] is thinking...
Aristotles [1] has taken the fork on his LEFT [fork 1]
Plato [0] is hungry...
Plato [0] has taken the fork on his LEFT [fork 0]
Kant [3] releases the forks.
Kant [3] is thinking...
Socrates [2] has taken the fork on his RIGHT [fork 3]
Socrates is eating ...

ESERCIZIO 2
Si realizzi in Java il seguente sistema.

- Un modulo PI esegue ripetutamente le seguenti operazioni: legge da tastiera una coppia di valori <*i, ch*>, dove *i* è un numero tra 0 e 3, *ch* un carattere, e inserisce il carattere *ch* nel buffer *i*-esimo (ognuno dei quattro buffer contiene al più un carattere).

- Un modulo PO considera a turno in modo circolare i quattro buffer e preleva il carattere in esso contenuto, scrivendo in uscita la coppia di valori <*i, ch*> se ha appena prelevato il carattere *ch* dal buffer *i*-esimo.

- L' accesso a ognuno dei buffer è in mutua esclusione; PI rimane bloccato se il buffer a cui accede è pieno, PO rimane bloccato se il buffer a cui accede è vuoto.



Data la seguente sequenza di valori letta da PI, scrivere la sequenza scritta in corrispondenza da PO.

<1, c> <0, b> <2, m> <0, f> <1, h> <3, n>

R. *<0, b> <1, c> <2, m> <3, n> <0, f> <1, h>*

Descrivere brevemente in quali casi si può verificare una situazione di *deadlock* tra PI e PO. Illustrare con un semplice esempio.

R: Deadlock: < 1, a > < 1, b >

```
// riceve l'input via linea di comando,es."0:a 1:b 2:c")
import java.lang.*;
import java.io.*;

class Buf {
  private char ch;
  private boolean full;
  Buf(){ full = false;  }

  public synchronized void put (char item) throws InterruptedException {

     while (full) {
        try{  wait(); } catch (InterruptedException ie) {};
     }
     ch = item;
     full = true;
     notify();
 }
 public synchronized char get() throws InterruptedException {

     while (!full) {
        try{  wait(); } catch (InterruptedException ie) {};
     }
     full = false;
     notify();
     return ch;
 }
}
```

```
class Pi extends Thread {

   private Buf[] buff;
   private String[] commands;
   // commands = ["1,A", "2,B", "0,D"...]
   Pi (Buf[] b, String[] c) {
      buff = b;
      commands = c;
   }

   public void run() {
       int i, index;
       for(i=0; i < commands.length; i++) {
           try {
                 index = (int)(commands[i].charAt(0))-(int)'0';
                 buff[index].put(commands[i].charAt(2));
           } catch (InterruptedException ie) {
                  System.out.println("Monitor Problems -- wait()! ");
           }
       }
   }
}

class Po extends Thread {
    private Buf[] buff;
    Po (Buf[] b) { buff = b; }
    public void run() {
        while (true) {
           try {
               for (int i=0; i < buff.length; i++)
                  System.out.println("Buff "+i+":"+buff[i].get());
           } catch (InterruptedException ie) {
               System.out.println("Monitor Problems -- wait()! ");
           }
        }
    }
}

public class pi_po {

  public static void main(String[] args) throws InterruptedException {
        Buf[] bfs = new Buf[4];

        bfs[0] = new Buf();bfs[1] = new Buf();
        bfs[2] = new Buf();bfs[3] = new Buf();

        Pi pi0 = new Pi(bfs, args);
        Po po0 = new Po(bfs);

        pi0.start();
        po0.start();

    }

}
```

ESERCIZIO 3

Si consideri un conto corrente bancario a cui si può accedere in modo concorrente per effettuare operazioni di versamento e di prelievo.
Due possibilità:

a) È definito un importo massimo prelevabile, e l' accesso al conto corrente da parte di un utente che vuole effettuare un prelievo è possibile solo se, nel conto corrente è presente un importo tale da rendere possibile il prelievo della cifra massima senza "andare in rosso".

b) L' accesso all' utente che vuole effettuare un prelievo è permesso se l' importo che si intende prelevare è inferiore all'attuale disponibilità.

```
class ContoCor{
  ……
  private int disponibile;
  private int soglia;
  ……
  ContoCor() { …… disponibile=0; soglia=2000; …… }
  ……
  public synchronized void deposita(int importo) {
         disponibile += importo;
         notifyAll();
  }
  //  Soluzione a)
  public synchronized void ritira(int importo) {
                                                    // Hp. importo è già verificato essere <=soglia
      while ( !(disponibile >= soglia) )
        try { wait(); }
        catch (InterruptedException e){System.out.println(e.toString());}
      disponibile -= importo;
  }
  ……
//Soluzione b)
//public synchronized void ritira(int importo) {
//        // Hp. importo è già verificato essere <=soglia
//  while ( !(disponibile >= importo) )
//    try { wait(); }
//    catch (InterruptedException e){System.out.println(e.toString());}
//  disponibile -= importo;
//}
}
```