

# Java Concorrente e Monitor Java

# Perchè CJava?

- La libreria della concorrenza di Java (Doug Lea) è un esempio guida di come integrare oggetti e concorrenza
- Molti dei concetti introdotti d Lea sono stati poi portati su altri linguaggi come C#:
  - Strutture dati concorrenti,
  - Thread pool,
  - task asincroni,
  - Executor e future,
  - Parallel for

# Perchè CJava?

Alcuni linguaggi per sviluppo per App usano  
Concurrent Java

--- Android combina Linux e Java

Spunto per parlare di programmazione ad eventi

--- Swing e concorrenza

Librerie a diversi livelli di astrazione per la  
programmazione distribuita

--- dai Socket a Remote Method Invocation (RMI)

# Remember Java: Struttura File

Struttura di un programma Java “A.java”:

```
public class A {  
    ...  
    public A(int n){... }    Il costruttore della classe A  
    ...  
    public static void main(String[] args) {    Questo è il MAIN!!  
    ....  
}  
}
```

# Remember Java: Ereditarietà singola ...

```
import java.util.*;
class Impiegato
{
    private String Nome;

    public Impiegato(String n)
    {...}
    public void Stampa()
    {...}
}
```

```
class Boss extends Impiegato
{
    private String Segretaria;
    public Impiegato(String n) {
        super(n);
        Segretaria=" ";
    }
    ...
}
```

## ...vs Ereditarietà multipla

```
import java.util.*;
class Impiegato
{
    private String Nome;

    public Impiegato(String n)
    {...}
    public void Stampa()
    {...}
}
```

```
public interface GolfPlayer
{ void campo(String s);
}
```

```
class Boss extends Impiegato
implements GolfPlayer
{
    private String Segretaria;
    public Impiegato(String n) {
        super(n); Segretaria=" ";
    }
    void campo(String s){...}
    ...
}
```

# Remember Java: Creare Oggetti

Struttura di un programma Java “A.java”:

```
public class MyB {...}
```

```
public class MyA {
```

```
....
```

```
public static void main(String[] args) {
```

*Questo è il MAIN*

```
MyB o1=new MyB();
```

*Chiamo il costruttore di MyB*

```
}
```

```
}
```

**3 fasi:**

**Dichiarazione,**

**Creazione (new),**

**Inizializzazione (con il costruttore)**

# Multithreading in Java

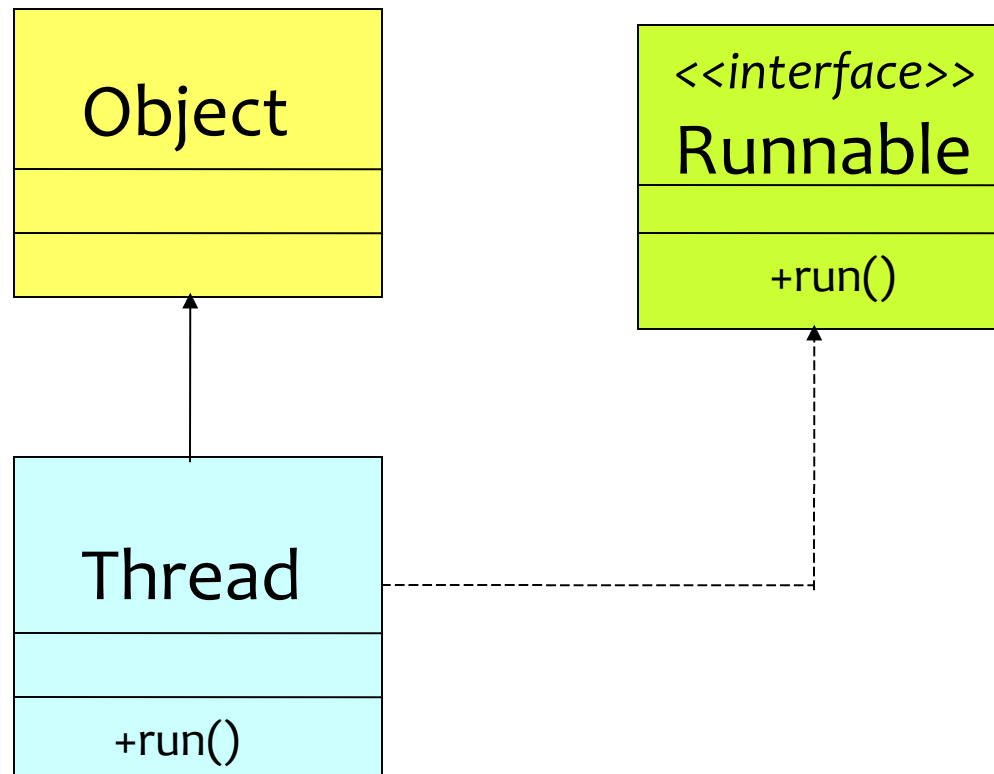
- Java supporta i thread in maniera indipendente dalla piattaforma sottostante attraverso la classe Thread
- Tutti gli oggetti (pubblici) possono essere condivisi
- L'ordine di esecuzione dei thread Java non è determinato a priori (potenziali race condition)
- Il linguaggio fornisce costrutti di sincronizzazione specifiche del Java Memory Model (modello della memoria di Java)



