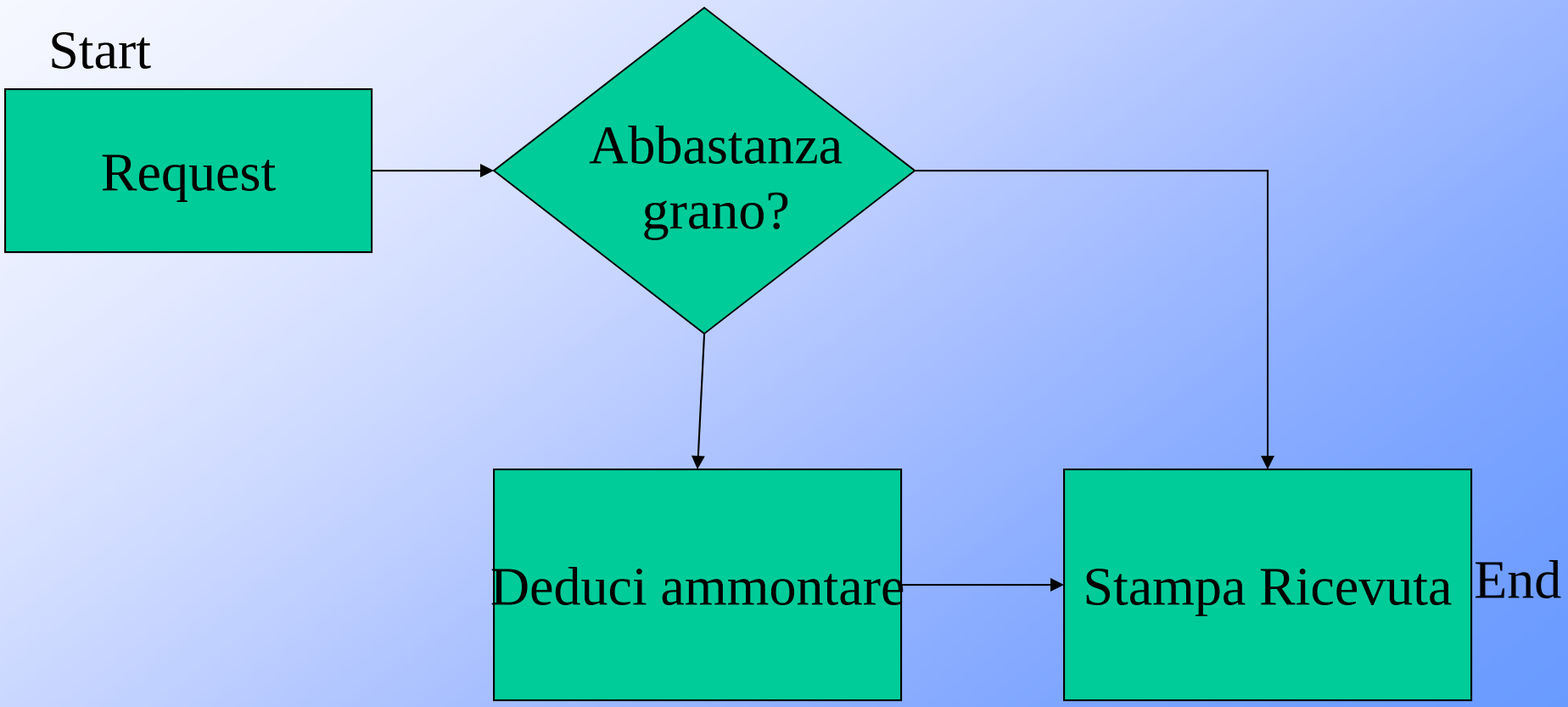
# Ingegneria del Software

## Threads 2

Mauro Migliardi Ph. D.

# Sincronizzazione

- Programma per ATM



# Codice

```
...
public void withdraw(float amount) {
    Account a = getAccount()//<come fa non ci interessa>
    if(a.deduct(amount)) {
        <caccia il grano>;
    }
    printReceipt();
}
```

# Codice di deduct(float amount)

```
...
public boolean deduct(float amount) {
    if(amount <= total) {
        total -= amount;
        return true;
    }
    return false;
}
```

# Funziona?

- Con un singolo utente si’
- Con utenti concorrenti, c’e’ una race condition

- sincronizzazione!
- modificatore ***synchronized*** per il metodo deduct

# Codice di deduct(float amount) raceproofed

```
...
public synchronized boolean deduct(float amount) {
    if(amount <= total) {
        total -= amount;
        return true;
    }
    return false;
}
```

- mutex garantita dalla VM
  - al piu' un thread in una singola JVM puo' eseguire deduct

# E se volessi depositare?

```
public void deposit(float amount) {
    total += amount;
}
```

- Si ripropone il rischio di “corsa” tra deduct e deposit
- devo rendere atomica ogni operazione su ogni conto

# deposit raceproofed

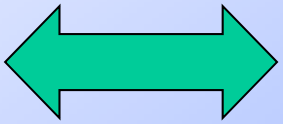
```
public synchronized void deposit(float amount) {
    total += amount;
}
```

- Ma questo e’ un metodo diverso
  - come fa la JVM a sapere che e’ in mutex con deduct?
  - mette per caso in mutex tutti i metodi dichiarati synchronized?
  - indipendentemente da classe, istanza, etc. ?

