

# Programmazione concorrente in Java

da materiale di  
Carlo Ghezzi e Alfredo Motta

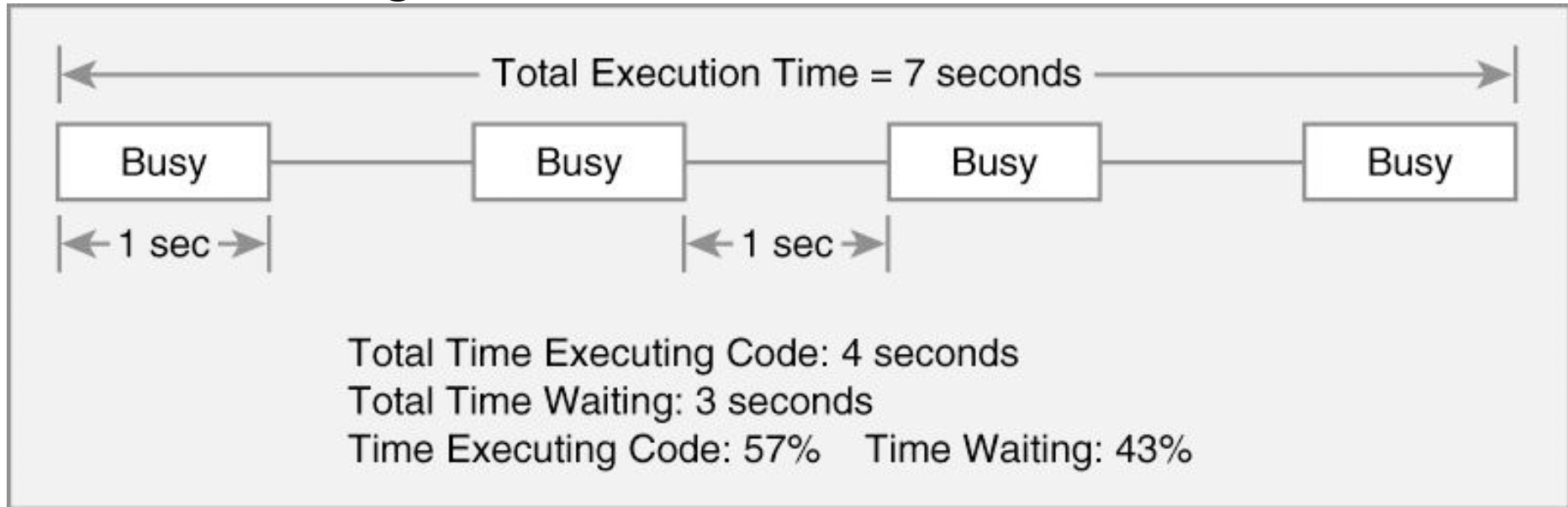
# Parallelismo = “multitasking”

- Possiamo scrivere un programma in cui diverse attività (task) evolvono in parallelo da un punto di vista fisico o logico
- Massimo **parallelismo fisico**
  - Ogni attività parallela ha a disposizione un processore fisico
- Altrimenti vengono eseguite da **processori condivisi**
  - Secondo modalità decise da uno **scheduler**, in generale non controllabile dal programmatore