# Thread e programmi Java

- I programmi Java “normali” vengono eseguito in un singolo thread il cui comportamento è definito dal metodo “main”
- Il main può creare altri thread (che vengono eseguiti in parallelo al main) usando istanze della classe Thread (quindi per creare thread si può usare **new**!)
- Il programma non termina fino a quando tutti i thread non hanno finito l'esecuzione
- La JVM (Java Virtual Machine) può associare kernel thread ai thread Java oppure simulare il multithreading

# Classe java.lang.Thread



# Metodi della Classe Thread

- Oggetto di tipo Thread = oggetto che esegue un task in modo *asincrono*
- Costruttori
  - Thread()
  - Thread(Runnable Obj)
  - Thread(Runnable Obj, String Name)

# Metodi della Classe Thread

## run

- Il codice di esecuzione del thread
- Ridefinito nelle sottoclassi di Thread o nelle classi che implementano l'interfaccia Runnable
- Non solleva eccezioni

# Metodi della Classe Thread

**run**

- In
- Ride
- che
- Non

```
public class MyThread extends Thread {  
    String name;
```

```
    public MyThread(String n){  
        this.name=n;  
    }
```

```
    public void run() {  
        System.out.println(name);  
    }  
    ...  
}
```

# Metodi della Classe Thread

## **start**

- Lancia il thread e ritorna al chiamante
- Richiama **run**
- Richiamare **start** due volte genera un errore

# Come usare i thread

- Derivare una nuova classe da **Thread**
- Ridefinire il metodo **run()**
- Creare un oggetto di tale sottoclasse
- Richiamare **start()** su tale oggetto

```
public class MyThread extends Thread {  
    ...  
  
    public MyThread(String n){...}  
    public void run() {...}  
  
    public static void main(String[] args) {  
  
        MyThread t1=new MyThread("thread 1");  
        //t1 non e' ancora in esecuzione  
  
        t1.start();  
        ...  
        // da qui in poi main e thr1 eseguono  
        // in concorrenza  
    }  
}
```



# Interfaccia Runnable

- Java non supporta l'ereditarietà multipla
- Multithreading per le classi già derivate:
  - Implementare l'interfaccia **Runnable**
  - Implementare il metodo **run()**
  - Creare un nuovo **Thread** passandogli un oggetto **Runnable**
- Oggetto Runnable = Task da eseguire
- Oggetto Thread = thread che esegue un task

# Da un oggetto con nome...

```
public class NamedObject {  
    String name;  
  
    public NamedObject(String n){  
        this.name=n;  
    }  
}
```

## ... ad un thread con nome

```
public class MyRunnable extends NamedObject  
    implements Runnable {
```

```
    public MyRunnable(String n){ super(n); }
```

```
    public void run() { ... }
```

```
    public static void main(String[] args) {
```

```
        Runnable t1= new MyRunnable("thread 1");
```

```
        Thread thr1 = new Thread(t1);
```

```
        thr1.start();
```

```
        ...
```

```
    }
```

```
}
```

# Run e Start

...

```
if (separateThread) {  
    t=new MyRunnable("thread 1");  
    (new Thread(t)).start();  
        // lancio un nuovo thread  
} else {  
    o=new MyRunnable("only main thread");  
    o.run();  
        // invoco metodo run dell'oggetto t1  
};
```

...

