

# Elementi di Object Oriented e Web Programming

Cristiano Longo, 2016  
[cristianolongo@gmail.com](mailto:cristianolongo@gmail.com)

Il corso verte sulle seguenti tematiche:

- programmazione orientata agli oggetti;
- il linguaggio java, sintassi, concetti e strumenti;
- design pattern;
- sezioni critiche e problemi di concorrenza;
- database relazionali;
- programmazione web;
  - protocolli per il web (HTTP, SOAP, REST);
  - servlet;
  - servizi web in Java;

# Le tecnologie Java

### Esercitazione

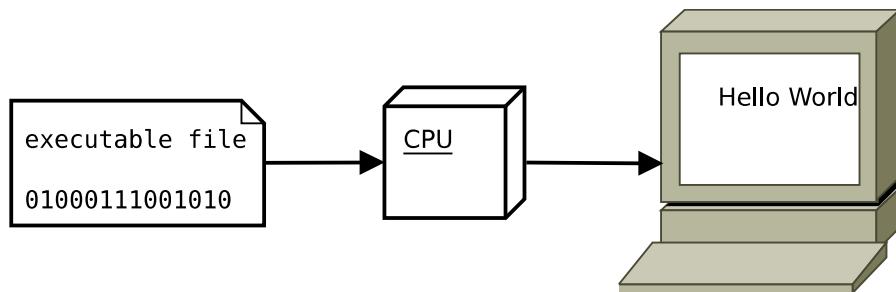
Se non ci sono parametri (a riga di comando) stampa “No Parameters”.

Altrimenti, per ogni parametro:

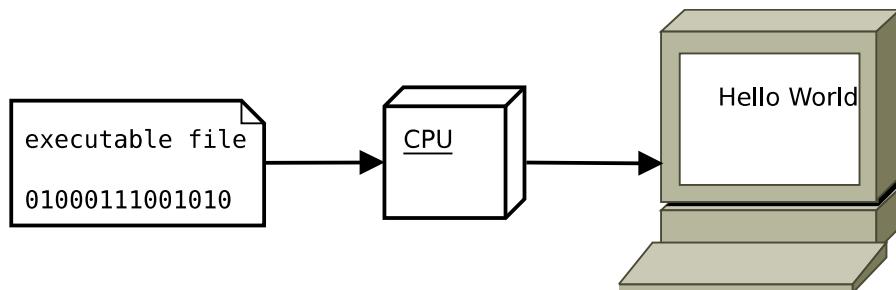
- se è pari stampa “even”
- se è dispari e multiplo di tre stampa “odd3Mult”
- altrimenti stampa “odd”

## Compilatori e Interpreti – file eseguibili

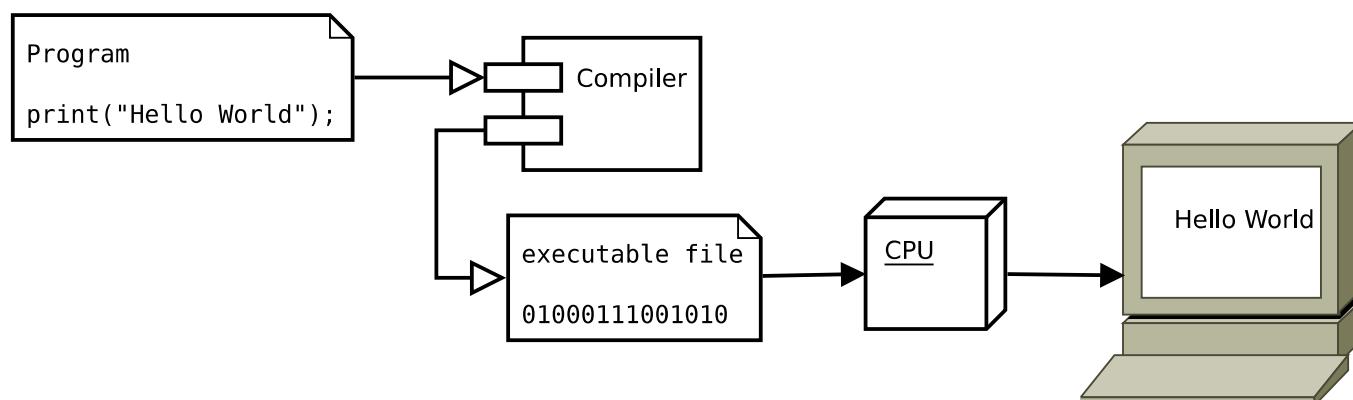
Le unità di processamento in un computer eseguono sequenze di istruzioni scritte in linguaggio specifico per l'architettura (ad. Esempio Assembly x86), chiamati **linguaggi macchina**. I file contenenti le sequenze di istruzioni sono detti **eseguibili**.



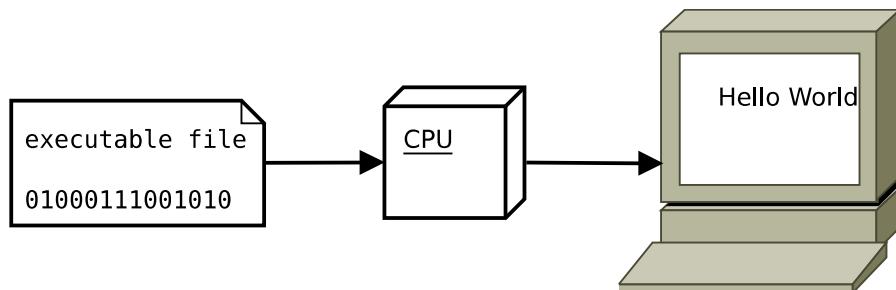
Le unità di processamento in un computer eseguono sequenze di istruzioni scritte in linguaggio specifico per l'architettura (ad. Esempio Assembly x86), chiamati **linguaggi macchina**. I file contenenti le sequenze di istruzioni sono detti **eseguibili**.



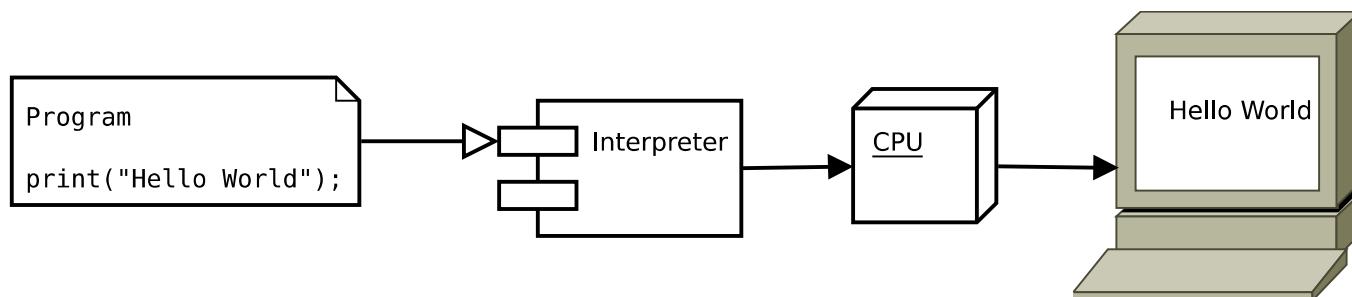
Un **compilatore** è un eseguibile che traduce un programma scritto in un linguaggio di *alto livello* in un eseguibile in linguaggio macchina.



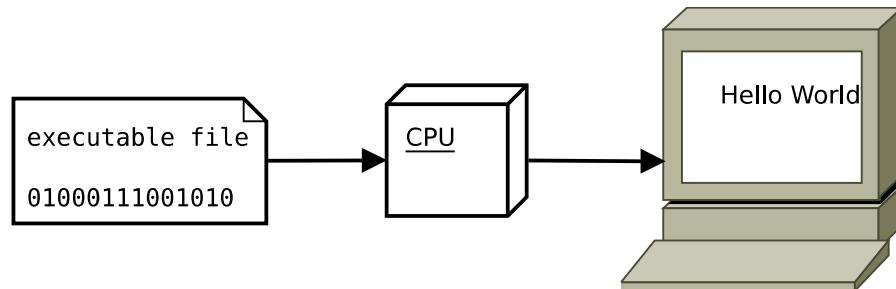
Le unità di processamento in un computer eseguono sequenze di istruzioni scritte in linguaggio specifico per l'architettura (ad. Esempio Assembly x86), chiamati **linguaggi macchina**. I file contenenti le sequenze di istruzioni sono detti **eseguibili**.



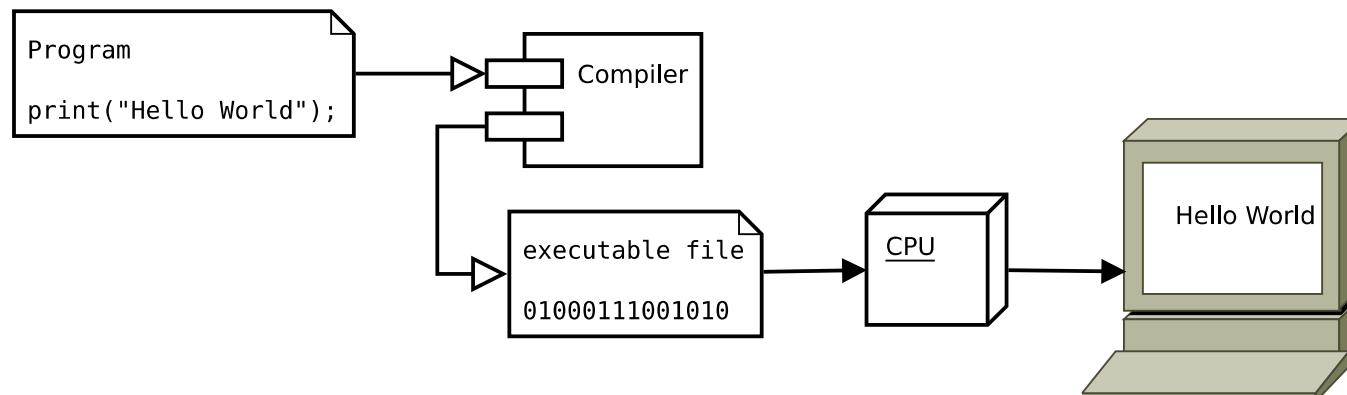
Un **interprete** è un programma eseguibile che interpreta ed esegue un programma scritto in un linguaggio di *alto livello*



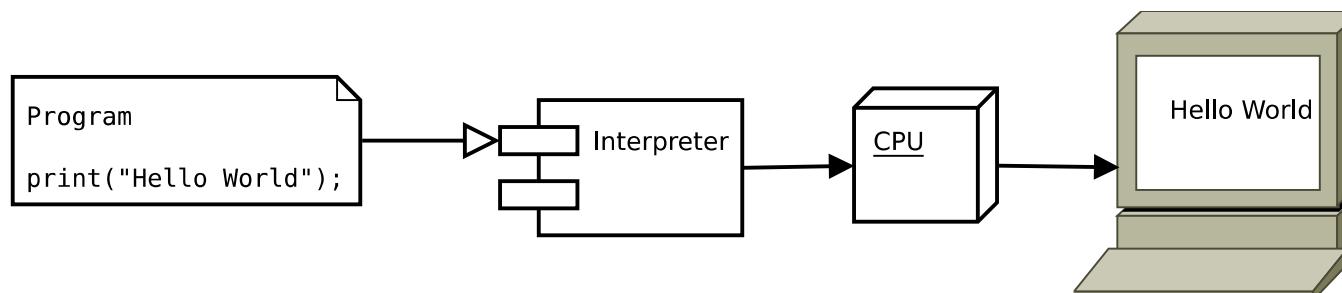
### File eseguibili



### Programmi Compilati

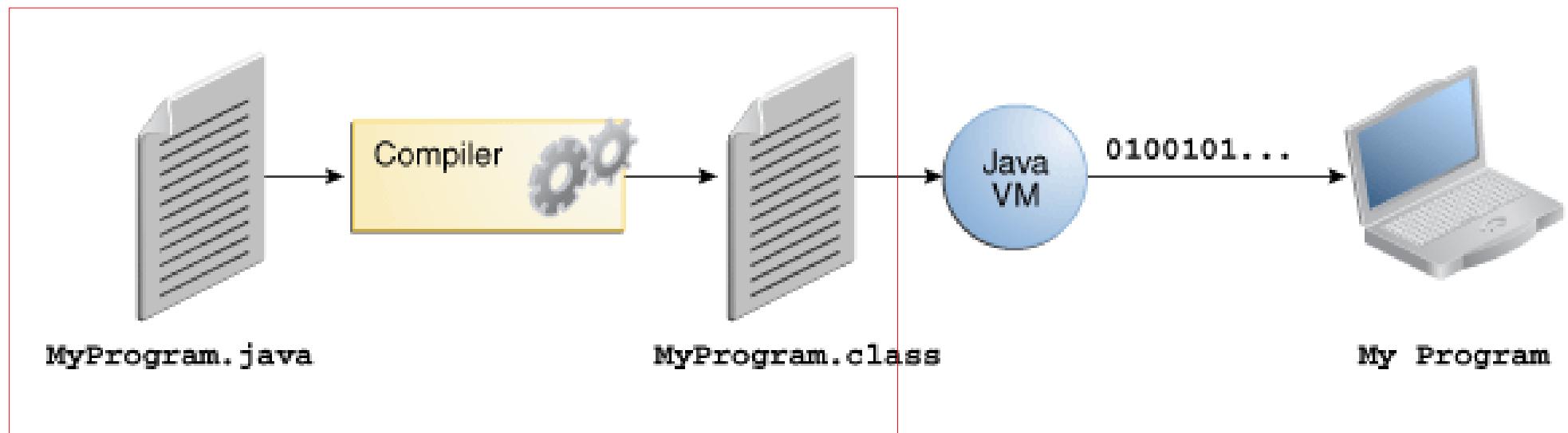


### Programmi Interpretati



L'approccio delle tecnologie Java è *ibrido*:

- il *compilatore java* genera (jSDK) del *bytecode* (linguaggio macchina) specifico per la *Java Virtual Machine (jre)*

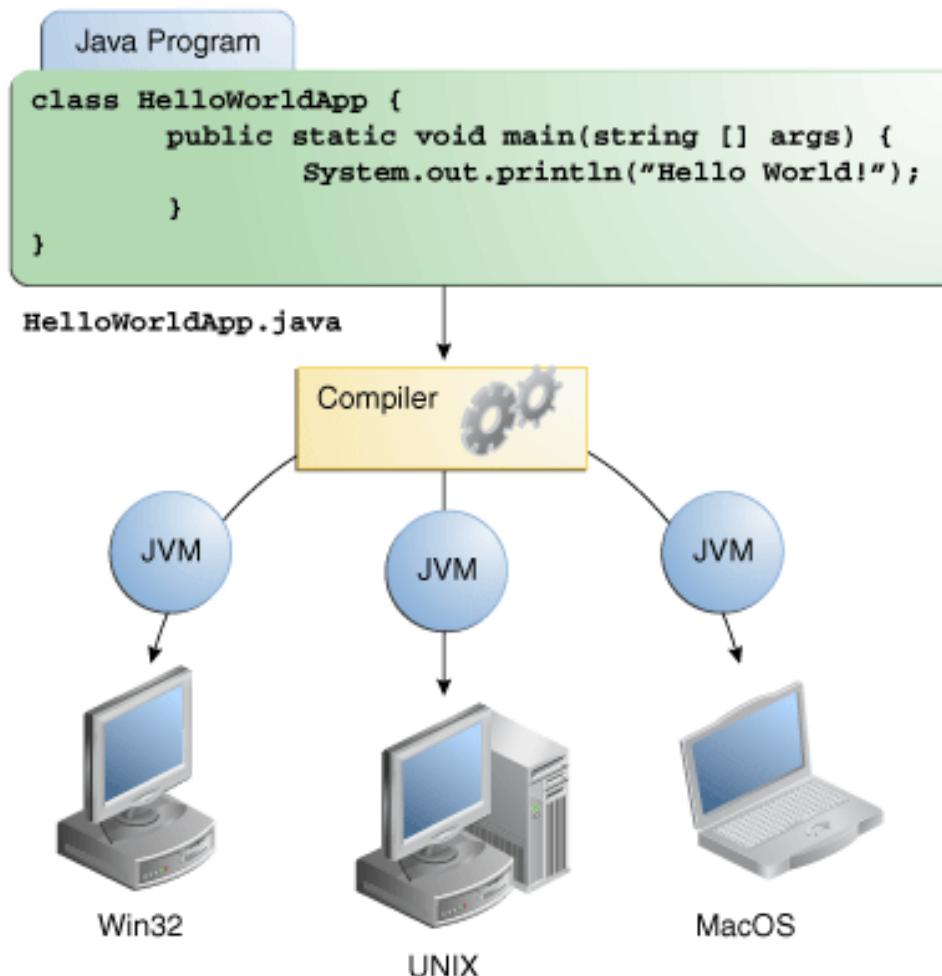


L'approccio delle tecnologie Java è *ibrido*:

- il *compilatore* java genera (jSDK) del *bytecode* (linguaggio macchina, cfle .class) specifico per la *Java Virtual Machine* (jre)
- la *Java Virtual Machine* (jre) è un interprete per bytecode java

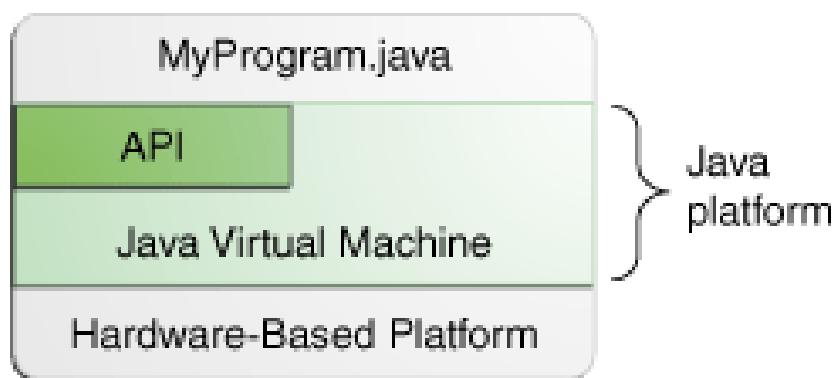
La JVM è disponibile per diversi sistemi operativi

*Write once, run everywhere.*



La piattaforma Java è costituita da due componenti:

- **Java Virtual Machine** (vedi prima)
- **Java Application Programming Interface (API)** è un insieme di librerie *native*.



Per eseguire bytecode java è sufficiente installare la Java Runtime Environment (JRE, esistono varie versioni generalmente retrocompatibili).

Per compilare programmi in Java è necessario invece installare un Java Development Kit (JDK), che contiene una JRE.

Per installare JDK e JRE vedi

[https://docs.oracle.com/javase/8/docs/technotes/guides/install/install\\_overview.html](https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html)

Vediamo come creare una semplice applicazione che stampi sul terminale la stringa “Hello World”.

Il sorgente di un applicativo Java è un file di testo (in linguaggio Java). Si può usare qualsiasi editor di testo per generarlo ed editarlo.

```
/**  
 * The HelloWorldApp class implements an application that  
 * simply prints "Hello World!" to standard output.  
 */  
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!"); // Display the string.  
    }  
}
```

Per compilare un sorgente Java si invoca il compilatore `javac`, che genererà un file `.class`.

```
> javac HelloWorldApp.java
> ls
...
HelloWorld.class
...
```

La JVM può essere invocata per eseguire un file .class attraverso il comando `java`. Si noti che il suffisso `.class` può essere omesso.

```
> java HelloWorldApp.java
```

```
Hello World!
```

Negli anni sono state prodotte varie versioni di Java: ..., 1.5, 1.6, 1.7, 1.8 (detta Java 8), con le corrispondenti JDK e JRE. Per conoscere quella correntemente installata sul proprio computer usare il comando `java -version`.

```
>java -version
openjdk version "1.8.0_91"
OpenJDK Runtime Environment (build 1.8.0_91-8u91-b14-
3ubuntu1~16.04.1-b14)
OpenJDK 64-Bit Server VM (build 25.91-b14, mixed mode)
```

*Eclipse* è un ambiente di sviluppo integrato (IDE) realizzato in java. Può essere utilizzato per diversi linguaggi.



<http://www.eclipse.org>

In Eclipse un *workspace* è una cartella nella quale vengono salvati i *metadati* dei progetti e alcune impostazioni.

E' possibile utilizzare diversi workspace, uno per volta.

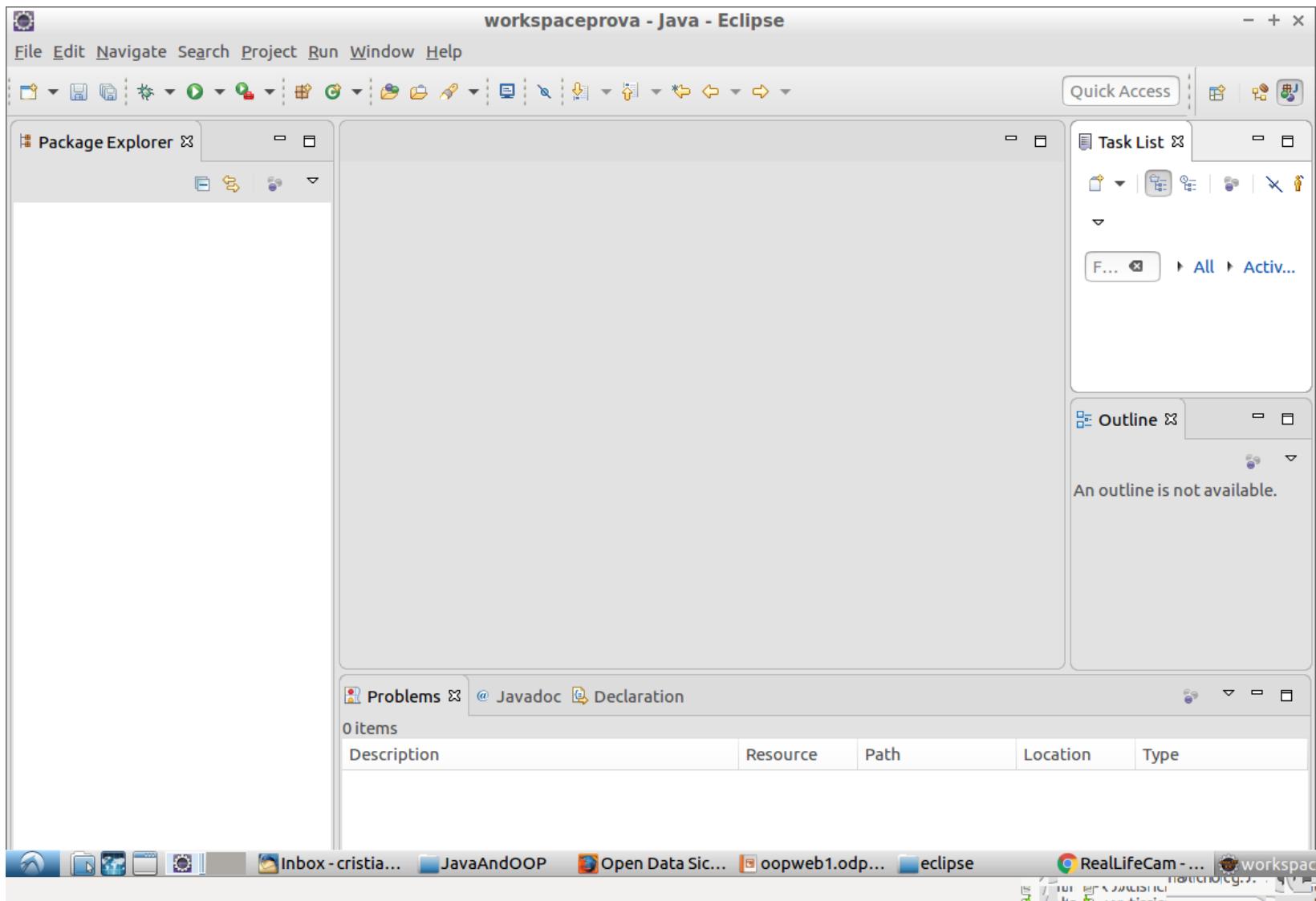
I progetti veri e propri possono essere posizionati in qualsiasi locazione del filesystem, non necessariamente all'interno del workspace.

I principali parametri di configurazione di Eclipse e dei progetti sono accessibili dal menù principale Window – Preferences.

Particolarmente rilevante tra le preferenze è la sezione Java – Installed JREs che permette di specificare le installazioni java e quella da usare.

Accertarsi di usare una JDK e non una JRE.

