Solution pseudocode:

```
while(true) {
    // Initially, thinking about life, universe, and everything
    think();

    // Take a break from thinking, hungry now
    pick_up_left_fork();
    pick_up_right_fork();
    eat();
    put_down_right_fork();
    put_down_left_fork();

    // Not hungry anymore. Back to thinking!
}
```

Deadlock example:

Here in this case Though it seems that solution is correct, there's an issue of a deadlock arising.

A deadlock is a situation where the progress of a system is halted as each process is waiting to acquire a resource held by some other process.

in this situation, each of the *Philosopher*s has acquired his left fork, but can't acquire his right fork, because his neighbor has already acquired it. This situation is commonly known as the **circular wait** and is one of the conditions that results in a deadlock and prevents the progress of the system

```
public class DiningPhilosophers {


  public static void main(String[] args) throws Exception {


    Philosopher[] philosophers = new Philosopher[5];

    Object[] forks = new Object[philosophers.length];


    for (int i = 0; i < forks.length; i++) {

      forks[i] = new Object();

    }


    for (int i = 0; i < philosophers.length; i++) {

      Object leftFork = forks[i];

      Object rightFork = forks[(i + 1) % forks.length];


      philosophers[i] = new Philosopher(leftFork, rightFork);


      Thread t

        = new Thread(philosophers[i], "Philosopher " + (i + 1));

      t.start();
```

```
    }
  }
}
```

Deadlock solution:

Hence, to avoid a deadlock situation we need to make sure that the circular wait condition is broken .. All Philosophers reach for their left fork first, except one who first reaches for his right fork.

```
public class DiningPhilosophers {

  public static void main(String[] args) throws Exception {

    final Philosopher[] philosophers = new Philosopher[5];
    Object[] forks = new Object[philosophers.length];

    for (int i = 0; i < forks.length; i++) {
      forks[i] = new Object();
    }

    for (int i = 0; i < philosophers.length; i++) {
      Object leftFork = forks[i];
      Object rightFork = forks[(i + 1) % forks.length];

      if (i == philosophers.length - 1) {
```

```
            // The last philosopher picks up the right fork first
            philosophers[i] = new Philosopher(rightFork, leftFork);
        } else {
            philosophers[i] = new Philosopher(leftFork, rightFork);
        }

        Thread t
          = new Thread(philosophers[i], "Philosopher " + (i + 1));
        t.start();
      }
    }
}
```

we introduce the condition that makes the last philosopher reach for his right fork first, instead of the left. This breaks the circular wait condition and we can avert the deadlock. It can be verified by running the code several times, that the system is free from the deadlock situation that occurred before.

Starvation example:

No deadlock, but starvation is possible.. If philosopher 1 is slow to take a fork, and if philosopher 2 is quick to think and pick its forks back up, then philosopher 1 will never get to pick up both forks. A fair solution must guarantee that each philosopher will eventually eat, no matter how slowly that philosopher moves relative to the others

```
class Philosopher extends Thread{
    int Num;
    static int Number=0;
    private Chopstick Chop;
    public Philosopher(Chopstick Chop){
    super();
    this.Chop=Chop;
        Num=Number;
    Number++;
    }

    private void eating(){
        System.out.format("Philosopher\t%d\tis Eating\n", Num);
        try {Thread.sleep(500);
        } catch (InterruptedException e) {}
    }
    private void thinking(){
        System.out.format("Philosopher\t%d\tis Thinking\n", Num);
        try { Thread.sleep(500);
        } catch (InterruptedException e) {}
    }
    public void run(){
        while(true){
            thinking();
            Chop.take();
            eating();
            Chop.release();
        }
    }
}
```

Starvation solution:

```
class Chopstick{
    private boolean[] taking={false,false,false,false,false};
    public synchronized void release(){
    Philosopher phi=(Philosopher) Thread.currentThread();
    int Num=phi.Num;
    System.out.format("Philosopher\t%d\treleases Chopstick\n", Num);
```

```
    taking[Num]=false;
    taking[((Num+1)%5)]=false;
    notifyAll();
    }
    public synchronized void take(){
    Philosopher phi=(Philosopher) Thread.currentThread();
    int Num=phi.Num;
    while(taking[((Num+1)%5)] || taking[Num]){
            try {wait();} catch (InterruptedException e){}
        }
        System.out.format("Philosopher\t%d\ttakes Chopstick\n", Num);
        taking[Num]=true;
        taking[((Num+1)%5)]=true;
    }
}
```

The previous source code represent The sleep_for() function simulates the time normally spend with business logic which gives permission to only one philosopher at a time until the philosopher has picked up both of their forks. Putting down a fork is always allowed where if a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

real world application and how did apply the problem:

A problem like this we can face in our daily life in any kind of dealings that need resources to be shared among many people.

One of this problems is Cooperating processes that need to share limited resources like: ***Money transfers between bank accounts.***

As the problem is that there are several bank accounts that exchange transactions between them, but there should not be more than two accounts that exchange transactions at the same time