# Creazione in-place

```
( new Thread(  
    new Runnable() {  
        ...  
        public void run() {  
            ...  
        }  
    }  
).start();
```

# Sleep

## **static sleep(milliseconds)**

Pausa per x millisecondi

Altri thread possono andare in esecuzione

# Interruzione

## **interrupt()**

- Viene messo a true il flag di interruzione
- Appena il thread entra in stato di blocked riceve una InterruptedException:
- Ci si affida alla correttezza del programma del thread: potrebbe ignorare tale eccezione!

## **boolean isAlive()**

- True se il thread non ha ancora terminato l'esecuzione

```
// main
thr1.start();
thr1.interrupt();
status = thr1.isAlive() ? "is alive" : " is not alive";
```

```
// thread thr1
try {
    System.out.println("thread 1");
    Thread.sleep(3000L);
} catch (InterruptedException iex) { ... };
```



# Interruzione attraverso flag

Un altro modo per interrompere un thread:

- Il thread ha un flag `timetostop` che controlla durante l'esecuzione
- Il thread fornisce un metodo `mystop` per mettere `timetostop` a `true`
- Il main o un'altro thread usa il metodo per fermare il thread

```
// Classe MyStop
```

```
public class MyStop extends NamedObject implements  
    Runnable {
```

```
    private volatile boolean timetostop=false;
```

```
    public MyStop(String n){ super(n); }
```

```
    public void mystop() {  
        timetostop=true;  
    }
```

```
    ...
```

```
// Classe MyStop
```

```
public class MyStop extends NamedObject implements  
    Runnable {
```

```
    private volatile boolean timetostop=false;
```

```
    public MyStop(String n){ super(n); }
```

```
    public void mystop() {  
        timetostop=true;  
    }
```

```
    ...
```

```
// Classe MyStop
```

```
public class MyStop extends NamedObject implements  
    Runnable {
```

```
    private volatile boolean timetostop=false;
```

```
    public MyStop(String name, int n); }
```

```
    public void my  
        timetostop=  
    }
```

```
    ...
```

Dal Java Memory Model:

The value of this variable will never be cached thread-locally

Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

// Classe MyStop

```
public class MyStop extends NamedObject implements  
    Runnable {
```

```
    private volatile boolean timetostop=false;
```

```
    public MyStop(String name, int n) {
```

```
        public void my  
            timetostop=  
        }  
    }
```

```
    ...
```

Tutte le letture e scritture di un campo volatile devono essere fatte in memoria principale

Le operazioni su campi volatili devono essere eseguite esattamente nell'ordine richiesto dal thread

// Metodo run della classe MyStop

```
public void run() {  
    try {  
        while (!timetostop) {  
            System.out.println(this.name);  
            Thread.sleep(1000L);  
        }  
    } catch (InterruptedException iex) {}  
}
```

```
// main
```

```
thr1.start();
```

```
status = thr1.isAlive() ? "is alive" : " is not alive";
```

```
System.out.println(status);
```

```
...
```

```
System.out.println("stop thread 1");
```

```
t1.mystop();
```

```
...
```

```
status = thr1.isAlive() ? "is alive" : " is not alive";
```

```
System.out.println(status);
```

# Attesa terminazione

**t.join();**

- Il caller attende che il thread **t** abbia terminato l'esecuzione

...

t1.start();