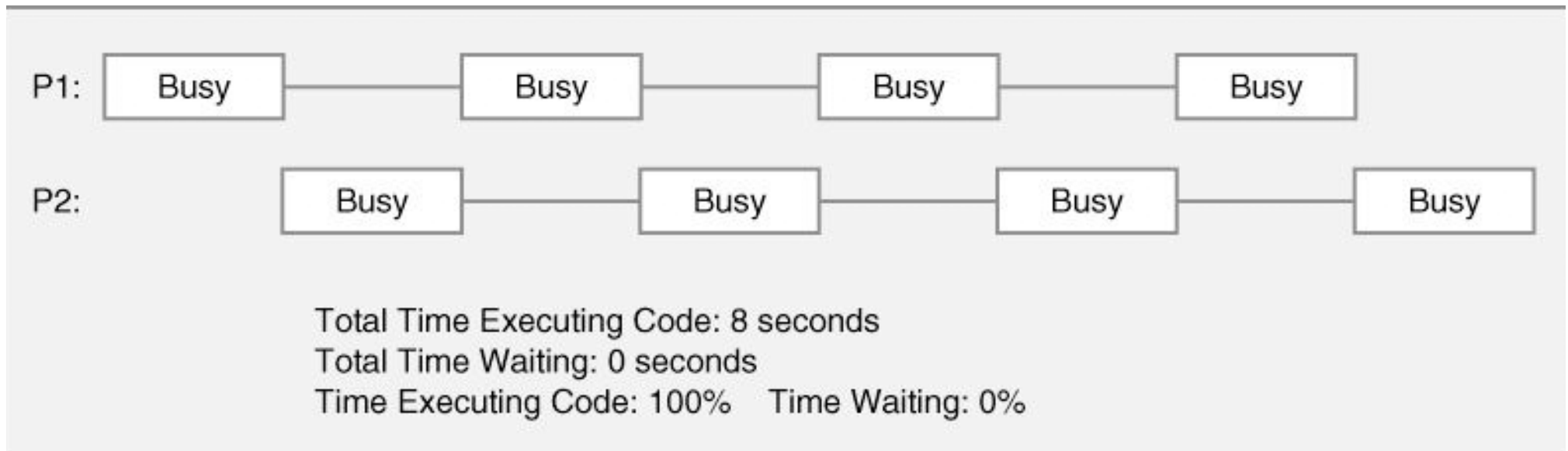
# Multitasking su singolo processore

- Approccio
  - Processore esegue un task
  - Passa velocemente a un altro
  - Sotto il governo dello scheduler
- Il processore **sembra** lavorare sui diversi task concorrentemente
- Passaggio da un task all'altro nei momenti di inattività o per esaurimento della finestra temporale (“time sharing”)

## Caso 1: Task singolo



## Caso 2: Due task



# Multitasking a livello di processi

- **Processo**

- Programma eseguibile caricato in memoria
- Ha un suo spazio di indirizzi (variabili e strutture dati in memoria)
- Ogni processo esegue un diverso programma
- I processi comunicano via SO, file, rete
- Può contenere più **thread**

# Multitasking a livello di thread

- **Thread** è un'attività logica sequenziale
- Un thread condivide lo spazio di indirizzi con gli altri thread del processo e comunica via **variabili condivise**
- Ha un suo contesto di esecuzione (program counter, variabili locali)
- Si parla spesso di **processo light-weight**

# Thread in Java - metodo 1

The thread object is defined by extending the `Thread` class

You must override `run()`, to make the thread do your bidding

Thread object is instantiated as usual through a `new()`

To run the thread you must invoke the `start()` method on it

```
public class MyThread extends Thread {
    private String message;
    public MyThread(String m) {message = m;}
    public void run() {
        for(int r=0; r<20; r++)
            System.out.println(message);
    }
}

public class ProvaThread {
    public static void main(String[] args) {
        MyThread t1,t2;
        t1=new MyThread("primo thread");
        t2=new MyThread("secondo thread");
        t1.start();
        t2.start();
    }
}
```

# Classe Thread

start	avvia il thread (eseguendo il metodo run)
join	chiamato su un thread specifico e ha lo scopo di mettere in attesa il thread attualmente in esecuzione fino a quando il thread su cui è stato invocato il metodo join() non termini
isAlive	controlla se il thread è vivo (in esecuzione, in attesa o bloccato)
sleep(int ms)	sospende l'esecuzione del thread
yield	mette temporaneamente in pausa il thread corrente e consente ad altri thread in stato Runnable (qualora ve ne siano) di avere una chance per essere eseguiti



```
public class A extends Thread {
    public void run(){
        for (int i=0; i<=5; i++) System.out.println("From Thread A: i= "+i);
        System.out.println("Exit from A");
    }
}

public class B extends Thread {
    public void run(){
        for (int j=0; j<=5; j++) System.out.println("From Thread B: j= "+j);
        System.out.println("Exit from B");
    }
}

public class C extends Thread {
    public void run(){
        for (int k=0; k<=5; k++) System.out.println("From Thread C: k= "+k);
        System.out.println("Exit from C");
    }
}

public class ThreadTest {
    public static void main(String[] args) {
        new A().start();
        new B().start();
        new C().start();
    }
}
```