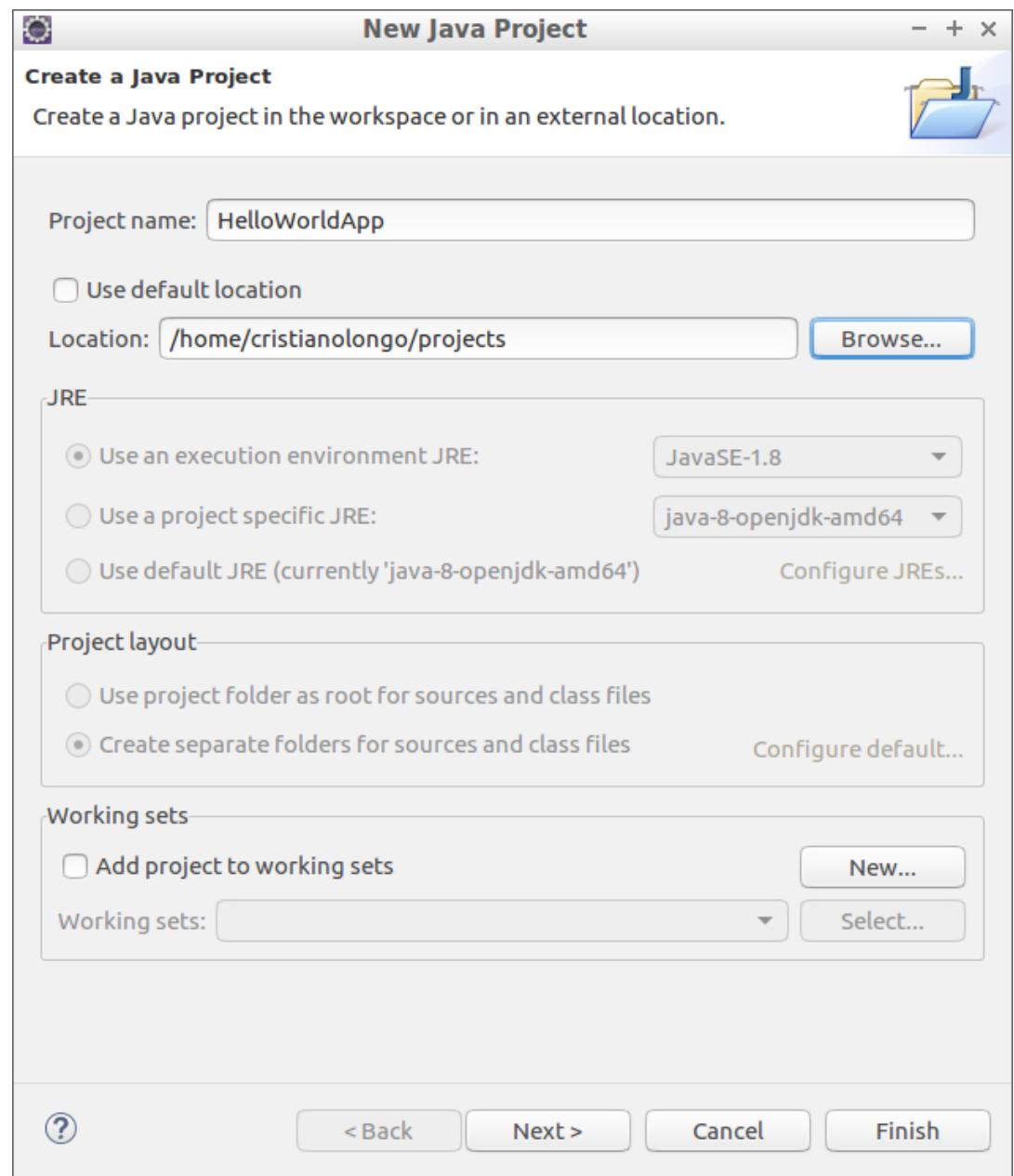
La *perspective* Java (menù – window – perspectives) si presenta con un pannello a sinistra per i progetti, un *editor* al centro e varie altre *view* e *tabs* a destra e in basso.



Per creare un nuovo progetto java in Eclipse dal menù File – New – Java Project. Si apre una finestra per inserire i dettagli del progetto.

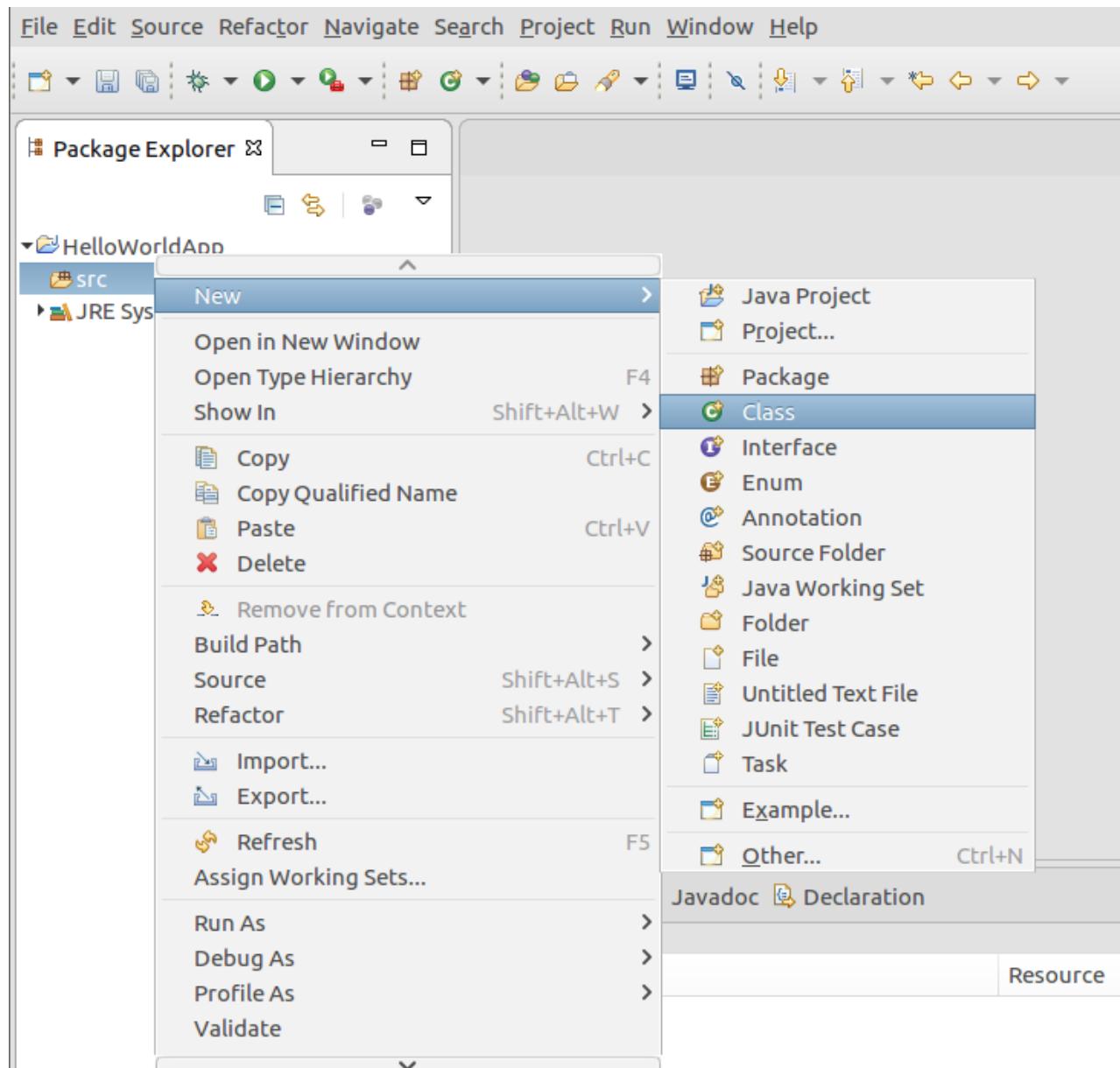
Il campo location permette di specificare una posizione esterna al workspace.

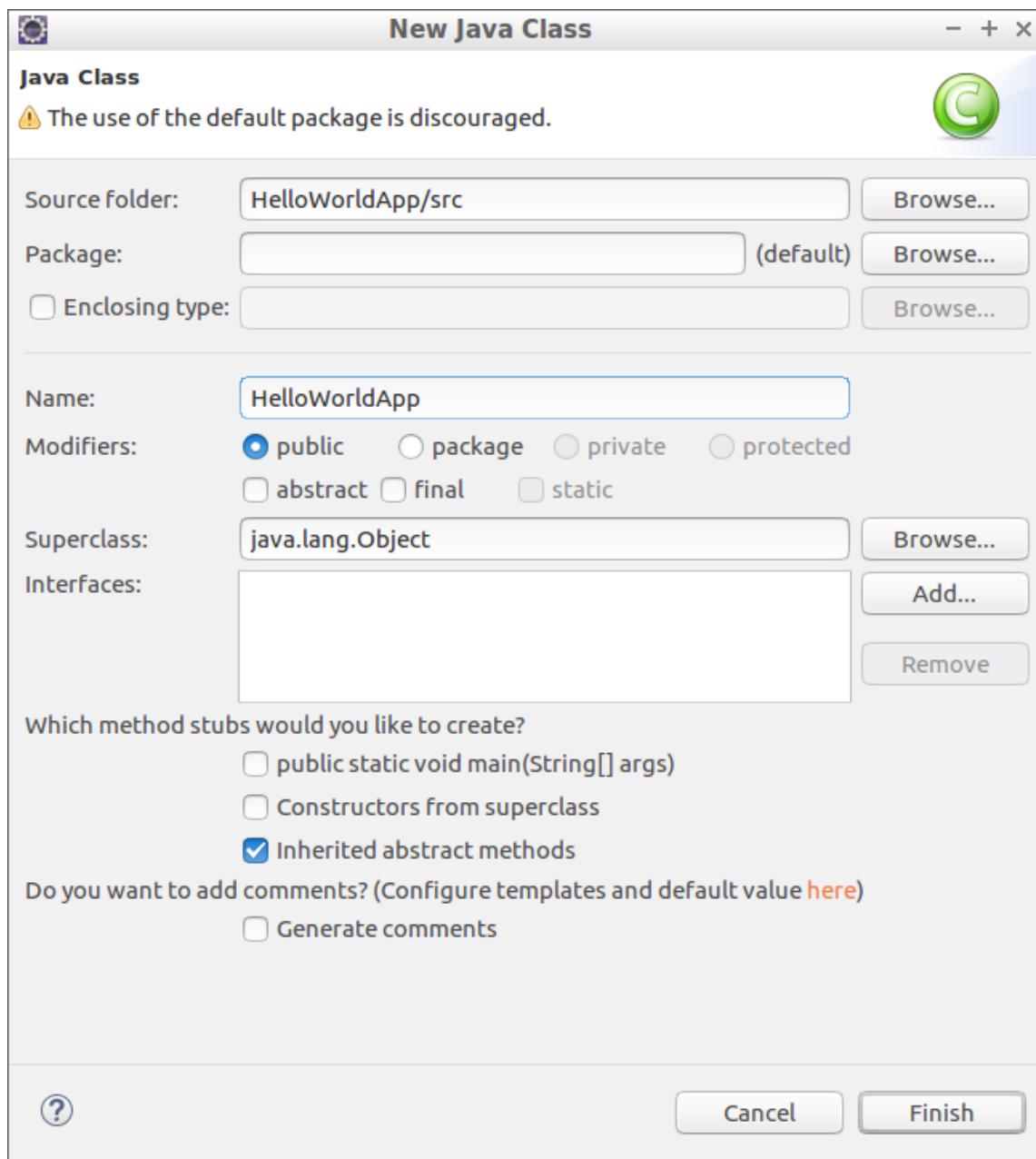
E' possibile specificare la JRE nel caso in cui ne siano disponibili più di una (questo parametro si può cambiare successivamente)



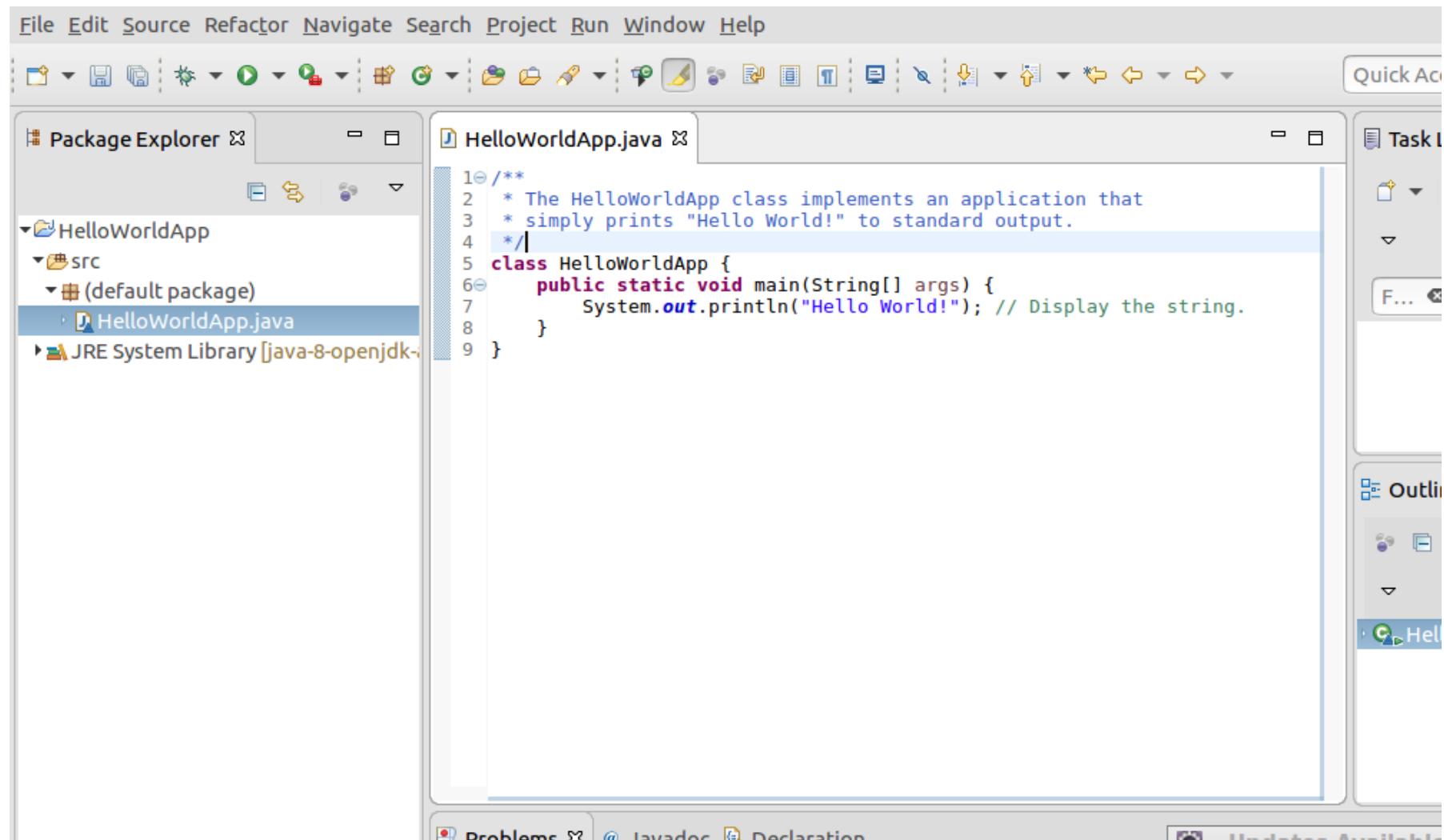
Il progetto sarà visibile nel pannello di sinistra.