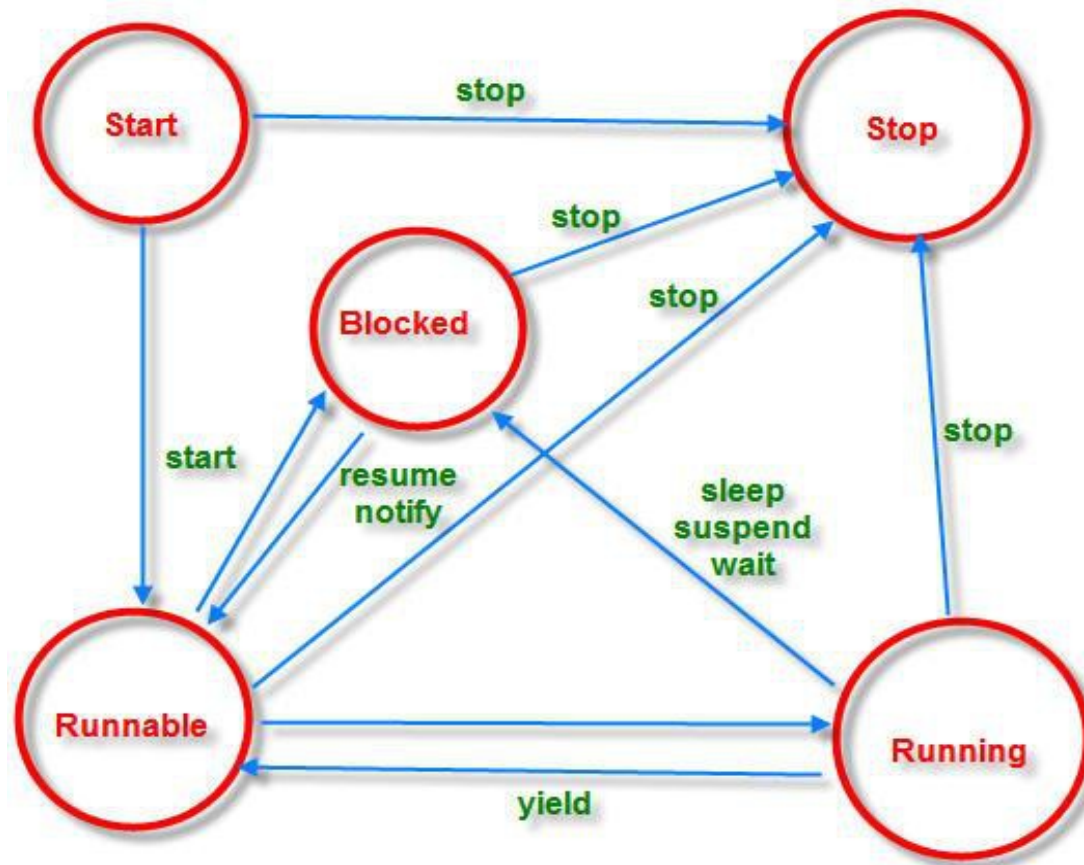
t2.start();

t1.join();

t2.join();



# Riepilogo: Ciclo di vita



# Stati di un thread

- **Start** state
  - Thread appena creato
  - Quando viene chiamato **start()** entra nel Ready state
- **Runnable** state
  - Pronto per essere eseguito
- **Running** state
  - Il thread è effettivamente in esecuzione
  - Quando **run()** termina entra in stop state
- **Stop** state
  - Il thread può essere rimosso dal sistema
  - Quando **run()** termina oppure per eccezione non gestita

# Stati di un thread

- **Blocked** state
  - Mentre è in running può entrare in questo stato
  - Ad es. in attesa per una richiesta I/O
- **Sleeping** state
  - Quando viene chiamato il metodo **sleep()**
  - Rientra in stato runnable quando lo sleep time è passato
- **Waiting** state
  - lo vedremo dopo ...

