`Synchronized` (ancora)

- Riscriviamo l'esempio di prima.
  - Usiamo una struttura modulare, con una classe **`Notificatore`** che ha opportuni metodi.
  - La classe ha due campi privati, la lista **`bufText`** e un suo thread. Dato che implementa **`Runnable`** potrà costruire i thread che usano il suo **`run()`**.

```
import java.util.*;
public class Notificatore implements Runnable
{ private LinkedList bufText;
  private Thread th;
  // metodi (alla prossima slide)
}
```

Synchronized (ancora)

```
public void addMesg(String messaggio)
{ synchronized(this) { bufText.add(messaggio); } }
public void visualizza()
{ synchronized(this)
  { if (bufText.size()>0)
    System.out.println(bufText.remove(0));
  }
}
public void run()
{ while (true)
  { try { visualizza(); Thread.sleep(1000); }
    catch (InterruptedException e) { ; } }
}
public void avvia()
{ th = new Thread(this);
  th.setDaemon(true); th.start();
}
```

entrambi prendono il monitor dell'istanza di **`Notificatore`** per non essere eseguiti insieme

`Synchronized` (ancora)

- Moltissimi metodi che usano **`synchronized`** lo fanno sull'istanza su cui sono invocati: in pratica fanno **`synchronized(this)`**
  - Nell'esempio di prima è vero per entrambi:

```
/* il metodo che aggiunge un messaggio in coda */
public void addMesg(String s)
{ synchronized(this) { bufText.add(messaggio); } }
```

```
/* il metodo che va invocato ogni secondo */
public void visualizza()
{ synchronized(this)
  { if (bufText.size()>0)
    System.out.println(bufText.remove(0));
  }
}
```

`Synchronized` (ancora)

- **`synchronized(this)`** è così comune che Java mette a disposizione la possibilità di dichiarare un intero metodo **`synchronized`**
  - L'esempio di prima diventa:

```
/* il metodo che aggiunge un messaggio in coda */
public synchronized void addMesg(String s)
{ synchronized(this) { bufText.add(messaggio); } }
```

```
/* il metodo che va invocato ogni secondo */
public synchronized void visualizza()
{ synchronized(this)
  { if (bufText.size()>0)
    System.out.println(bufText.remove(0));
  }
}
```

`Synchronized` (ancora)

- Quindi per un metodo **`synchronized`**

```
synchronized metodo(parametri)
{ //corpo del metodo }
```

va letto come

```
metodo(parametri)
{ synchronized(this) { //corpo del metodo } }
```

- Ma se il metodo è statico della classe `C`

```
synchronized static metodo(parametri)
{ //corpo del metodo }
```

va letto come

```
static metodo(parametri)
{ synchronized(C.class) { //corpo del metodo } }
```

Starvation

- Sincronizzare troppo è sconsigliato!
  - I blocchi sincronizzati sono più lenti, e causano attese in altri thread che aspettano il monitor.
- Si rischia la **starvation** (=morte per fame).
  - Un thread potrebbe attendere molto a lungo un monitor monopolizzato da un thread egoista.
- Addirittura potremmo sbagliarci e non uscire mai da un ciclo dentro **`synchronized`**.
  - Così il monitor non viene rilasciato mai, e i thread che lo attendono non vanno mai in esecuzione.

Deadlock

- Un altro possibile problema è il **deadlock** (=abbraccio mortale), una condizione in cui due o più thread non riescono a progredire e vengono tutti bloccati.
  - Si ha deadlock se il Thread 1 deve prendere un monitor in possesso del Thread 2, e viceversa.
- L'uso sapiente dei monitor evita questi problemi: bisogna assicurarsi che se diversi thread chiedono più di un monitor li chiedano tutti **nello stesso ordine**.

