```
package bank2;

// This example has a race condition -- you have a very good chance of losing money.
// To fix it, you must ensure that only one thread at a time can deposit/withdraw money.
// One of two ways to ensure this:

// Option A: declare the deposit & withdraw methods as "synchronized."  This forces whatever
//           thread calls deposit or withdraw to own the object's lock before running the
//           deposit or withdraw code.

// Option B: put a "synchronized (acc)" block around the acc.deposit(1) line inside of run().
//           This will have the exact same effect as option A, because it forces the thread
//           that calls deposit to own the lock for the bank account object.

class BankAccount {

        private int balance = 0;

        public void deposit(int x) {
                balance += x;
        }

        public void withdraw(int x) {
                if (balance >= x) {
                        balance -= x;
                }
        }

        public int getBalance() {
                return balance;
        }
}

class DepositorThread extends Thread {

        private BankAccount acc;

        public DepositorThread(BankAccount acc) {
                this.acc = acc;
        }

        public void run() {
                for (int x = 0; x < 1000; x++) {
                        acc.deposit(1);
                }
                System.out.println("done");
        }
}

public class Race {

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

                BankAccount acc = new BankAccount();

                for (int x = 0; x < 5; x++) {
                        DepositorThread i = new DepositorThread(acc);
                        i.start();
                }

                Thread.sleep(1000); // hack: wait for all the threads to finish
                System.out.println("Account has " + acc.getBalance()); // should have $5000
        }
}
```

```
==========================
package philosophers;

class Fork {}

class Philosopher extends Thread
{
        private Fork left, right;

        public Philosopher(Fork l, Fork r)
        {
                left = l; right = r;
        }

        public void run()
        {
                synchronized (left)
                {
                        System.out.println(Thread.currentThread() + " takes left fork.");
                        try { Thread.sleep(1000);  } catch (InterruptedException e) {}
                        synchronized (right)
                        {
                                System.out.println(Thread.currentThread() + " takes right fork and eats.");
                                try { Thread.sleep(1000);  } catch (InterruptedException e) {}
                        }
                }
        }
}


public class Philosophers {
        public static void main(String args[])
        {
                Fork f1 = new Fork(), f2 = new Fork(), f3 = new Fork(), f4 = new Fork(), f5 = new Fork();
                Philosopher p1 = new Philosopher(f1, f2);
                Philosopher p2 = new Philosopher(f2, f3);
                Philosopher p3 = new Philosopher(f3, f4);
                Philosopher p4 = new Philosopher(f4, f5);
                Philosopher p5 = new Philosopher(f5, f1);

                p1.start();
                p2.start();
                p3.start();
                p4.start();
                p5.start();

        }
}
```

```
======================
// Chef and waiter operate concurrently without
// any communication.

package restaurant;

class PickupArea
{
        // note: bad data hiding!
        // orderNumbers are positive, or 0 for no order waiting
        public int orderNumber;
}

class Chef extends Thread {
        private final PickupArea pickupArea;

        public Chef(PickupArea a) { this.pickupArea = a; }

        public void run() {
                for (int orderNum = 1; orderNum <= 10; orderNum++) {
                        try { // simulate time to cook
                                Thread.sleep((int) (Math.random() * 1000));
                        }
                        catch (InterruptedException e) {}
                        System.out.println("Cooked order #" + orderNum);
                        pickupArea.orderNumber = orderNum;
                }
        }
}

class Waiter extends Thread {
        private final PickupArea pickupArea;

        public Waiter(PickupArea a) { this.pickupArea = a; }

        public void run() {
                for (int orderNum = 1; orderNum <= 10; orderNum++) {
                        // retrieve an order
                        int order = pickupArea.orderNumber;
                        // reset the pickup area
                        pickupArea.orderNumber = 0;
                        // serve food
                        System.out.println("Served order #" + order);
                        try { // simulate time to serve
                                Thread.sleep((int) (Math.random() * 1000));
                        }
                        catch (InterruptedException e) {}
                }
        }
}

public class Restaurant {
        public static void main(String args[])
        {
                PickupArea area = new PickupArea();
                Chef chef = new Chef(area);
                chef.start();
                Waiter waiter = new Waiter(area);
                waiter.start();
        }
}
```

```
=======================
// busy waits -- deadlock because one thread hogs
// the scheduler and never switches to the other thread.
package restaurant15;

class PickupArea
{
        // note: bad data hiding!