# Sincronizzazione

# Scheduler

- L'esecuzione dei thread è organizzata da uno scheduler che decide quale thread eseguire e per quanto tempo mantenerlo in esecuzione
- La strategia di schedulazione è basata su code multiple di priorità con round-robin su ogni coda
- Per cambiare la priorità (MIN priority=1)

**void setPriority(int Priority)**

# Accesso concorrente

- Due o più thread possono accedere (leggere e modificare) lo stesso oggetto
- In caso di context switch da un thread ad un'altro si potrebbero verificare delle race condition sulla parte di oggetto condivisa

```
class Prenotazione {  
    private int posti=20;  
    public int get() { return posti; }  
  
    public boolean prenota(int richiesta) {  
        if (posti >= richiesta) {  
            posti -= richiesta;  
            return true;  
        }  
        return false;  
    }  
}
```

```
public class Cliente extends Thread {
```

```
    private Prenotazione p;
```

```
    private int richiesta;
```

```
    public Cliente(Prenotazione X,int r){
```

```
        this.p=X;
```

```
        this.richiesta=r;
```

```
    }
```

```
    public void run(){
```

```
        if (p.prenota(this.richiesta))
```

```
            System.out.println("ok: "+richiesta);
```

```
    }
```

```
    ...
```

```
}
```



```
public static void main(String[] args) throws  
    InterruptedException {  
        Prenotazione P=new Prenotazione(); // oggetto condiviso  
        Cliente t1=new Cliente(P,3);  
        Cliente t2=new Cliente(P,10);  
        Cliente t3=new Cliente(P,5);  
        Cliente t4=new Cliente(P,2);  
        Cliente t5=new Cliente(P,4);  
        Cliente t6=new Cliente(P,2);  
  
        t1.start();  
        t2.start();  
        t3.start();  
        t4.start();  
        t5.start();  
        t6.start();  
        System.out.println("Posti disponibili: "+P.get());  
    }
```

# Race condition

- Diversi thread fanno riferimenti alla stessa istanza di Prenotazione aggiornando il numero di posti disponibili
- Il test e l'assegnamento al campo posti non viene eseguito in modo atomico (cioe' tra il test e l'assegnamento, un thread potrebbe essere temporaneamente sospeso)
- Il numero di posti assegnato potrebbe essere maggiore di quello realmente disponibile!

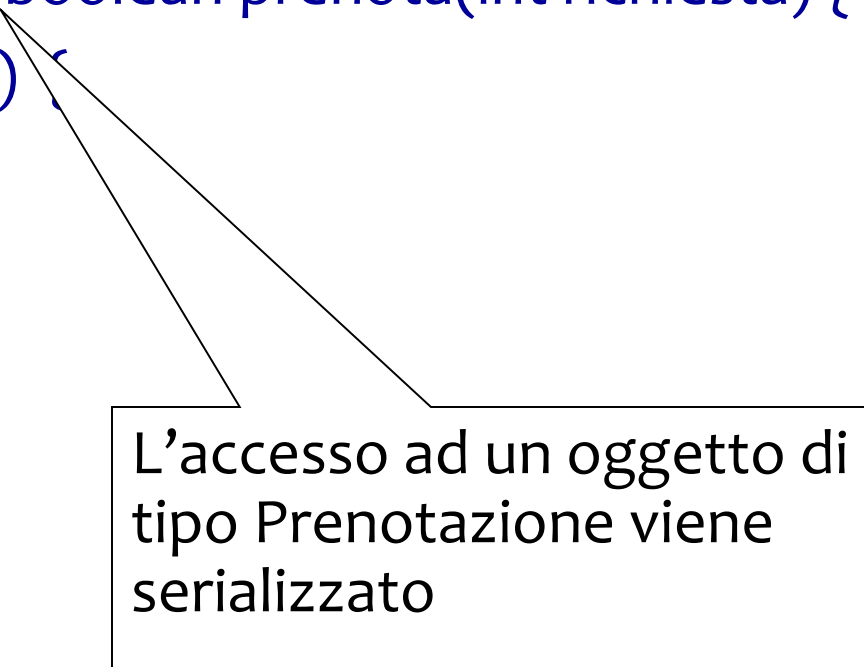
```
class Prenotazione {  
    private int posti=20;  
    public int get() { return posti; }  
  
    public boolean prenota(int richiesta) {  
        if (posti >= richiesta) {  
            try { // simuliamo un delay  
                Thread.sleep ((int) Math.random() * 2);  
            } catch (InterruptedException e) {}  
            posti -= richiesta;  
            return true;  
        }  
        return false;  
    }  
}
```