Col tasto destro sul folder src all'interno del progetto è possibile creare una classe vuota.

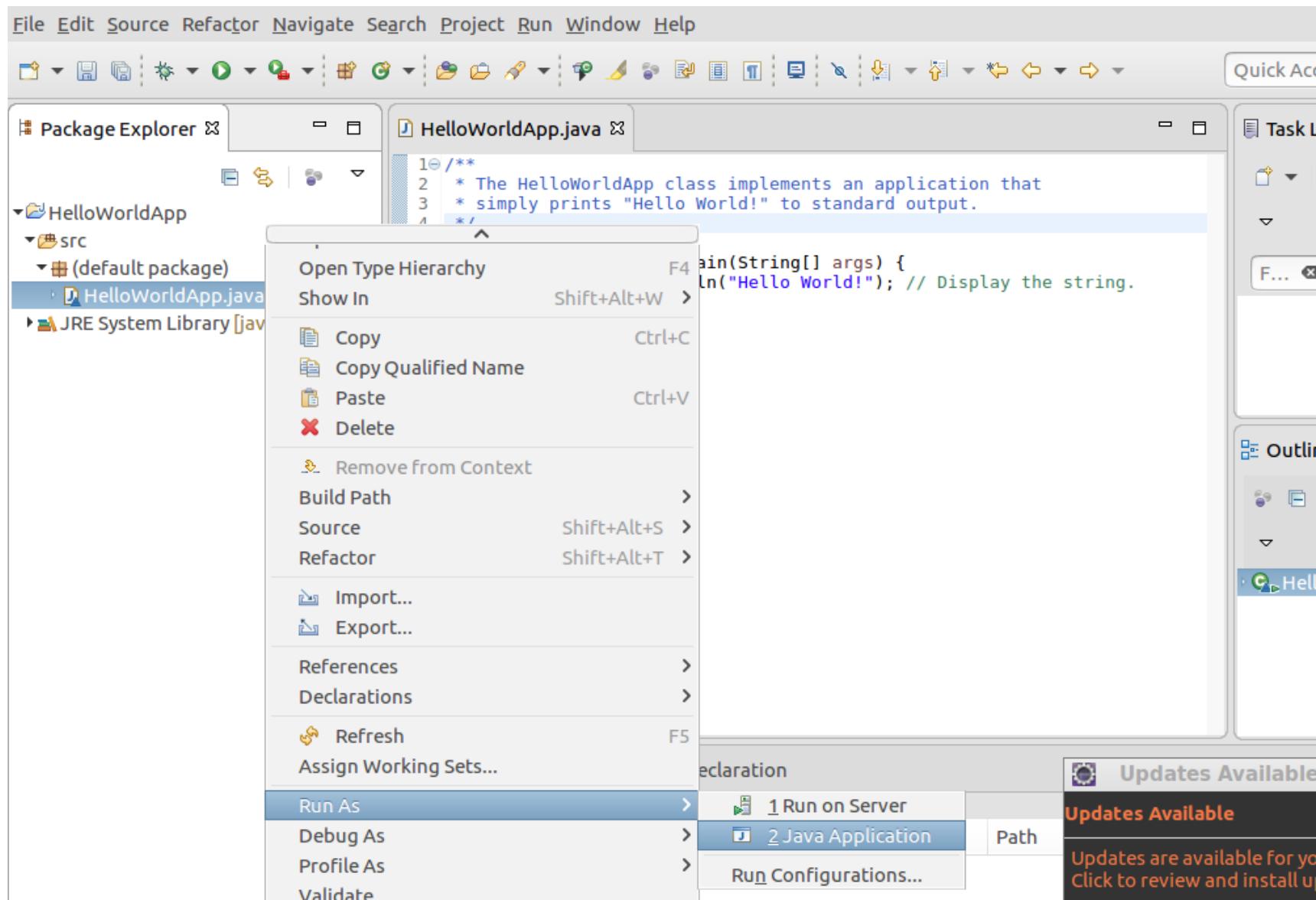




Con un doppio click sulla classe il sorgente verrà aperto nell'editor per essere modificato.



Ultimate le modifiche sarà possibile eseguire il codice con tasto destro sulla classe e nel menù contestuale Run As - Java Application.



# Il Linguaggio Java

Java è un linguaggio *imperativo* che segue il paradigma della programmazione orientata ad oggetti.

La specifica di Java 8 è disponibile all'indirizzo

<http://docs.oracle.com/javase/specs/jls/se8/html/index.html>

Altre risorse interessanti

- [https://en.wikipedia.org/wiki/Java\\_syntax](https://en.wikipedia.org/wiki/Java_syntax)
- <http://www.oracle.com/technetwork/java/langenv-140151.html>
- <https://docs.oracle.com/javase/tutorial/java/TOC.html>

E` possibile inserire commenti in ogni parte del codice. I commenti possono essere *inline* oppure estendersi su più linee. Di questi ultimi, quelli che iniziano con `/**` sono *Documentation comments*.

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 *
 * This is a documentation comment.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        /* Two lines
         * comment
         */
        System.out.println("Hello World!"); // inline comment
    }
}
```

I *blocchi* di codice sono delimitati da { } e possono essere annidati.

```
/**  
 * The HelloWorldApp class implements an application that  
 * simply prints "Hello World!" to standard output.  
 *  
 * This is a documentation comment.  
 */  
class HelloWorldApp {  
    public static void main(String[] args) {  
        /* Two lines  
         * comment  
         */  
        System.out.println("Hello World!"); // inline comment  
    }  
}
```

La sintassi per la dichiarazione delle variabili è la seguente:

[final] <type> <variableName> [= <espressione>]

```
/**  
 * Some examples of variable declarations.  
 */  
class VariableDeclaratioExamples {  
    public static void main(String[] args) {  
        int v1;  
        char v2 = 'c';  
        final int v3 = 3;  
    }  
}
```

E' possibile assegnare un valore ad una variabile nella dichiarazione della stessa o successivamente.

Una variabile *final* può essere assegnata una sola volta.

```
/**  
 * Some examples of variable declarations.  
 */  
class VariableDeclaratioExamples {  
    public static void main(String[] args) {  
        int v1;  
        char v2 = 'c';  
        final int v3 = 3;  
    }  
}
```

I tipi primitivi forniti dal linguaggio Java sono i seguenti:

- byte
- short
- char
- int
- long
- float
- double
- boolean

NOTA1 : I tipi numerici sono tutti *signed* (con segno)

NOTA2 : le *stringhe* (delimitate da “ ”) non sono di un tipo primitivo ma sono *oggetti* (vedremo dopo).

Il valore di una variabile può essere assegnato mediante una espressione definita. Di seguito i più comuni operatori per le espressioni.

**Aritmentici** +, -, \*, /, %, ^, ++ (unario), – (unario)

Esempi: 1+2, 5-4, 3\*2,

**Booleani** &&, ||, !, ==, !=

**Concatenazione di stringhe** +

Esempio “ciao” + “ciao” = “ciao ciao”

**Operatore Condizionale** <cond> ? <v1> : <v2>

Se <cond> è vero allora <v1>, altrimenti <v2>

Il valore di una variabile può essere assegnato mediante una espressione definita. Di seguito i più comuni operatori per le espressioni.

**Aritmentici** +, -, \*, /, %, ^, ++ (unario), – (unario)

Esempi: 1+2, 5-4, 3\*2,

**Booleani** &&, ||, !, ==, !=**Concatenazione di stringhe** +

Esempio "ciao "+" ciao"="ciao ciao"

**Operatore Condizionale** <cond> ? <v1> : <v2>

Se <cond> è vero allora <v1>, altrimenti <v2>

Esistono forme abbreviate per gli assegnamenti che coinvolgano operatori: a += b, a -= b, a \*= b, a /= b, a %= b, a <<= b, a >>= b, a >>>= b, a &= b, a |= b, a ^= b

Vedi

<http://docs.oracle.com/javase/specs/jls/se8/html/jls-16.html#jls-16.1>

Un array è una sequenza (ad accesso casuale) di oggetti dello stesso tipo. Una variabile di tipo array può essere definita in due modi equivalenti

```
<type>[] <name>  
<type> <name>[]
```

## Esempi

```
int[] a;  
int a[];
```

La dichiarazione di un array prevede invece di specificarne, oltre al tipo, la dimensione. Nella definizione inline è possibile anche specificare gli elementi.

```
int[] a;  
a = new int[10]; //array di 10 interi  
int b[] = {1, 2, 3}; //array che contiene gli elementi 1,2 e 3
```

Il linguaggio Java fornisce le seguenti istruzioni per il controllo del flusso di esecuzione

- if, else, elseif
- switch
- while, do
- for
- break, continue, return

```
if (<cond>)
    <body>
```

Se `<cond>` è vero esegue `<body>`

```
if (<cond>)
    <body>
```

```
else
```

```
    <elseBody>
```

Se `<cond>` è vero esegue `<body>` altrimenti esegue `<elseBody>`

```
if (<cond1>)
    <body1>
```

```
else if (<cond2>)
    <body2>
```

...

```
else if (<condn>)
    <bodyn>
```

```
else
```

```
    <elseBody>
```

Se `<cond>` è vero esegue `<body>` altrimenti se `<cond1>` è vero esegue `<body1>`

Altrimenti ... se `<condn>` è vero esegue `<bodyn>` altrimenti (se nessuna delle precedenti condizioni è vera) esegue `<elseBody>`

Vedi

<http://docs.oracle.com/javase/specs/jls/se8/html/jls-16.html#jls-16.2.7>

```
switch (<expression>) {  
    case <value_1> : <body_1>  
    case <value_2> : <body_2>  
    ...  
    case <value_n> : <body_n>  
    default : <bodyDefault>  
}
```

La clausola default è opzionale.

Se <value*i*> è vero, per qualche *i* compreso tra 1 ed n, esegue  
<body\_<i>>, <body\_<(i+1)>>, ..., <body\_n> e <bodyDefault>, se presente.  
Altrimenti esegue solo <bodyDefault>, se presente.

### Esercitazione

Se non ci sono parametri (a riga di comando) stampa “No Parameters”.

Altrimenti, per ogni parametro:

- se è pari stampa “even”
- se è dispari e multiplo di tre stampa “odd3Mult”
- altrimenti stampa “odd”

# Object Oriented Programming

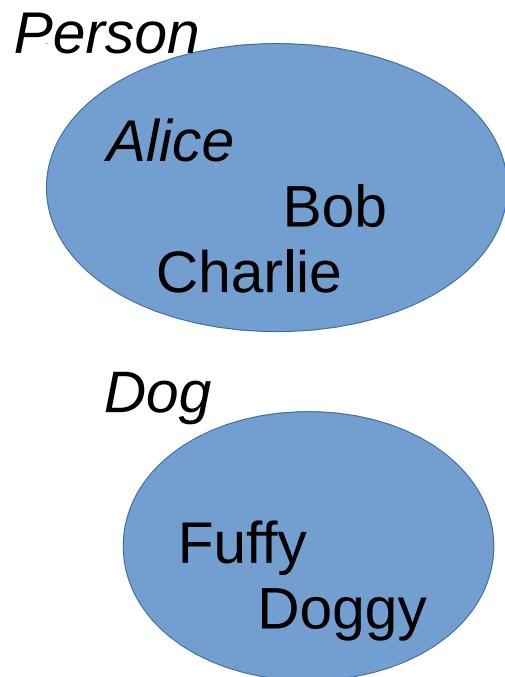
La *programmazione orientata agli oggetti (OOP)* è un paradigma di programmazione basato sulla nozione di *oggetto*. Un oggetto può contenere dati, sotto forma di attributi, e codice, sotto forma di metodi.