# Output possibili

## Caso 1

```
From Thread A: i= 0
From Thread A: i= 1
From Thread A: i= 2
From Thread A: i= 3
From Thread A: i= 4
From Thread A: i= 5
Exit from A
From Thread B: j= 0
From Thread B: j= 1
From Thread B: j= 2
From Thread B: j= 3
From Thread B: j= 4
From Thread B: j= 5
Exit from B
From Thread C: k= 0
From Thread C: k= 1
From Thread C: k= 2
From Thread C: k= 3
From Thread C: k= 4
From Thread C: k= 5
Exit from C
```

## Caso 2

```
From Thread A: i= 0
From Thread A: i= 1
From Thread A: i= 2
From Thread A: i= 3
From Thread A: i= 4
From Thread A: i= 5
Exit from A
From Thread C: k= 0
From Thread C: k= 1
From Thread C: k= 2
From Thread C: k= 3
From Thread C: k= 4
From Thread C: k= 5
From Thread B: j= 0
From Thread B: j= 1
From Thread B: j= 2
From Thread B: j= 3
From Thread B: j= 4
From Thread B: j= 5
Exit from B
Exit from C
```

# Thread in Java – metodo 2

- La classe Thread in realtà implementa un'interfaccia chiamata Runnable
- L'interfaccia Runnable definisce un solo metodo run che contiene il codice del thread
- Su ciò si basa un modo alternativo

# Thread in Java—metodo 2

- Metodo più generale si usa se si deve ereditare da qualche classe

Your class must implement `Runnable` interfaces

`run()` must be overridden

To produce a thread from a `Runnable` object, you must create a separate `Thread` object

`start()` method is invoked to execute the thread

```
public class MyThread implements Runnable {
    private String message;
    public MyThread(String m) {message = m;}
    public void run() {
        for(int r=0; r<20; r++)
            System.out.println(message);
    }
}

public class ProvaThread {
    public static void main(String[] args) {
        Thread t1, t2;
        MyThread r1, r2;
        r1 = new MyThread("primo thread");
        r2 = new MyThread("secondo thread");
        t1 = new Thread(r1);
        t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

# Dati condivisi

- Può essere necessario imporre che certe sequenze di operazioni che accedono a dati condivisi vengano eseguite dai task in mutua esclusione

```
class ContoCorrente {  
    private float saldo;  
    public ContoCorrente(float saldoIniz){  
        saldo=saldoInizi;  
    }  
    public void deposito(float soldi){  
        saldo += soldi;}  
    public void prelievo (float soldi){  
        saldo -=soldi;}  
}
```

...

# Interferenza

- Fenomeno causato dall'interleaving di operazioni di due o più thread
  - Lettura di y e x
  - Esecuzione espressione
  - Scrittura in x
- L'esecuzione concorrente di  $x+=y$  e  $x-=y$  può avere uno dei seguenti effetti
  - Incrementare x di y
  - Lasciare x immutata
  - Decrementare x di y

# Sequenze “atomiche”

- Generalizzazione del problema
  - A volte si vuole che certe sequenze di istruzioni vengano eseguite in isolamento, senza interleaving con istruzioni di altre sequenze parallele che altri thread potrebbero eseguire
  - Si parla di “**sequenze atomiche**”

# Altro problema di concorrenza

- A volte, oltre a voler l'atomicità di certe sequenze, si vogliono imporre certi ordinamenti nell'esecuzione di operazioni
- Per esempio, che l'operazione A eseguita da un thread venga eseguita prima dell'operazione B di un altro thread
- Di solito ciò deriva dal fatto di voler garantire certe proprietà di **consistenza nei dati**



# Come rendere i metodi "atomici"

- La parola chiave **synchronized** applicata a metodi o blocchi di codice li rende atomici
  - Il thread che li esegue non potrà essere interrotto!

```
class ContoCorrente {  
    private float saldo;  
    public ContoCorrente(float saldoIniz){  
        saldo = saldoIniz;}  
    public synchronized void deposito(float soldi){  
        saldo += soldi;}  
    public synchronized void prelievo(float soldi){  
        saldo -= soldi;}  
    ...  
}
```

# Esempio

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
    public synchronized void decrement() {  
        c--;  
    }  
    public synchronized int value() {  
        return c;  
    }  
}
```

```
public class TaskA extends Thread {  
    private SynchronizedCounter counter;  
  
    public TaskA(SynchronizedCounter c) {  
        counter = c;  
    }  
  
    public void run(){  
        counter.increment();  
        System.out.println(counter.value());  
    }  
}
```

```
public class TaskB extends Thread {  
    private SynchronizedCounter counter;  
  
    public TaskB(SynchronizedCounter c) {  
        counter = c;  
    }  
  
    public void run(){  
        counter.decrement();  
        System.out.println(counter.value());  
    }  
}
```

```
public class ThreadTest {  
    public static void main(String[] args) {  
        SynchronizedCounter c = new SynchronizedCounter();  
  
        TaskA ta = new TaskA(c);  
        TaskB tb = new TaskB(c);  
        ta.start();  
        tb.start();  
    }  
}
```