# Synchronized Methods

- I metodi di un oggetto si possono dichiarare **synchronized**
- Solo un thread alla volta può eseguire un tale metodo su uno stesso oggetto
- Se un thread sta eseguendo un metodo **synchronized** altri thread non possono eseguire altri metodi **synchronized** su uno stesso oggetto
- Lo stesso thread può eseguire altri metodi **synchronized** sullo stesso oggetto

```
class PrenotazioneSicura {  
    private int posti=20;  
    public int get() { return posti; }  
  
    public synchronized boolean prenota(int richiesta) {  
        if (posti >= richiesta) {  
            posti -= richiesta;  
            return true;  
        }  
        return false;  
    }  
}
```

```
class PrenotazioneSicura {  
    private int posti=20;  
    public int get() { return posti; }  
  
    public synchronized boolean prenota(int richiesta) {  
        if (posti >= richiesta) {  
            posti -= richiesta;  
            return true;  
        }  
        return false;  
    }  
}
```



L'accesso ad un oggetto di tipo Prenotazione viene serializzato

# Monitor

Quando un thread deve eseguire un metodo synchronized su un oggetto si blocca finche' non riesce ad ottenere il lock sull'oggetto

Quando lo ottiene puo' eseguire il metodo

Gli altri thread rimarranno bloccati finche' il lock non viene rilasciato

Quando il thread esce dal metodo synchronized rilascia automaticamente il lock

A quel punto gli altri thread proveranno ad acquisire il lock  
Solo uno ci riuscirà e gli altri torneranno in attesa

# Blocchi Synchronized

- Singoli blocchi di codice possono essere dichiarati synchronized su un certo oggetto
- Un solo thread alla volta puo' eseguire un tale blocco su uno stesso oggetto
- Permette di minimizzare le parti di codice da serializzare ed aumentare il parallelismo (es. se un solo field e' condiviso tra piu' thread)
- Si perde la possibilita' di documentare a livello di classe la sincronizzazione sui dati



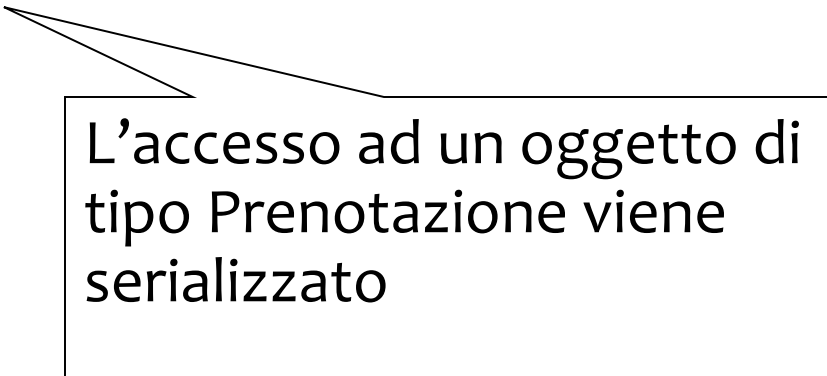
# Metodi e blocchi sincronizzati

```
public synchronized int read() {  
    return field;  
}
```

```
public int read() {  
    synchronized (this) {  
        return field;  
    }  
}
```

```
class PrenotazioneSicura {  
    private int posti=20;  
    ...  
  
    public bool prenota (int richiesta) {  
        synchronized (this) {  
            if (posti >= richiesta) {  
                posti -= richiesta;  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

```
class PrenotazioneSicura {  
    private int posti=20;  
    ...  
  