```
final int[] a = {1,2,3};  
final int[] b = {1,2};  
  
/* attributo length di un array  
 * contiene la lunghezza di due array.  
 */  
System.out.println("Lunghezza di a="+a.length);  
System.out.println("Lunghezza di b="+b.length);  
  
/* il metodo equals permette di verificare se due oggetti  
 * sono uguali.  
 */  
System.out.println("a=b ? "+a.equals(b));
```

Java è *class-based*: gli oggetti vengono suddivisi in *Classi*. Ogni classe definisce gli attributi e i metodi di tutte le proprie *istanze* (oggetti).

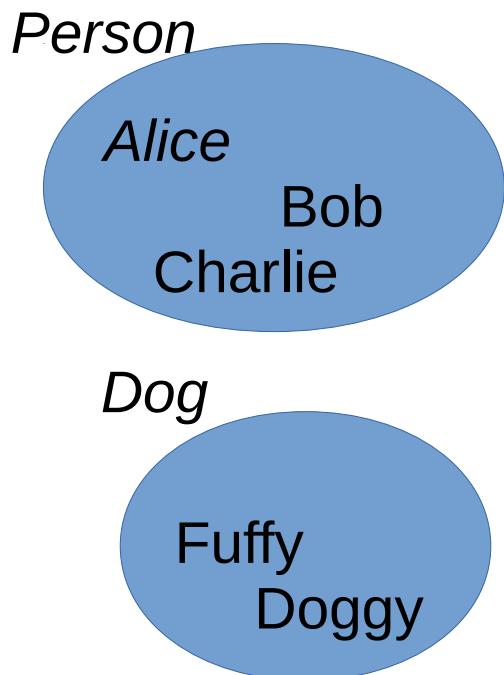
```
/**  
 * A class with attributes and methods.  
 *  
 * @author Cristiano Longo  
 *  
 */  
public class ExampleClass {  
    int a = 1; //an attribute  
  
    /**  
     * Increment the value of the attribute.  
     */  
    void inc() {  
        ++a;  
    }  
}
```

Intuitivamente, le classi rappresentano insiemi di oggetti (concetti) del mondo reale mentre le istanze di una classe rappresentano gli oggetti veri e propri nell'insieme. Le classi sono *tipi* a tutti gli effetti.



Intuitivamente, le classi rappresentano insiemi di oggetti (concetti) del mondo reale mentre le istanze di una classe rappresentano gli oggetti veri e propri nell'insieme. Le classi sono *tipi* a tutti gli effetti.

La parola chiave *new* permette di costruire istanze di una classe.



```
Person alice = new Person();  
Person bob = new Person();  
Person charlie = new Person();  
  
Dog fuffy = new Dog();  
Dog doggy = new Dog();
```

Gli attributi seguono le stesse regole delle variabili in merito a dichiarazione e gestione dei valori. La loro *visibilità* (scope) è tutta la classe. Solitamente rappresentano lo *stato* di un oggetto.

```
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter.  
     */  
    void inc() {  
        value++;  
    }  
}
```

Il valore di un attributo in una istanza può essere ottenuto, al di fuori della classe, con la sintassi

<instance>.<attrName>

```
Counter c = new Counter();
System.out.println("Il valore del contatore e' "+c.value);
```

I *metodi* di un oggetto ne modellano il comportamento. Possono essere visti come delle *porte* per inviare messaggi ad un oggetto.

```
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter.  
     */  
    void inc() {  
        value++;  
    }  
  
    /**  
     * Increment the value of the counter.  
     */  
    void dec() {  
        value--;  
    }  
}
```

Un metodo di una istanza valore di un attributo in una istanza può essere invocato, al di fuori della classe, con la sintassi

<instance>.<methodName>()

```
Counter c = new Counter();
c.inc();
System.out.println("Il valore del contatore e' "+c.value);
```

I metodi possono avere dei *parametri*.

```
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter.  
     */  
    void inc(int n){  
        value+=n;  
    }  
}
```

I metodi possono avere dei *parametri*. E' buona pratica dichiararli final.

```
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter.  
     * @param n number of units the counter will be incremented  
     */  
    void inc(final int n){  
        value+=n;  
    }  
}
```

Vedi <http://www.oracle.com/technetwork/java/object-141359.html#343>.

Nella stessa classe è possibile dichiarare metodi con lo stesso nome ma con parametri differenti.

```
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter of one,  
     */  
    void inc(){  
        value++;  
    }  
  
    /**  
     * Increment the value of the counter.  
     *  
     * @param n number of units the counter will be incremented  
     */  
    void inc(final int n){  
        value+=n;  
    }  
}
```

Un metodo può restituire un valore quando invocato grazie alla parola chiave `return`. E' necessario specificare il tipo di ritorno nella firma del metodo.

```
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter of one unit.  
     *  
     * @return the new value of the counter  
     */  
    int inc() {  
        return ++value; //note that we used the pre-increment op  
    }  
}  
  
----  
  
Counter c = new Counter();  
System.out.println("The value of the counter incremented "+c.inc());
```

Il **costruttore** è un metodo della classe che viene invocato automaticamente subito dopo l'istanziazione di un oggetto. Ha lo stesso nome della classe e nessun tipo di ritorno. Solitamente viene usato per inizializzare gli attributi.

```
public class Counter {  
    int value;  
  
    Counter() { //this is the constructor  
        value=0;  
    }  
    /**  
     * Increment the value of the counter of one unit.  
     *  
     * @return the new value of the counter  
     */  
    int inc(){  
        return ++value; //note that we used the pre-increment op  
    }  
----  
    Counter c = new Counter();  
    System.out.println("The value of the counter "+c); //will output 0
```

Una classe può avere svariati costruttori che si differenziano per i parametri.

```
public class Counter {  
    int value;  
  
    Counter() { //this is the constructor  
        value=0;  
    }  
  
    Counter(int initialValue) { //this is another constructor  
        value=initialValue;  
    }  
    /**  
     * Increment the value of the counter of one unit.  
     *  
     * @return the new value of the counter  
     */  
    int inc() {  
        return ++value; //note that we used the pre-increment op  
    }  
---  
Counter c1 = new Counter(); //initialized with value 0  
Counter c2 = new Counter(6); //initialized with value 6
```

Attributo e metodo è associato un livello di visibilità.

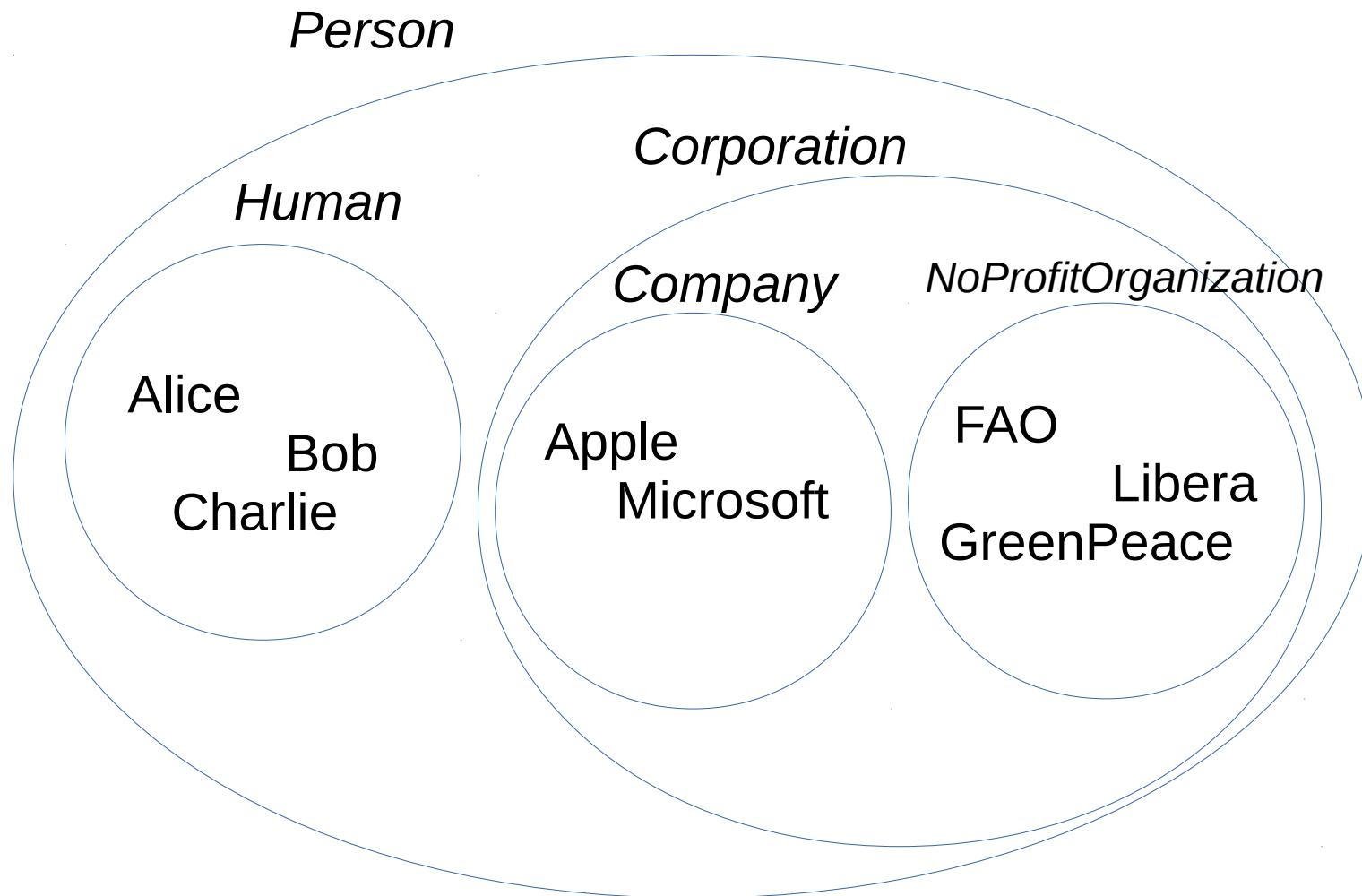
- Un attributo (metodo) *public* è visibile sempre.
- Un attributo (metodo) *private* è visibile solo all'interno della stessa classe.
- Un attributo (metodo) per il quale non è specificata la visibilità è visibile solo all'interno dello stesso package (si dice *package-private*).

Gli attributi *statici* sono collegati direttamente alla classe. Di conseguenza hanno lo stesso valore in tutte le istanze della stessa classe.

```
public class Entity {  
    private static int nextId=0;  
    public final int entityId;  
    Entity() {  
        entityId=(nextId)++;  
    }  
}  
  
-----  
Entity e0 = new Entity(); //e0.entity=0, Entity.nextId==1  
Entity e1 = new Entity(); //e1.entity=1, Entity.nextId==2
```

# Ereditarietà

Le classi rappresentano insiemi di oggetti. E' possibile rappresentare *gerarchie* di classi



In Java è possibile definire una classe come *sottoclasse* di una classe padre con la parola chiave *extends*.

```
public class Person {  
    ...  
}  
  
public class Human extends Person {  
    ...  
}  
  
public class Corporation extends Person {  
    ...  
}  
  
public class Company extends Corporation {  
    ...  
}  
  
public class NoProfitOrganization extends Corporation {  
    ...  
}
```

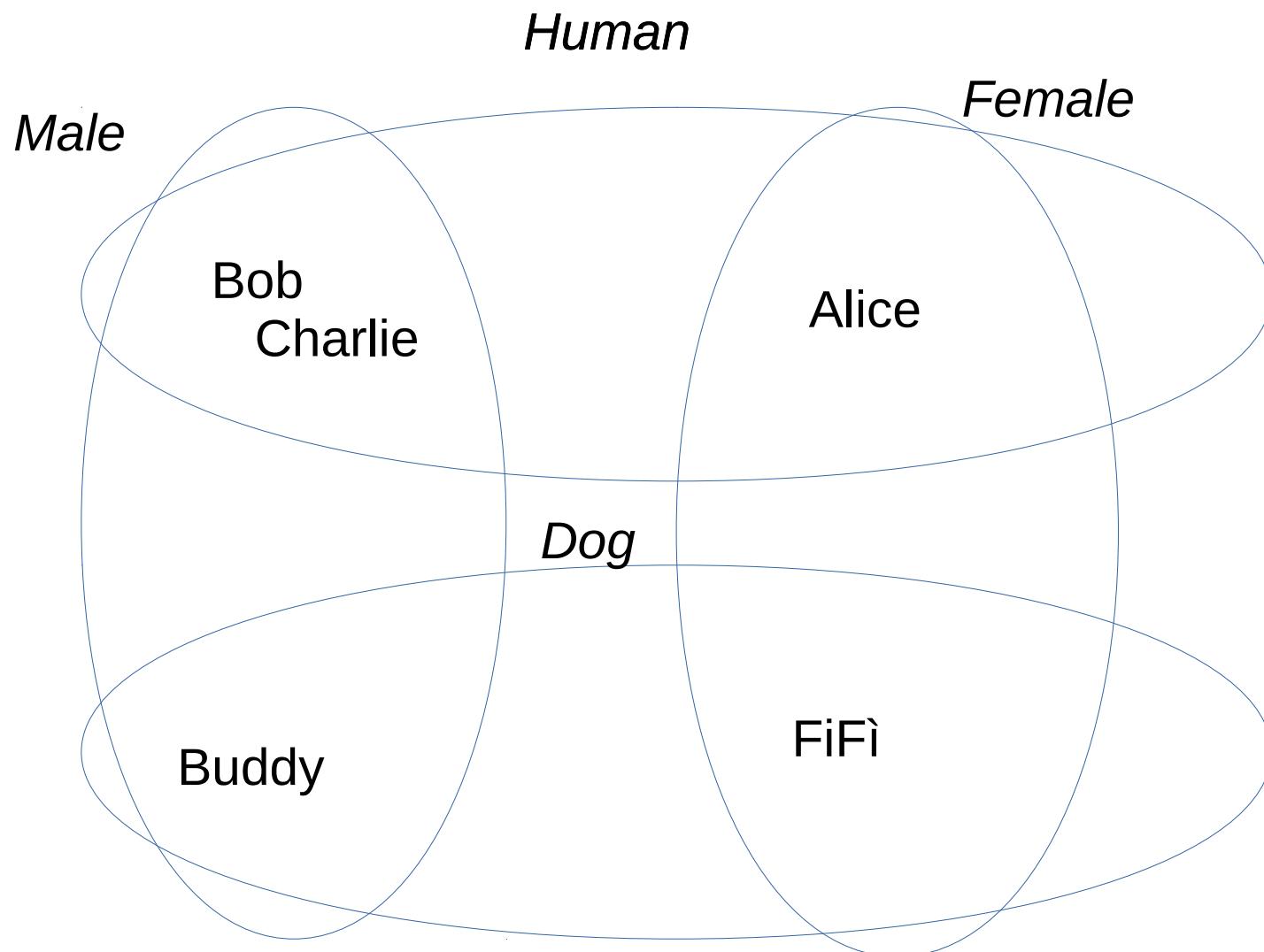
Una sottoclasse *eredita* tutti metodi e gli attributi della superclasse.