Perché non usare `stop()`

- Il metodo **`stop()`** è deprecato in quanto fa terminare il thread su cui è invocato, ma non rilascia i monitor che questo si era preso.
- Ciò può causare una condizione di deadlock.
- Invece di invocare questo metodo, conviene fare terminare in modo naturale il thread.
  - Ad esempio, circondando tutto con un ciclo e inserendo una variabile booleana di terminazione.
  - Questo farà uscire l'esecuzione da tutti i blocchi **`synchronized`** rilasciando ogni monitor.

volatile

- Certe operazioni che sembrano innocue possono essere pericolose senza sincronia.
  - **`n++`** , **`double a = 3.0`** : sono entrambe due operazioni (**`double`** = 2 parole di memoria)
  - Tutto dipende dall'implementazione della JVM
  - Per robustezza, si può dichiarare un dato **`volatile`**: garantisce che l'accesso ad esso sarà atomico (non suddiviso in più operazioni)
- Usare questo costrutto può essere più efficiente che non un blocco **`synchronized`**.

Produttore/consumatore

- Riprendiamo l'esempio di stampa messaggi.
  - Non è molto efficiente guardare ogni secondo se ci sono messaggi in **`bufText`**.
  - Potremmo invocare **`visualizza()`** solo se ce n'è bisogno, per esempio, perché è stato aggiunto un messaggio a **`bufText`** con **`addMesg()`**

```
public synchronized void addMesg(String s)
{ bufText.add(messaggio); }
```

```
public synchronized void visualizza()
{ if (bufText.size()>0)
     System.out.println(bufText.remove(0));
}
```

`wait()` e `notify()`

- A tale scopo esistono **`wait()`** e **`notify()`** che abbiamo visto nel diagramma per entrare o uscire dallo stato di **attesa**.
- Praticamente:
  - **`wait()`**: aspetta finché non c'è un evento rilevante
  - **`notify()`**: è capitato un evento rilevante
- In questo modo si può implementare uno schema che si chiama produttore/consumatore, in cui un thread produce ciò che serve all'altro.

Produttore/consumatore

- Modifichiamo così il tutto come segue:

```
public synchronized void addMesg(String s)
{ bufText.add(messaggio); notify(); }
```

```
public synchronized void visualizza()
{ if (bufText.size()==0) wait();
  if while (bufText.size()>0)
    System.out.println(bufText.remove(0));
}
```

da notare che potremmo essere svegliati dall'attesa anche per altri motivi oltre a **notify()**. Per questo è bene verificare che la lista sia davvero piena.

```
public void run()
{ while (true)
  { try { visualizza(); Thread.sleep(1000); }
    catch (InterruptedException e) { ; } }
}
```

Classi involucro

- Le classi involucro (wrapper) di Java sono associate ad ogni tipo primitivo. Fornisono operazioni di supporto che non sarebbero possibili in quanto i numeri non sono oggetti.
- Tipicamente hanno il nome del tipo primitivo con la maiuscola: **`Double`**, **`Integer`**, **`Boolean`** **`Character`**, **`Short`**, **`Long`**, **`Byte`**, **`Float`**
- Tutte le classi involucro hanno:
  - due costruttori: da tipo primitivo e da **`String`**
  - metodi **`toString`**, **`equals`** e **`–Value`** (→**`intValue`**)

Classi involucro

- Le classi involucro implementano **`Comparable`** (e hanno **`compareTo`**) ma non **`Serializable`** (non servirebbe, i numeri sono serializzabili)
- Possono essere convertire in numeri tramite **unboxing** (= estrarre dalla scatola) e **autoboxing** (= creare la scatola al volo)

```
Integer i = new Integer(123);
int j = i.intValue();
```

autoboxing

unboxing

Tipi enumerativi

- Può servire rappresentare insiemi enumerativi (giorni della settimana, stagioni, semi di carte,...)
- Un modo di farlo può essere come segue:

```
public static final int CARTA_FIORI  = 0;
public static final int CARTA_QUADRI = 1;
public static final int CARTA_CUORI  = 2;
public static final int CARTA_PICCHE = 3;
```

- Non è un sistema molto efficace perché:
  - i valori che vengono poi letti sono solo interi
  - si possono sommare anche se non ha senso

Tipi enumerativi

- Un'alternativa consiste nell'usare stringhe ma:
  - non c'è tutela contro valori sbagliati (esempio: scrivere **`briscola="SPADE"`** oppure "**`QUORI`**")
  - non si possono usare nei costrutti **`switch`**
- Java 5 risolve questi problemi con **`enum`**
  - definisce tipi enumerativi in modo molto più potente di altri linguaggi (ad esempio C)
- La sintassi è, ad esempio:

```
enum Seme { FIORI, QUADRI, CUORI, PICCHE };
```