# Metodi synchronized

- Java associa un **intrinsic lock** a ciascun oggetto
  - I lock operano a livello di thread
- Quando il metodo synchronized viene invocato
  - Se nessun metodo synchronized è in esecuzione, l'oggetto viene bloccato (locked) e quindi il metodo viene eseguito
  - Se l'oggetto è bloccato, il task chiamante viene **sospeso** fino a quando il task bloccante libera il lock
  - ...quindi, al massimo un singolo thread può trovarsi ad eseguire istruzioni all'interno di uno stesso metodo synchronized

# Commenti sul lock

- Diverse invocazioni di metodi synchronized sullo stesso oggetto **non** sono soggette a interleaving
- I costruttori non possono essere synchronized
  - Solo il thread che crea l'oggetto deve avere accesso ad esso mentre viene creato
- Eventuali dati final possono essere letti con metodi **non** synchronized
  - I dati sono comunque in sola lettura e non possono essere modificati

# Ulteriori commenti sul lock

- L'intrinsic lock viene acquisito automaticamente all'invocazione del metodo `synchronized` e rilasciato al ritorno
  - ...sia normale che eccezionale dovuto ad una `uncaught exception`
- Se il metodo `synchronized` fosse static
  - Il thread acquisisce l'intrinsic lock per il **.class** object associato alla classe
  - Pertanto l'accesso ai campi static è controllato da un lock speciale, diverso da quelli associati alle istanze della classe

# Synchronized statements

- Devono specificare **l'oggetto** a cui applicare il lock

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

- Si rilascia il lock all'oggetto prima di invocare un metodo che potrebbe a sua volta richiedere di attendere il rilascio di un lock



# Controllo “fine” della concorrenza

- Esempio: classe con due campi che non vengono modificati mai insieme

```
public class TestBlock {  
    private long c1 = 0;  
    private long c2 = 0;  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        synchronized(lock1) {c1++;}  
    }  
  
    public void inc2() {  
        synchronized(lock2) {c2++;}  
    }  
}
```

# Alcune regole pratiche

- Usare lock per la modifica degli attributi dell'oggetto
  - Per essere certi di avere uno stato consistente
- Usare lock per l'accesso a campi dell'oggetto probabilmente modificati
  - Per evitare di leggere valori “vecchi”
- Non c'è bisogno di lock per accedere alle parti “stateless” di un metodo, o ad attributi final

# Liveness

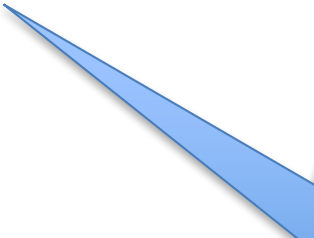
- È una proprietà molto importante in pratica
- Significa che un'applicazione concorrente viene eseguita entro accettabili limiti di tempo
- Le situazioni da evitare attraverso un'attenta progettazione sono
  - **Deadlock**
  - **Starvation**
  - **Livelock**

# Deadlock

- Due o più thread sono bloccati per sempre, in attesa l'uno dell'altro
  - Esempio: Anna e Giacomo sono amici e credono nel galateo, che dice che se una persona si inchina a un amico, deve restare inchinata fino a che l'amico restituisce l'inchino
- E se si inchinano tutti e due allo stesso tempo?

```
public class Friend {  
    private final String name;  
  
    public Friend(String name) {  
        this.name = name;  
    }  
    public String getName() {  
        return this.name;  
    }  
  
    public synchronized void bow(Friend bower) {  
        System.out.format("%s: %s" + " has bowed to me!\n",  
            this.name, bower.getName());  
        bower.bowBack(this);  
    }  
    public synchronized void bowBack(Friend bower) {  
        System.out.format("%s: %s" + " has bowed back to me!\n",  
            this.name, bower.getName());  
    }  
}
```

```
public class ThreadTest {  
    public static void main(String[] args) {  
        final Friend anna = new Friend("Anna");  
        final Friend giacomo = new Friend("Giacomo");  
  
        new Thread(new Runnable() {  
            public void run() {  
                anna.bow(giacomo);  
            }  
        }).start();  
  
        new Thread(new Runnable() {  
            public void run() {  
                giacomo.bow(anna);  
            }  
        }).start();  
    }  
}
```