```
public class Person {  
    String cf;  
}  
  
public class Human extends Person {  
    // String cf; is implicit  
    boolean isFemale;  
}  
  
public class Greeter extends Person {  
    void sayWelcome(final Human p) {  
        System.out.println("Hello "+(p.isFemale ? "Mrs." : "Mr.") +  
            p.cf);  
    }  
}
```

Una sottoclasse *eredita* tutti metodi e gli attributi della superclasse, ma ha visibilità solo su metodi e attributi non privati della superclasse.

```
public class Person {  
    protected String cf;  
  
    protected setCF(final String cf) {  
        this.cf=cf;  
    }  
}  
  
public class Human extends Person {  
    // String cf; is implicit  
    public boolean isFemale;  
    public Human(final String cf, final boolean isFemale) {  
        super.setCF(cf);  
        this.isFemale=isFemale;  
    }  
}  
  
public class Greeter extends Person {  
    void sayWelcome(final Human p) {  
        System.out.println("Hello "+(p.isFemale ? "Mrs." : "Mr.") +  
            p.cf);  
    }  
}
```

Spesso una rappresentazione puramente gerarchica non è sufficiente a modellare il dominio di conoscenza.



Un interfaccia è simile ad una classe ma per I metodi vengono fornite solo le firme e non il corpo.

```
public interface Human {  
    void sayHello();  
}  
  
public interface Dog {  
    void sayBau();  
}  
  
public interface Male{  
    void hunt();  
}  
  
public interface Female{  
    void nurse();  
}
```

Per utilizzare le interfaccie bisogna fornirne delle implementazioni.

```
public interface Human {  
    void sayHello();  
}  
  
public class SmallVoicedHuman implements Human {  
    public void sayHello() {  
        System.out.println("hello");  
    }  
}  
  
//another implementation of Human  
public class LoudVoicedHuman implements Human {  
    public void sayHello() {  
        System.out.println("HELLO");  
    }  
}
```

Una classe può implementare diverse interfacce.

```
public interface Human {  
    void sayHello();  
}  
  
public interface Male{  
    void hunt();  
}  
  
public class Man implements Human, Male{  
    public void sayHello() {  
        System.out.println("HELLO");  
    }  
  
    public void hunt() {  
        System.out.println("I'm hunting");  
    }  
}
```

E' possibile definire classi "incomplete" demandando l'implementazione di alcuni metodi *astratti* alle sottoclassi.

```
public abstract class Vehicle{
    private final String plate;
    protected Vehicle(final String plate) {
        this.plate=plate;
    }
    public final getPlate() {
        return plate;
    }
    public abstract void move();
}

public class Car extends Vehicle{
    public Car(final String plate) {
        super(plate);
    }

    public void move() {
        //togli il freno a mano, premi la frizione, gira la chiave
    }
}
```

# Gestione Errori

Il meccanismo delle *Eccezioni* permette di sancire e gestire situazioni di errore o non previste.

```
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter.  
     * @param n number of units the counter will be incremented  
     */  
    void inc(final int n) {  
        If (n<0)  
            throw new IllegalArgumentException("n must be >=0");  
        value+=n;  
    }  
}
```

Le eccezioni vengono sollevate con la parola chiave throw.

```
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter.  
     * @param n number of units the counter will be incremented  
     */  
    void inc(final int n) {  
        If (n<0)  
            throw new IllegalArgumentException("n must be >=0");  
        value+=n;  
    }  
}
```

Le eccezioni eventualmente *sollevate* possono essere gestite con dei blocchi try-catch.

```
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter.  
     * @param n number of units the counter will be incremented  
     */  
    void inc(final int n){  
        If (n<0)  
            throw new IllegalArgumentException("n must be >=0");  
        value+=n;  
    }  
}  
...  
Counter c=new Counter();  
try{  
    c.inc(Integer.parseInt(args[0]));  
} catch(final IllegalArgumentException e) {  
    e.printStackTrace();  
}
```

Tutte le eccezioni estendono la classe Throwable. I metodi che lanciano eccezioni devono dichiararlo (`throws`).

```
public class NegativeNumberException extends Throwable{  
  
    public NegativeNumberException() {  
        super("Negative number not allowed as parameter");  
    }  
}  
  
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter.  
     * @param n number of units the counter will be incremented  
     */  
    void inc(final int n) throws NegativeNumberException{  
        If (n<0)  
            throw new NegativeNumberException("n must be >=0");  
        value+=n;  
    }  
}
```

Tutte le eccezioni estendono la classe Throwable. I metodi che lanciano eccezioni devono dichiararlo (`throws`).

```
public class NegativeNumberException extends Throwable{  
  
    public NegativeNumberException() {  
        super("Negative number not allowed as parameter");  
    }  
}  
  
public class Counter {  
    int value = 1;  
  