        // orderNumbers are positive, or 0 for no order waiting
        public int orderNumber;
}

class Chef extends Thread {
        private final PickupArea pickupArea;

        public Chef(PickupArea a) { this.pickupArea = a; }

        public void run() {
                for (int orderNum = 1; orderNum <= 10; orderNum++) {
                        try { // simulate time to cook
                                Thread.sleep((int) (Math.random() * 1000));
                        }
                        catch (InterruptedException e) {}

                        // wait until the pickup area is available (has no order in it)
                        while (pickupArea.orderNumber > 0) { }

                        synchronized (pickupArea) {
                                System.out.println("Cooked order #" + orderNum);
                                pickupArea.orderNumber = orderNum;
                        }
                }
        }
}

class Waiter extends Thread {
        private final PickupArea pickupArea;

        public Waiter(PickupArea a) { this.pickupArea = a; }

        public void run() {
                for (int orderNum = 1; orderNum <= 10; orderNum++) {

                        // wait until there is food in the pickup area
                        while (pickupArea.orderNumber == 0) { }

                        synchronized (pickupArea) {
                                // retrieve an order
                                int order = pickupArea.orderNumber;
                                // reset the pickup area
                                pickupArea.orderNumber = 0;

                                // serve food
                                System.out.println("Served order #" + order);
                        }

                        try { // simulate time to serve
                                Thread.sleep((int) (Math.random() * 1000));
                        }
                        catch (InterruptedException e) {}
                }
        }
}

public class Restaurant {
        public static void main(String args[])
        {
                PickupArea area = new PickupArea();
                Chef chef = new Chef(area);
                chef.start();
                Waiter waiter = new Waiter(area);
                waiter.start();
        }
}
```

```
========================
// Restaurant 2: One chef, one waiter, correctly synched.

package restaurant2;

class PickupArea
{
        // note: bad data hiding!
        // orderNumbers are positive, or 0 for no order waiting
        public int orderNumber = 0;
}

class Chef extends Thread {
        private final PickupArea pickupArea;

        public Chef(PickupArea a) { this.pickupArea = a; }

        public void run() {
                for (int orderNum = 1; orderNum <= 10; orderNum++) {
                        try {
                                // simulate time to cook
                                Thread.sleep((int) (Math.random() * 1000));

                                synchronized (pickupArea) {
                                        // wait until the pickup area is free
                                        while (pickupArea.orderNumber > 0) {
                                                System.out.println("Chef: is waiting");
                                                pickupArea.wait();
                                                System.out.println("Chef: woke up");
                                        }
                                        // we are now guaranteed that the pickup area is empty.
                                        // since we own the pickup area's lock, nobody could have
                                        // changed it between the end of the wait() above and here.

                                        // put the food in the pickup area.
                                        pickupArea.orderNumber = orderNum;
                                        System.out.println("Chef: Sent out order #" + orderNum);

                                        // signal the waiter to come get it
                                        pickupArea.notifyAll();
                                        System.out.println("Chef: Waiter notified of order #" + orderNum);
                                }
                        } catch (InterruptedException e) {
                        }
                }
        }
}

class Waiter extends Thread {
        private final PickupArea pickupArea;

        public Waiter(PickupArea a) { this.pickupArea = a; }

        public void run() {
                for (int orderNum = 1; orderNum <= 10; orderNum++) {
                        try {
                                int order;
                                synchronized (pickupArea) {
                                        // wait until the pickup area has food
                                        while (pickupArea.orderNumber == 0) {
                                                System.out.println("Waiter: is waiting");
                                                pickupArea.wait();
                                                System.out.println("Waiter: woke up");
                                        }

                                        // we are now guaranteed that the pickup area has food.
                                        // since we own the pickup area's lock, nobody could have
                                        // changed it between the end of the wait() above and here.

                                        // get the food in the pickup area and clear the area.
                                        order = pickupArea.orderNumber;
                                        pickupArea.orderNumber = 0;
                                        System.out.println("Waiter: Picked up order #" + order);

                                        // signal the chef that the pickup area is free.