Classi **anonime**: approccio comodo per creare nuovi thread con un semplice comportamento

# Starvation

- Situazione in cui un thread ha difficoltà ad accedere a una risorsa condivisa e quindi ha difficoltà a procedere
- Esempio:
  - Thread “greedy” che frequentemente invocano metodi lunghi ritardano costantemente altri thread
  - Uno scheduler che usa priorità cede sempre precedenza ai task greedy

# Livelock


- Errore di progetto che genera una sequenza ciclica di operazioni inutili ai fini dell'effettivo avanzamento della computazione
- Esempio:
  - La sequenza infinita di “vada prima lei”
- Diverso dal deadlock: la computazione **non** è bloccata, qualcosa viene fatto ma mai niente di utile



# Guarded blocks

- Come evitare il prelievo se il conto va in rosso?

```
public class ContoCorrente {  
    private float saldo;  
  
    public synchronized void prelievo (float soldi){  
        while (saldo-soldi<0) wait();  
        saldo -= soldi;  
    }  
}
```



Rilascia il lock e  
sospende il thread!

# Come risvegliare un task in wait?

```
public class ContoCorrente {  
    private float saldo;  
  
    public ContoCorrente (float saldoI) {  
        saldo = saldoI;  
    }
```

```
    synchronized public void deposito (float soldi) {  
        saldo += soldi;  
        notify();  
    }
```

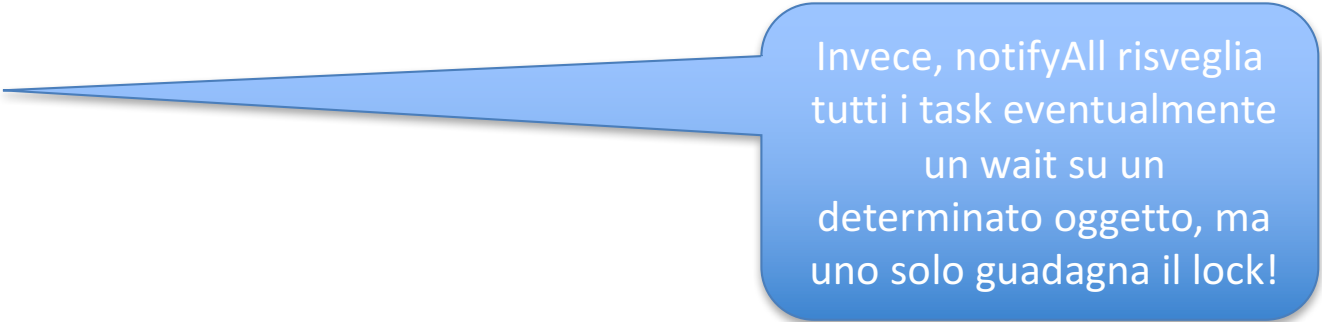
Risveglia un task (scelto a caso) in sospenso da wait su questo oggetto, se esiste

```
    public synchronized void prelievo (float soldi) {  
        while (saldo-soldi < 0) wait();  
        saldo -= soldi;  
    }  
}
```

Potrebbe non essere sufficiente... Il thread potrebbe ripartire anche senza essere notificato da nessun'altro thread...