    /**  
     * Increment the value of the counter.  
     * @param n number of units the counter will be incremented  
     */  
    void inc(final int n) throws NegativeNumberException{  
        If (n<0)  
            throw new NegativeNumberException("n must be >=0");  
        value+=n;  
    }  
}
```

L'unica eccezione sono le classi che estendono `RuntimeException`, che non devono essere necessariamente gestite.

## Gestione delle Eccezioni - finally

Al termine del blocco try-catch è possibile inserire una clausola *finally* che verrà eseguita comunque dopo il blocco try catch ed eventualmente dopo l'eccezione.

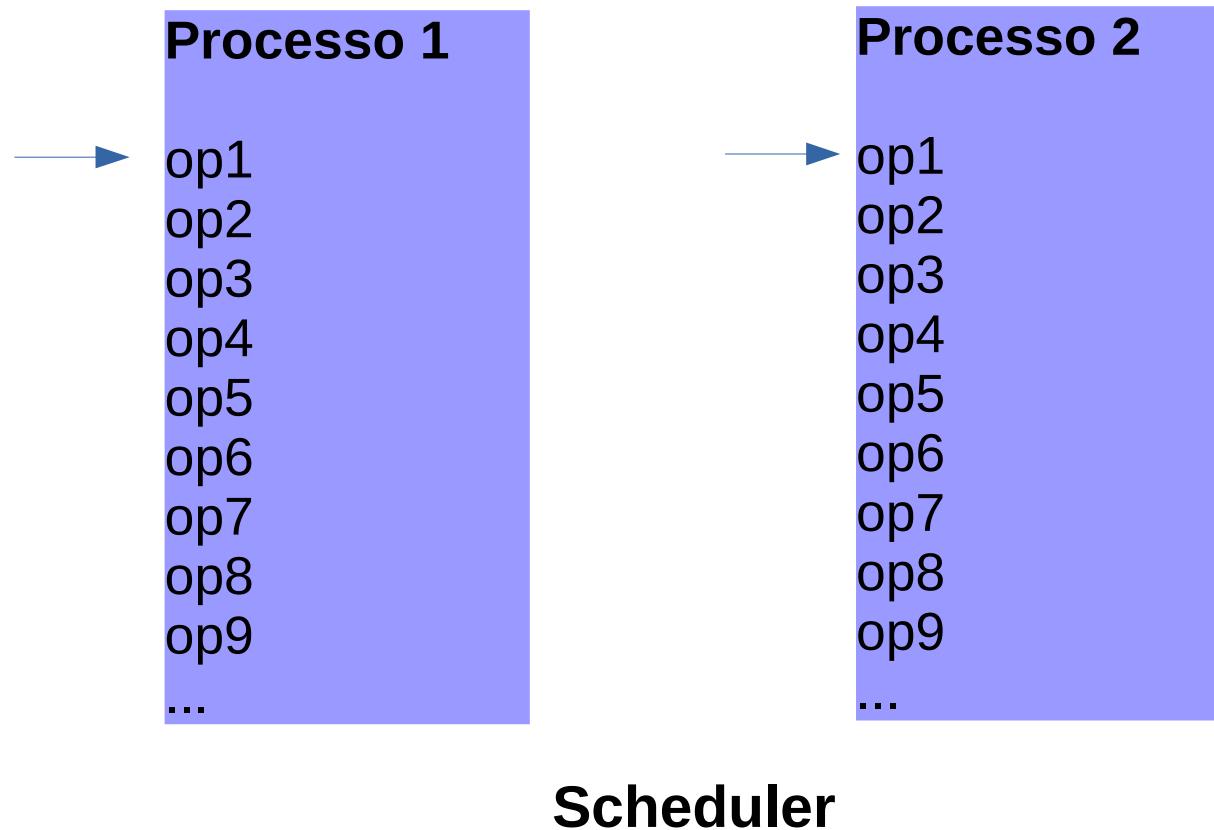
```
...
Counter c=new Counter();
try{
    c.inc(Integer.parseInt(args[0]));
} catch(final NegativeNumberException e) {
    e.printStackTrace();
} catch(final ParseException e) {
    e.printStackTrace();
} finally {
    System.out.println(c.value);
}
```

# Multitasking e Programmazione Concorrente

Col termine *multitasking* si intende la capacità di eseguire due o più *task* (programmi) contemporaneamente. Ad esempio elaborare un documento mentre un'altro è in stampa.

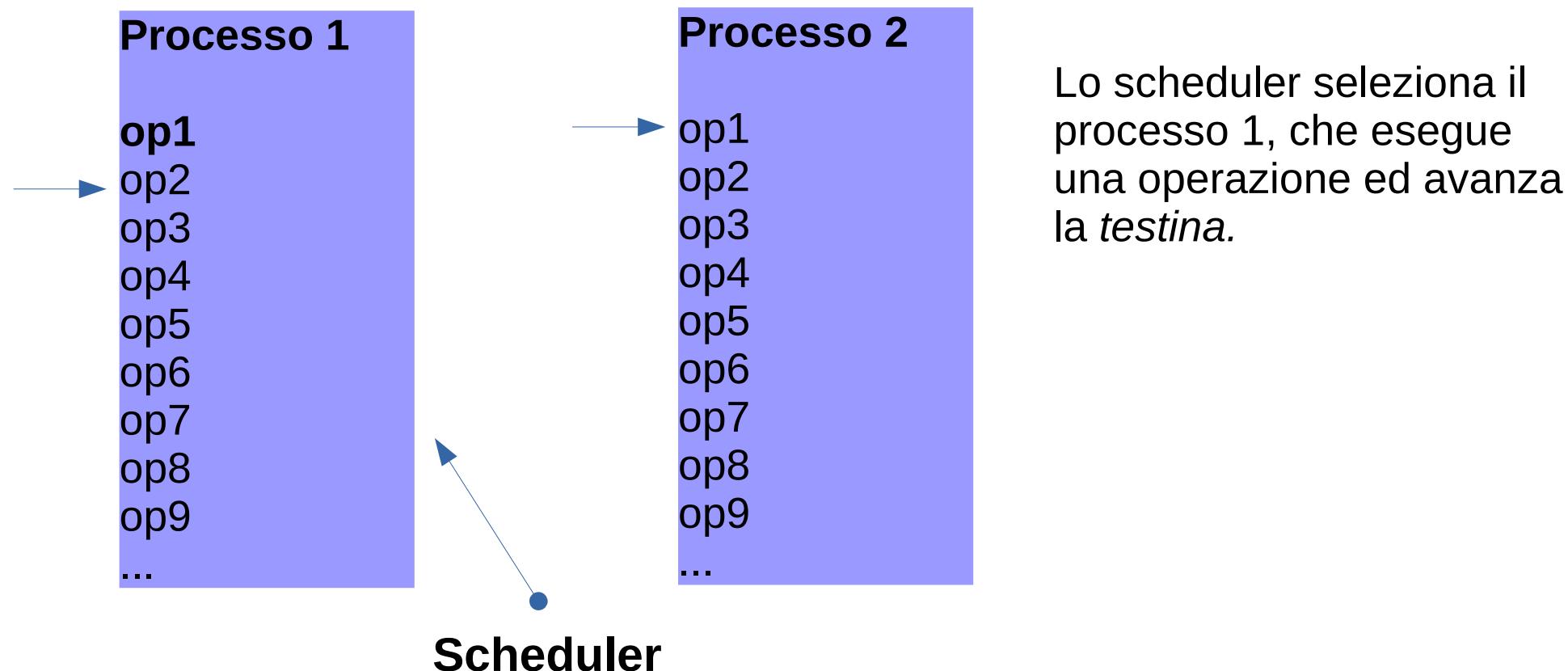
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

L'esecuzione è governata da uno **scheduler**.



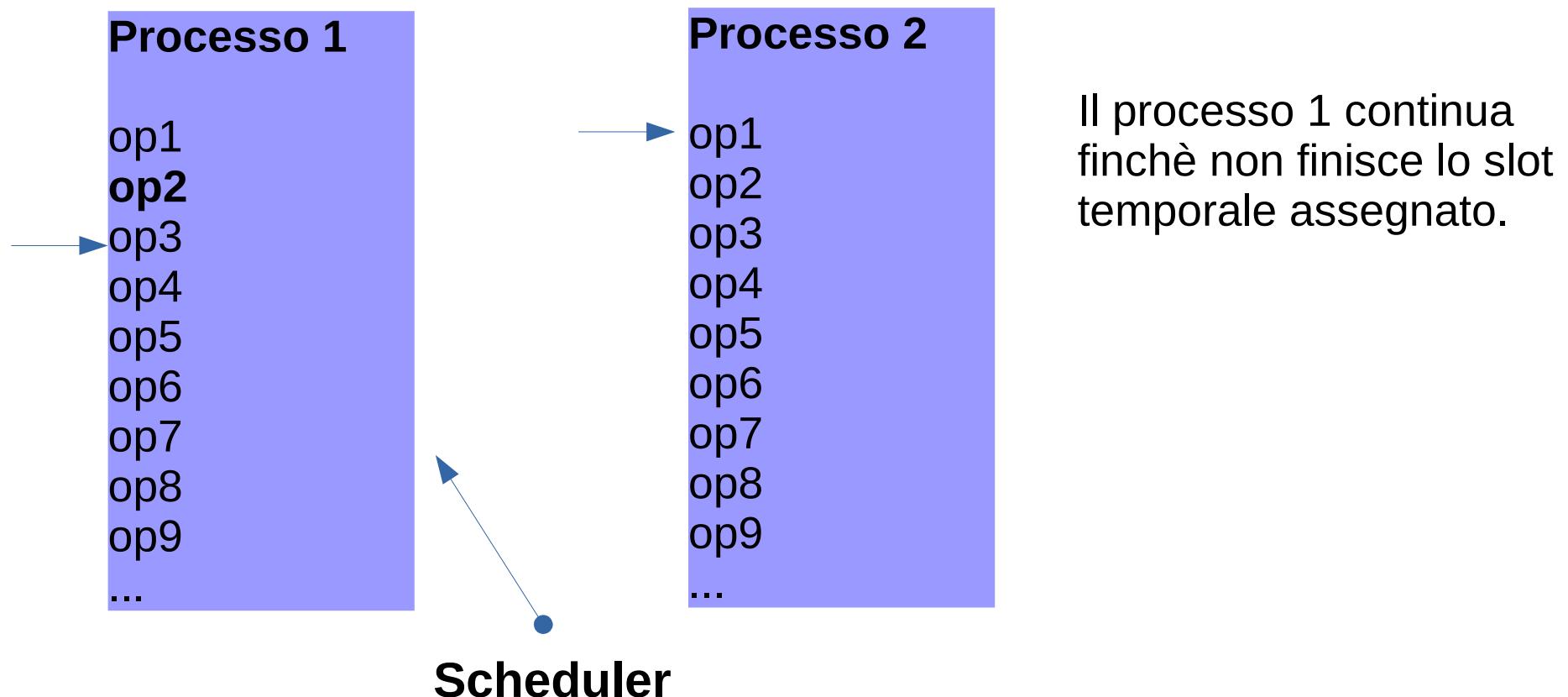
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

L'esecuzione è governata da uno **scheduler**.



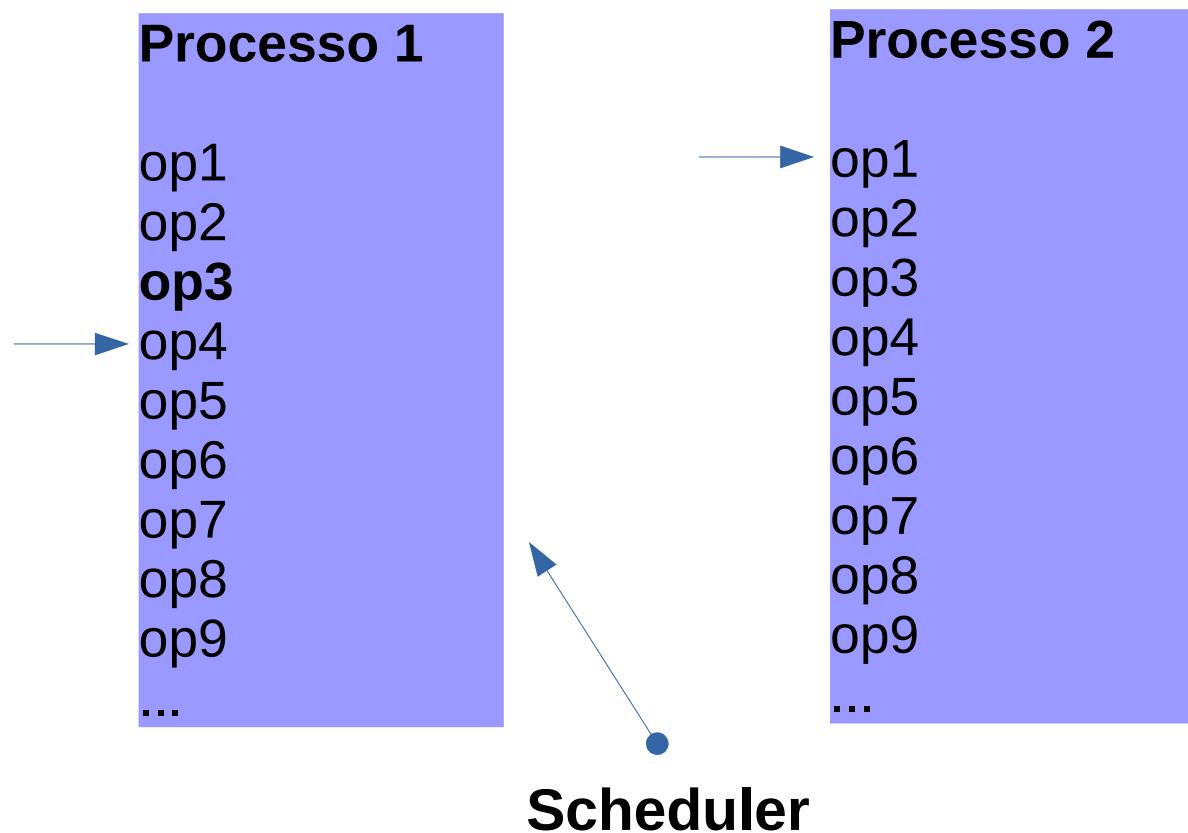
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso il *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

L'esecuzione è governata da uno **scheduler**.



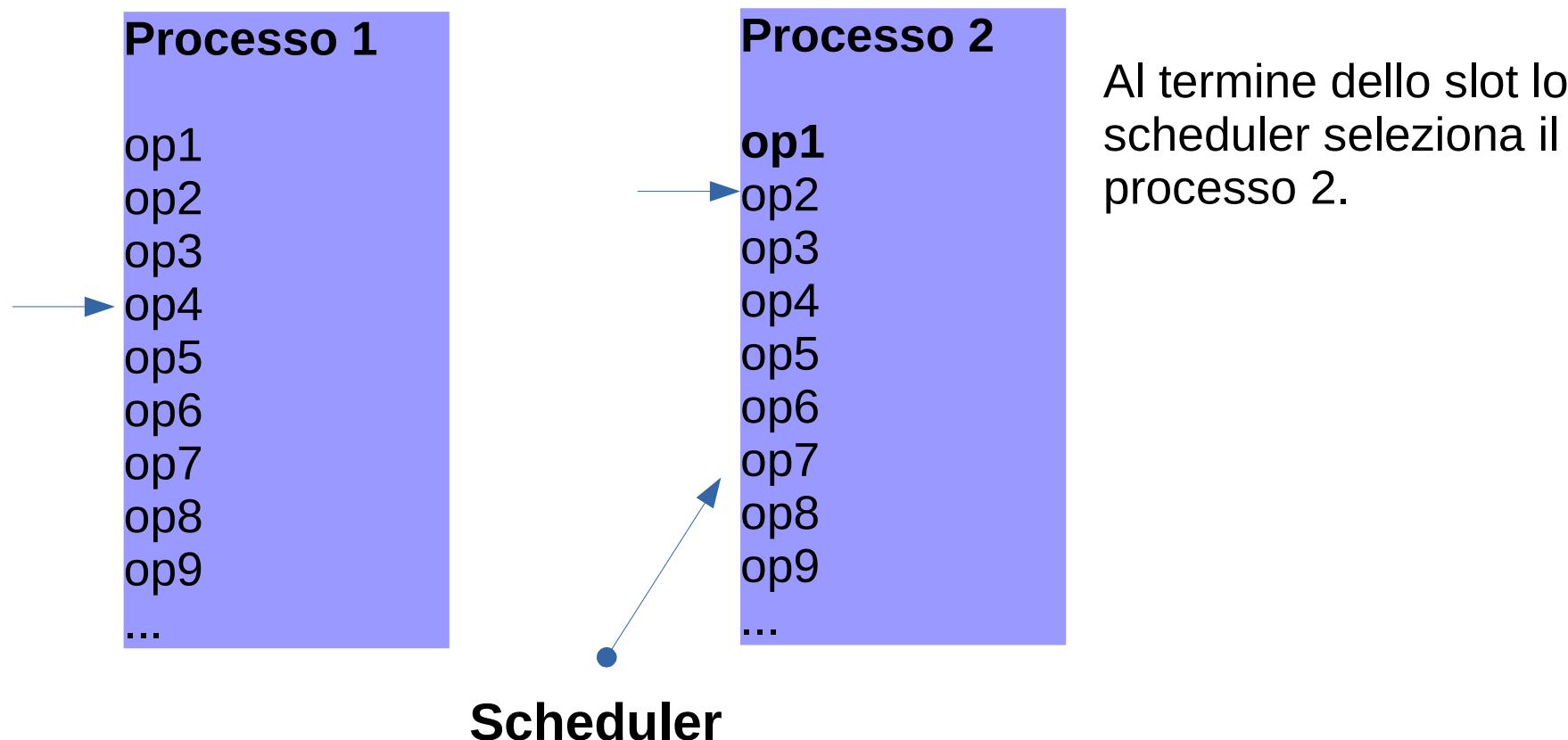