    public bool prenota (int richiesta) {  
        synchronized (this) {  
            if (posti >= richiesta) {  
                posti -= richiesta;  
                return true;  
            }  
        }  
        return false;  
    }  
}
```



L'accesso ad un oggetto di tipo Prenotazione viene serializzato

# Reentrant synchronization

```
public class MyClass {  
    public synchronized int foo (int x) {  
        if (x>0) return x+foo(x-1);  
        else return 0;  
    }  
  
    public static void main(String arg[]) {  
        MyClass obj=new MyClass();  
        System.out.println("risultato: "+obj.foo(4));  
    }  
}
```

# Reentrant synchronization

```
public class MyClass {  
    public synchronized int foo (int x) {  
        if (x>0) return x+foo(x-1);  
        else return 0;  
    }  
  
    public static void main (String args[]) {  
        MyClass obj = new MyClass();  
        System.out.println(obj.foo(10));  
    }  
}
```

Una volta ottenuto il lock su un oggetto invochiamo un altro (o lo stesso) metodo sincronizzato

In questo caso il thread chiamante non si blocca!

# Altri usi di synchronized

- Se non vogliamo cambiare i metodi della classe Prenotazione:
  - Un cliente puo' sincronizzarsi sull'oggetto di tipo Prenotazione condiviso

```
class Cliente extends Thread {  
    Prenotazione obj;  
    public void run(){  
        ...  
        synchronized(obj) { res=obj.prenota(N)); }  
    }  
}
```

# Ereditarieta'

- La specifica `synchronized` non fa parte vera e propria della segnatura di un metodo
- Una classe derivata puo' ridefinire un metodo `synchronized` come non `synchronized` e viceversa

# Ereditarietà

- La specifica `synchronized` non è parte vera e propria della segnatura di un metodo

- Una classe `Base` può avere un metodo `synchronized` e una classe `Derivata` che eredita da `Base` può sovrascriverlo con un metodo `synchronized` o non `synchronized`.

```
class Base {  
    public void synchronized m1() { /* ... */ }  
}  
  
class Derivata extends Base {  
    public void m1() {  
        // not synchronized  
        super.m1(); // synchronized  
        // not synchronized  
    }  
}
```

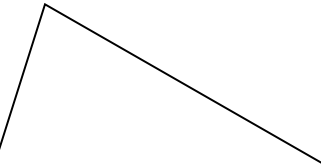


# Ereditarieta'

Possiamo ridefinire PrenotazioneSicura rendendo  
sincronizzato il metodo della superclasse

# Ereditarietà

Possiamo ridefinire PrenotazioneSicura rendendo sincronizzato il metodo della superclasse



```
class PrenotazioneSicura {  
    ....  
    public synchronized bool prenota (int X) {  
        return super.prenota(X);  
    }  
}
```

# Static Data

- I metodi e blocchi synchronized non assicurano l'accesso mutuamente esclusivo ai dati statici
- I dati statici sono condivisi da tutti gli oggetti della stessa classe
- In Java ad ogni classe e' associato un oggetto di classe **Class**
- Per accedere in modo sincronizzato ai dati statici si deve ottenere il lock su questo oggetto Class
  - si puo' dichiarare un metodo statico come synchronized
  - si puo' dichiarare un blocco come synchronized sull'oggetto

# Produttore e consumatore

- Due processi comunicano tramite un oggetto condiviso
- Vogliamo:
  - Non leggere cose già lette
  - Non sovrascrivere cose non ancora lette
- Usare solo parti di codice `synchronized` non `e sufficiente
  - Produttore acquisisce il lock
  - Scrive una nuova informazione
  - Produttore riacquisisce il lock nuovamente, prima del consumatore
  - Produttore sovrascrive un'informazione non ancora recuperata dal consumatore

```
class Produttore extends Thread {  
    CellaCondivisaErrata cella;  
  
    public Produttore (CellaCondivisa cella)  
    { this.cella = cella;}  
  
    public void run() {  
        for (int i = 1; i <= 10; ++i) {  
            synchronized (cella) {  
                ++(cella.valore);  
                System.out.println ("Prodotto: " + i);  
            }  
        }  
    }  
}
```