# Esempio: una coda FIFO condivisa

- Operazione di inserimento di elemento
  - Sospende task se coda piena
    - `while (codaPiena()) wait();`
  - Al termine
    - `notify();`
- Operazione di estrazione di elemento
  - Sospende task se coda vuota
    - `while (codaVuota()) wait();`
  - Al termine
    - `notify();`



Invece, `notifyAll` risveglia tutti i task eventualmente un `wait` su un determinato oggetto, ma uno solo guadagna il lock!

```

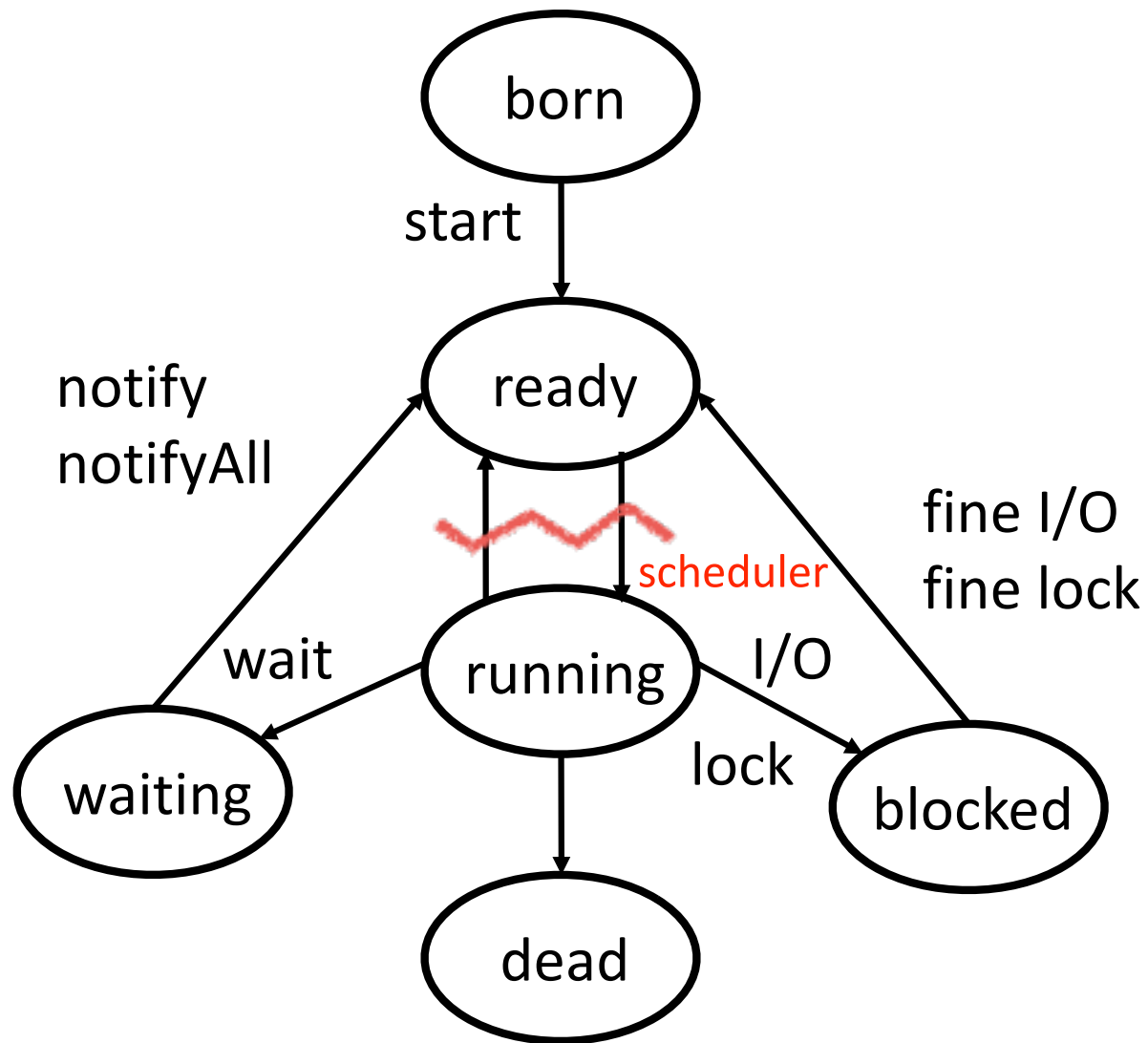
public class FIFO {
    private boolean empty;
    private String message;

    public synchronized String take() {
        // Wait until message is available
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status
        empty = true;
        // Notify producer that status has changed
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        // Wait until message has been retrieved
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status
        empty = false;
        // Store message
        this.message = message;
        // Notify consumer that status has changed
        notifyAll();
    }
}

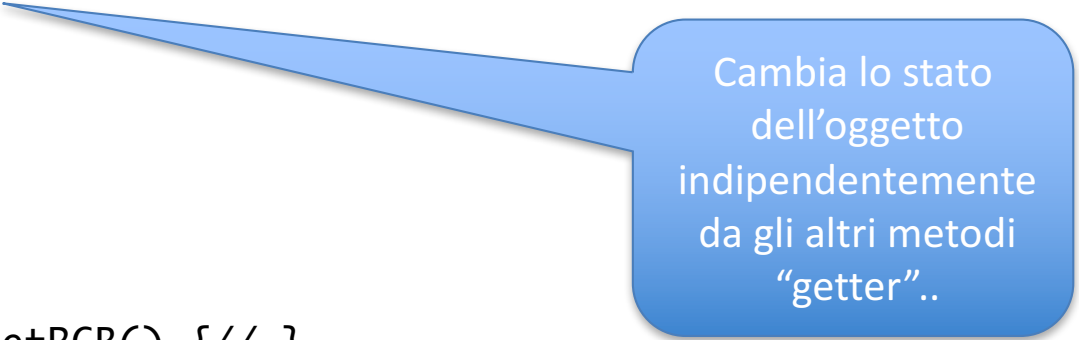
```