# Cosa succede in realta’

- Concetto di base: LOCK

•Iniziare l’esecuzione di un metodo synchronized



•Acquisire un lock

•Terminare l’esecuzione di un metodo synchronized

Rilasciare un lock

# Lock di istanza

- due metodi di istanza sono in mutex se e solo se sono invocati sulla stessa istanza
  - e.g. deduct e deposit di un singolo conto sono in mutex
  - deduct e deposit di due conti differenti NON sono in mutex
- cosa succede in un caso come questo?

public static synchronized test() {.....}

- Esiste il concetto di lock di classe
  - Uno solo per tutti i metodi statici
  - Separato e distinto dagli n lock di istanza

# Una classe per Lock

```
public class Lock {
    public boolean getLock() {
        ...
    }
    public boolean releaseLock() {
        ...
    }
}
```

# Implementazione

```
public boolean getLock() {
        while (busyFlag != Thread.currentThread()) {
                if(busyFlag == null) {
                        busyFlag = Thread.currentThread();
                        try {
                                Thread.sleep(100);
                        } catch(InterruptedException ie) { ; }
                }
        }
        return true;
}
```

# Implementazione (Cont.)

```
public boolean releaseLock() {
    if(busyFlag == Thread.currentThread()) {
        busyFlag = null;
        return true;
    }
    return false;
}
```

# Non funziona

Puo’ succedere:

1. thread 1 test busyFlag
2. thread 2 test busyFlag
3. thread 1 setta busyFlag
4. thread 1 aspetta 100ms
5. thread 1 controlla busyFlag ed esce
6. thread 2 setta busyFlag
7. thread 2 aspetta 100ms
8. thread 2 controlla busyFlag ed esce
9. AHIAHIAHIAHIAHIAHIAHI!!!!!!!!!!

# Uso keyword synchronized

```
public synchronized boolean getLock() {
        while (busyFlag != Thread.currentThread()) {
                if(busyFlag == null) {
                        busyFlag = Thread.currentThread();
                        try {
                                Thread.sleep(100);
                        } catch(InterruptedException ie) { ; }
                }
        }
        return true;
}
```

# Soluzione Migliore

```
public synchronized boolean tryLock {
    if(busyFlag == null) {
        busyFlag = Thread.currentThread();
        return true;
    }
    return false;
}
public boolean getLock() {
    while(!tryLock()) {
        try {Thread.sleep(100); } catch(InterruptedException
          ie) { ; }
    }
}
```

# Synchronized block

- Una soluzione alternativa per getLock()

```
public boolean getLock() {
        while(true) {
                synchronized(this) {
                        if(busyFlag == null) {
                                busyFlag = Thread.currentThread();
                                break;
                        }
                }
                try{Thread.sleep(100);}catch(InterruptedException ie){;}
        }
        return true;
}
```

# Vantaggi?

- Controllo lo scope
  - def. dicesi scope di un lock la parte di codice in cui il lock deve essere posseduto per permettere l'esecuzione del codice stesso.
  - non piu' necessariamente il metodo completo
- Controllo l'oggetto utilizzato come lock
  - non necessariamente this, ma un qualunque oggetto Java

# Caveat

- Object != variable

  Object sync = new Object();

```
THREAD A
{
        //Object sync;
        sync = new Object();
        synchronized(sync) {
                ...
        }
}
```

```
THREAD B
{
        //Object sync;
        sync = new Object();
        synchronized(sync) {
                ...
        }
}
```

# Lock annidati

- Cosa accade qui?

```
public class Test {
    public static void main(String[] argv) {
        Test t1 = new Test();
        t.ok();

    }
```

# Class Test (Cont.)

```
public synchronized void ok() {
        ok2();
}
public synchronized void ok2() {
        try {
                Thread.sleep(1000); }
        catch(InterruptedException ie) { ; }
}

// perche’ non si incastra?
```

# Allo stesso modo

```
public void pippo() {
    synchronized(this) {
        synchronized (this) {
            ...
        }
    }
}
```

- Non va in deadlock
- JVM controlla la ownership del lock
- In generale, si puo', controllando la ownership, evitare i deadlock?

# Lock annidati (malamente)

```
public void removeUseless(File file) {
    synchronized (file) {
        if(file.isUseless()) {
            Directory dir = file.getDir();
            synchronized(dir) {
                dir.remove(file);
            }
        }
    }
}
```

# Lock Annidati Malamente (Cont.)

```
public void updateFiles(Directory dir) {
    synchronized(dir) {
        for(File f = dir.first(); f!=null;f=dir.next(f)) {
            synchronized(f) {
                f.update();
            }
        }
    }
}
```

# Risposta: NO!

- E’ necessario controllare l’utilizzo
- Non c’e’ una singola regola da seguire
  - se non quelle viste a OS per fare deadlock avoidance
    - risorse accessibili in modo ordinato
    - algoritmo del banchiere
    - etc.