                                        pickupArea.notifyAll();
                                        System.out.println("Waiter: Notified chef of open pickup area.");
                                }
                                // simulate time to serve the food
                                Thread.sleep((int) (Math.random() * 1000));
                        } catch (InterruptedException e) {}
                }
        }
}

public class Restaurant {
```

```
        public static void main(String args[])
        {
                PickupArea area = new PickupArea();
                Chef chef = new Chef(area);
                chef.start();
                Waiter waiter = new Waiter(area);
                waiter.start();
        }
}

=============
// Restaurant 3: One chef, two waiters, correctly synched.

package restaurant3;

class PickupArea
{
        // note: bad data hiding!
        // orderNumbers are positive, or 0 for no order waiting
        public int orderNumber = 0;
}

class Chef extends Thread {
        private final PickupArea pickupArea;

        public Chef(PickupArea a) { this.pickupArea = a; }

        public void run() {
                for (int orderNum = 1; orderNum <= 10; orderNum++) {
                        try {
                                // simulate time to cook
                                Thread.sleep((int) (Math.random() * 1000));

                                synchronized (pickupArea) {
                                        // wait until the pickup area is free
                                        while (pickupArea.orderNumber > 0) {
                                                System.out.println("Chef: is waiting");
                                                pickupArea.wait();
                                                System.out.println("Chef: woke up");
                                        }
                                        // we are now guaranteed that the pickup area is empty.
                                        // since we own the pickup area's lock, nobody could have
                                        // changed it between the end of the wait() above and here.

                                        // put the food in the pickup area.
                                        pickupArea.orderNumber = orderNum;
                                        System.out.println("Chef: Sent out order #" + orderNum);

                                        // signal the waiter to come get it
                                        pickupArea.notifyAll();
                                        System.out.println("Chef: Waiter notified of order #" + orderNum);
                                }
                        } catch (InterruptedException e) {
                        }
                }

                // end of day: close restaurant
                synchronized (pickupArea) {
                        try {
                                // wait until the pickup area is free
                                while (pickupArea.orderNumber > 0) {
                                        System.out.println("Chef: is waiting");
                                        pickupArea.wait();
                                        System.out.println("Chef: woke up");
                                }
                                // we are now guaranteed that the pickup area is empty.
                                // since we own the pickup area's lock, nobody could have
                                // changed it between the end of the wait() above and here.

                                // put "close restaurant" order in.
                                pickupArea.orderNumber = -1;

                                // signal the waiters
                                pickupArea.notifyAll();
                                System.out.println("Chef: Waiters notified of closing");
                        } catch (InterruptedException e) {
                        }
                }
        }
}

class Waiter extends Thread {
        private final PickupArea pickupArea;
        private final int waiterNumber;

        public Waiter(int n, PickupArea a) {
```

```
                this.pickupArea = a; this.waiterNumber = n; }

        public void run() {
                while (true) {
                        try {
                                int orderNum;
                                synchronized (pickupArea) {
                                        // wait until the pickup area has food
                                        while (pickupArea.orderNumber == 0) {
                                                System.out.println("Waiter" + waiterNumber + ": is waiting");
                                                pickupArea.wait();
                                                System.out.println("Waiter" + waiterNumber + ": woke up");
                                        }

                                        // we are now guaranteed that the pickup area has food.
                                        // since we own the pickup area's lock, nobody could have
                                        // changed it between the end of the wait() above and here.

                                        // get the food in the pickup area and clear the area.
                                        orderNum = pickupArea.orderNumber;

                                        // restaurant closing!
                                        if (orderNum == -1)
                                                break;

                                        pickupArea.orderNumber = 0;
                                        System.out.println("Waiter" + waiterNumber + ": Picked up order #" +
orderNum);

                                        // signal the chef that the pickup area is free.
                                        pickupArea.notifyAll();
                                        System.out.println("Waiter" + waiterNumber + ": Notified chef of open
pickup area.");
                                }
                                // simulate time to serve the food
                                Thread.sleep((int) (Math.random() * 1000));
                        } catch (InterruptedException e) {}
                }
                System.out.println("Waiter" + waiterNumber + ": Leaving work.");
        }
}

public class Restaurant {
        public static void main(String args[])
        {
                PickupArea area = new PickupArea();
                Chef chef = new Chef(area);
                chef.start();
                Waiter waiter1 = new Waiter(1, area);
                waiter1.start();
                Waiter waiter2 = new Waiter(2, area);
                waiter2.start();
        }
}
```