# Ciclo di vita di un thread



# Problemi con oggetti mutabili

```
public class SynchronizedRGB {  
    // Values must be between 0 and 255  
    private int red;  
    private int green;  
    private int blue;  
    private String name;  
  
    private void check(int red, int green, int blue) {//...}  
    public SynchronizedRGB(int red, int green, int blue, String name) {//...}  
  
    public void set(int red, int green, int blue, String name) {  
        check(red, green, blue);  
        synchronized (this) {  
            this.red = red;  
            this.green = green;  
            this.blue = blue;  
            this.name = name;  
        }  
    }  
  
    public synchronized int getRGB() {//...}  
    public synchronized String getName () {//...}  
    public synchronized void invert () {//...}  
}
```



Cambia lo stato  
dell'oggetto  
indipendentemente  
da gli altri metodi  
"getter"..

# Problema

```
SynchronizedRGB color =  
    new SynchronizedRGB(0,0,0, "Pitch Black");  
...  
int myColorInt = color.getRGB();           //Statement 1  
String myColorName = color.getName();      //Statement 2
```

- Se un altro thread invoca set dopo Statement 1 ma prima di Statement 2, il valore di myColorInt non corrisponde al valore di myColorName
- Il problema sorge perché l'oggetto è mutabile

# Come creare oggetti immutabili

- Non fornire metodi che modificano lo stato
  - Non fornire metodi "setter"
- Definire tutti gli attributi di istanza (non static) final e private
- Non consentire alle sottoclassi di fare override dei metodi
  - Dichiarando la classe o i singoli metodi final
- Se gli attributi di istanza hanno riferimenti a oggetti mutabili, non consentire la loro modifica
- Non fare sharing di riferimenti a oggetti mutabili
- Non salvare riferimenti a oggetti esterni mutabili passati al costruttore, se necessario fare copie e salvare riferimenti alle copie
- Inoltre creare copie degli oggetti interni mutabili se necessario per evitare di restituire gli originali attraverso i metodi



# Esempio ImmutableRGB

- Tutti gli attributi sono già private; vengono ulteriormente qualificati final
- Il metodo invert viene adattato creando un nuovo oggetto invece di modificare l'oggetto corrente
- La classe viene qualificata final
- Un solo attributo fa riferimento a un oggetto, e l'oggetto è immutabile
  - Non è quindi necessario far nulla per salvaguardare lo stato di eventuali oggetti mutabili contenuti

# Soluzione ImmutableRGB

```
final public class ImmutableRGB {  
  
    // Values must be between 0 and 255.  
    final private int red;  
    final private int green;  
    final private int blue;  
    final private String name;  
  
    private void check(int red, int green, int blue) {  
        if (red < 0 || red > 255  
            || green < 0 || green > 255  
            || blue < 0 || blue > 255) {  
            throw new IllegalArgumentException();  
        }  
    }  
  
    public ImmutableRGB(int red, int green, int blue, String name) {  
        check(red, green, blue);  
        this.red = red;  
        this.green = green;  
        this.blue = blue;  
        this.name = name;  
    }  
    public ImmutableRGB invert() { //... }  
}
```