Tipi enumerativi

- I tipi enumerativi definiti con **`enum`** sono classi vere e proprie e hanno i seguenti vantaggi:
  - si possono usare in costrutti **`switch`**
  - non sono dei semplici alias di interi
  - hanno automaticamente i metodi **`toString`** e **`compareTo`** (sfrutta l'ordine in cui sono elencati)
- Inoltre possiedono ulteriori proprietà avanzate, consultabili sulla documentazione: aggiungere un comportamento, consentire cicli for in modo efficiente, eccetera.

Errori sui costruttori

- I costruttori non hanno tipo di ritorno
  - Creano oggetti della classe di cui hanno il nome.
- È possibile sbagliarsi e scrivere un tipo di ritorno nel costruttore. Cosa succede?
- Il compilatore vede un tipo di ritorno e crede sia la dichiarazione di un **metodo**.
  - Anche se è sconsigliato, è possibile avere un metodo con lo stesso nome di una classe.
  - Ovviamente si hanno effetti indesiderati (ad esempio errori quando si invoca **`new`**).

Errori sul metodo **`equals`**

- Il prototipo del metodo **`equals`** è:

```
boolean equals(Object o)
```

- Va sovrascritto per ogni classe per cui serve.
  - Di default ritorna true solo se due variabili riferiscono lo stesso oggetto, ma servirebbe per dire se contengono oggetti uguali.
- È possibile sbagliarsi dimenticando **`Object`**, ad esempio per la classe **`Studente`**, scrivendo **`equals(Studente s)`**. Cosa succede?
- Il metodo **`equals`** sarà **sovraccarico**.

Errori sul metodo **`equals`**

- Esempio sbagliato di **`equals`** per **`Studente`**:

```
public boolean equals (Studente s)
{ return s.matric == this.matric; }
```

- Esempio giusto di **`equals`** per **`Studente`**:

```
public boolean equals (Object s)
{ if ( s == null || !(s instanceof Studente) )
  { return false; }
  else
  { return (Studente s).matric == this.matric; }
}
```

o sollevare un'eccezione

- Possono presentarsi problemi simili anche per i metodi **`compareTo()`** e **`clone()`** .

Errori sul metodo `equals`

- Cosa succede usando la versione sbagliata?

```
Studente a = new Studente("Luca Rossi",21533);
Studente b = new Studente("L. Rossi",21533);
Object c = b; //lecito: Studente estende Object
```

- Le tre variabili dovrebbero essere tutte uguali secondo il metodo **`equals()`**. Invece:

`a.equals(b)` vale **true** (ed è **`equals(Studente)`**)
`b.equals(c)` vale **true** (ma è **`equals(Object)`**)
`c.equals(a)` vale **false** (ed è.. ?)
`a.equals(c)` vale **false** (ed è.. ?)

Override del tipo di ritorno

- Quando si sovrascrive un metodo in una sottoclasse, tipicamente il metodo riscritto è identico all'originale nell'intestazione, ma:
  - il modificatore di accesso (**`public`**, **`private`**...) può essere cambiato se viene allargato
  - dalla versione 5 di Java in poi, si può cambiare il tipo di ritorno con uno **sostituibile** al precedente (principio della ***covarianza dei tipi di ritorno***)
- Il caso tipico è che il nuovo tipo di ritorno sia una sottoclasse del vecchio.

Override del tipo di ritorno

- Un esempio notevole si ha nel metodo **`clone()`** che per la classe **`Object`** ritorna **`Object`**. Va quindi fatto un cast nel metodo chiamante.

```
MyClass a = new MyClass(parametri);
MyClass b = (MyClass) a.clone();
```

- Però si può evitare con un override di questo tipo:

```
public class MyClass implements Cloneable
{ ...
  public MyClass clone()
  { try
    { MyClass copia = (MyClass) super.clone();
      /* altre istruzioni per copia profonda */
      return copia;
    } catch (CloneNotSupportedException e) {...}
  }
}
```