```
class Consumatore extends Thread {  
    CellaCondivisa cella;  
    public Consumatore (CellaCondivisa cella) {  
        this.cella = cella;  
    }  
  
    public void run() {  
        int valore letto;  
        for (int i = 0; i < 10; ++i) {  
            synchronized (cella) {  
                valore letto = cella.valore;  
                System.out.println ("Consumato: " + valore letto);  
            }  
        }  
    }  
}
```

```
class Consumatore extends Thread {  
    CellaCondivisa cella;  
    public Consumatore (CellaCondivisa cella) {  
        this.cella = cella;  
    }  
}
```

```
    public void run() {  
        int valore letto;  
        for (int i = 0; i < ...  
            synchronized (c  
                valore letto = c  
                System.out.pri  
            }  
        }  
    }  
}
```

## Problema

L'accesso e' sincronizzato ma  
il produttore potrebbe  
sovrascrivere la cella prima  
che il consumatore legga!

# Wait e Notify

Un thread pu`o chiamare **wait()** su un oggetto sul quale ha il lock:

- Il lock viene rilasciato
- Il thread va in stato di waiting

Altri thread possono ottenere tale lock

Effettuano le opportune operazioni e invocano su un oggetto:

- **notify()** per risvegliare un singolo thread in attesa su quell'oggetto
- **notifyAll()** per risvegliare tutti i thread in attesa su quell'oggetto

I thread risvegliati devono comunque riacquisire il lock

I notify non sono cumulativi



# Wait-set

- Ad ogni oggetto Java e' associato un wait-set: l'insieme dei thread che sono in attesa per l'oggetto
- Un thread viene inserito nel wait-set quando esegue la wait
- I thread sono rimossi dal wait-set attraverso le notifiche:
  - notify ne rimuove solo uno
  - notifyAll li rimuove tutti

# Eccezioni

- Wait:
  - `IllegalMonitorStateException` - if the current thread is not the owner of the object's monitor.
  - `InterruptedException` - if another thread has interrupted the current thread.
- Notify:
  - `IllegalMonitorStateException` - if the current thread is not the owner of the object's monitor.

```
class Produttore extends Thread {  
    public void run() {  
        for (int i = 1; i <= 10; ++i) {  
            synchronized (cella) {  
                while (!cella.scrivibile) {  
                    try { cella.wait();  
                    } catch (InterruptedException e) { return;}  
                }  
                ++(cella.valore);  
                cella.scrivibile = false; // cede il turno  
                cella.notify(); // risveglia il consumatore  
            }  
        }  
    }  
}
```

```

class Produttore extends Thread {
    public void run() {
        for (int i = 1; i <= 10; ++i) {
            synchronized (cella) {
                while (!cella.scrivibile) {
                    try { cella.wait();
                } catch (InterruptedException e) { return;}
            }
            ++(cella.valore);
            cella.scrivibile = false; // cede
            cella.notify(); // risveglia il
        }
    }
}

```

Funziona anche se  
si risveglia senza  
notify

```
class Consumatore extends Thread {  
    public void run() {  
        int valore letto;  
        for (int i = 0; i < 10; ++i) {  
            synchronized (cella) {  
                while (cella.scrivibile) {  
                    try { cella.wait();  
                    } catch (InterruptedException e) { return; }  
                }  
                valore letto = cella.valore;  
                cella.scrivibile = true; // cede il turno  
                cella.notify(); // notifica il produttore  
            }  
        }  
    }  
}
```

# notify vs notifyAll

- Un notify effettuato su un oggetto su cui nessun thread e' in wait viene perso (non è un semaforo)
- Se ci possono essere piu' thread in attesa usare notifyAll:
  - Risveglia tutti i thread in attesa
  - Tutti si rimettono in coda per riacquisire il lock
  - Ma solo uno alla volta riprenderà il lock
- Prima che un thread in attesa riprenda l'esecuzione il thread che ha notificato deve rilasciare il monitor (uscire dal blocco synchronized)