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

L'esecuzione è governata da uno **scheduler**.



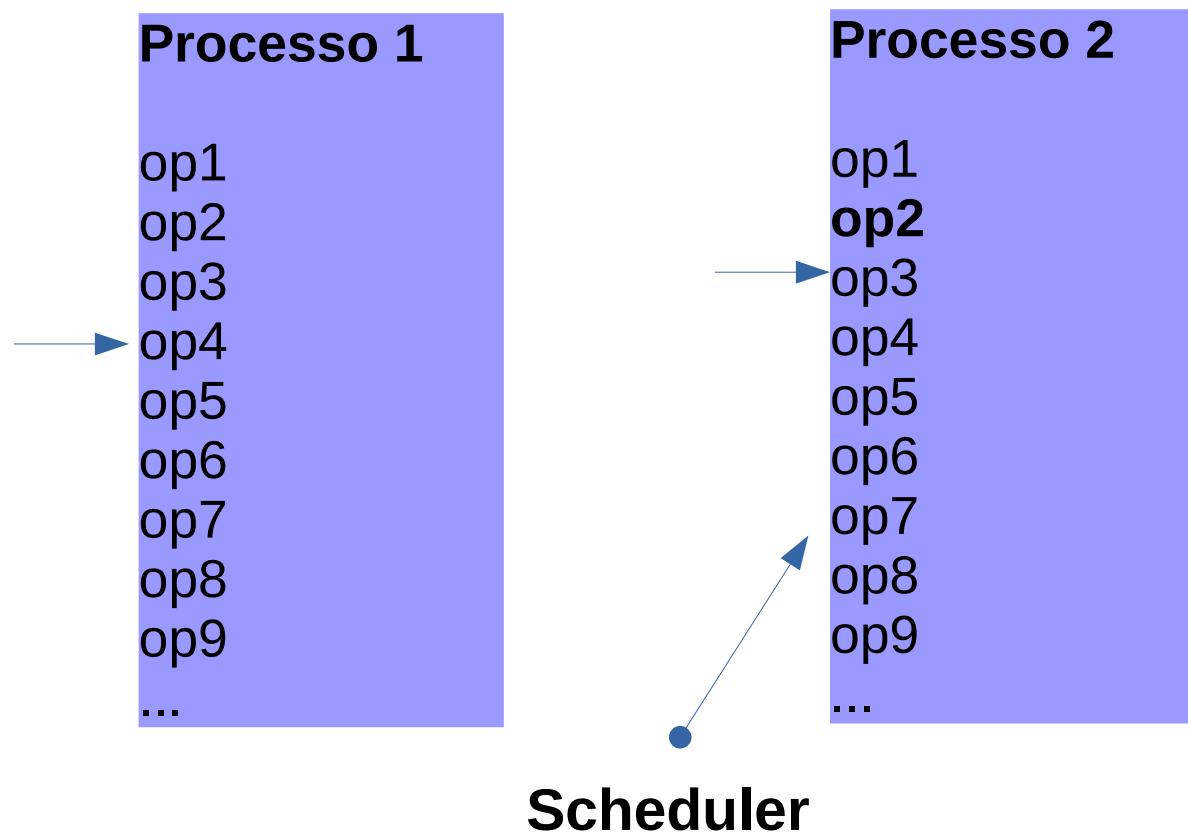
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso il *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

L'esecuzione è governata da uno **scheduler**.



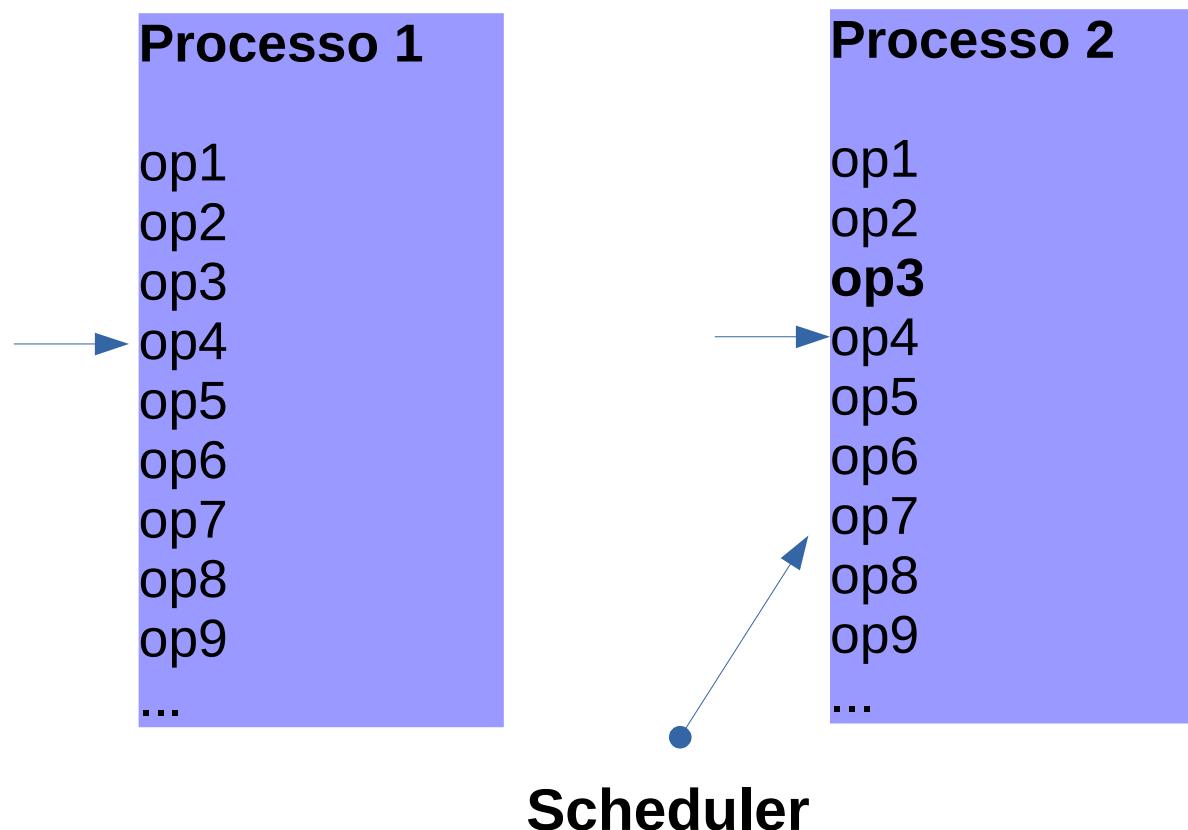
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

L'esecuzione è governata da uno **scheduler**.



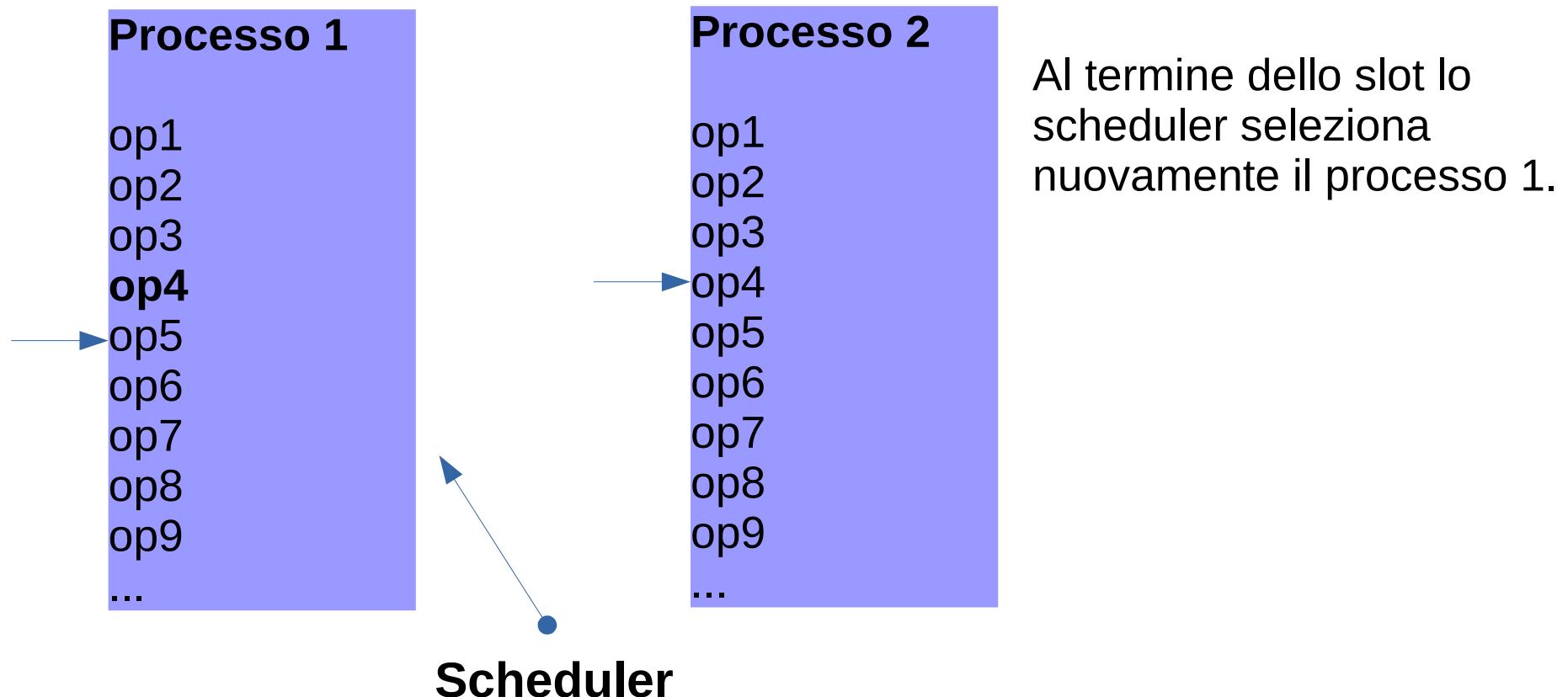
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

L'esecuzione è governata da uno **scheduler**.



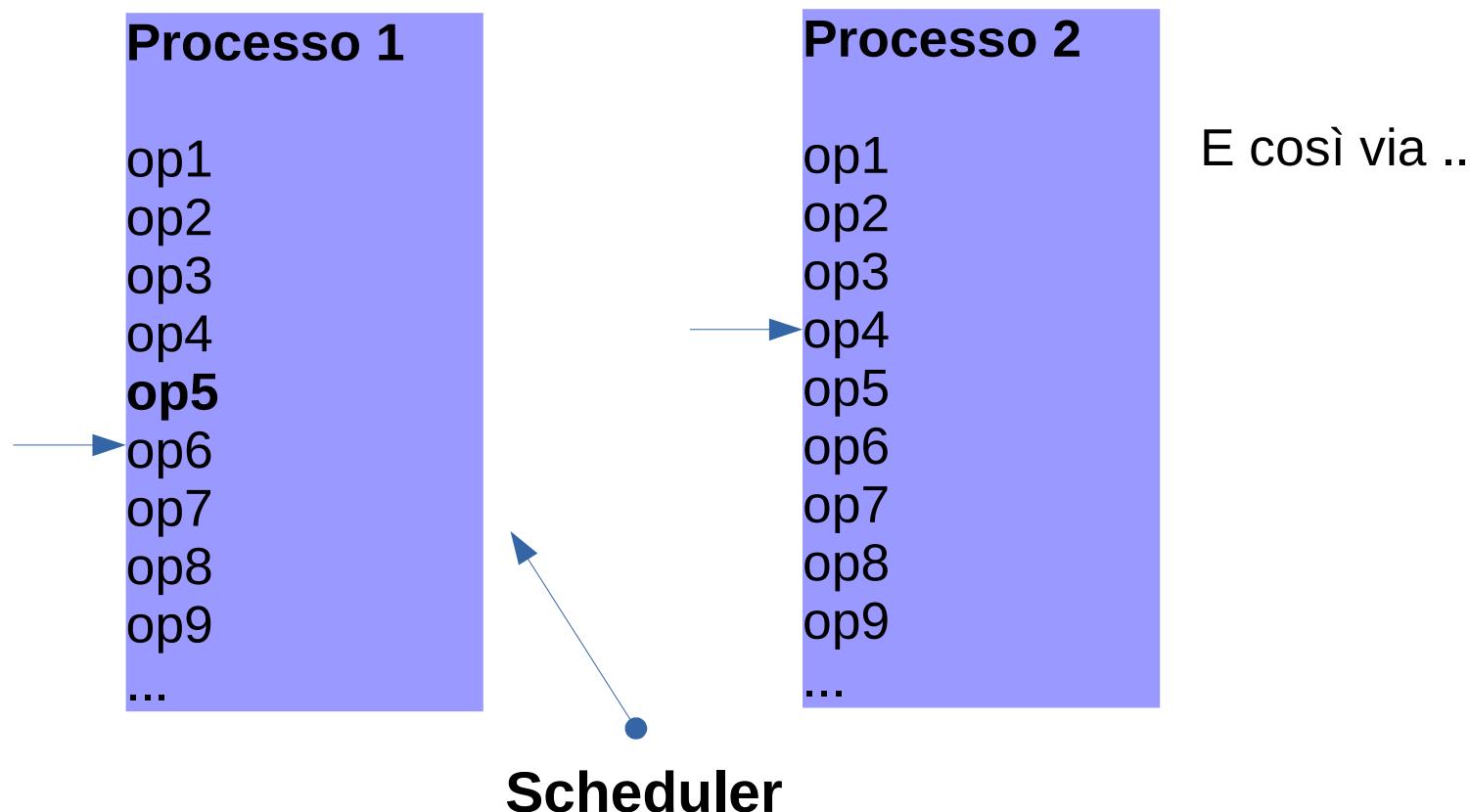
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

L'esecuzione è governata da uno **scheduler**.



Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

L'esecuzione è governata da uno **scheduler**.



Un *Thread* è un processo, figlio di un processo padre, che condivide lo spazio di indirizzamento (le variabili) col padre.

Per creare un thread è necessario fornire una implementazione di *Runnable*, che conterrà il codice eseguito nel thread, ed avviare una istanza di *Thread* che faccia riferimento al *Runnable*.

```
final Counter c = new Counter();
final Runnable incCounter = new Runnable() { // anon class

    @Override
    public void run() {
        c.inc();
    }
}
final Thread t = new Thread(incCounter);
t.start();
t.join(); // wait until the thread t ends
System.out.println(c.getValue());
```

Il multitasking *simulato* permette di ottimizzare l'utilizzo delle risorse. Un processo può utilizzare la CPU mentre gli altri sono in attesa di completare operazioni di IO, ad esempio.

Supponiamo ad esempio di avere un task suddiviso nelle seguenti parti:

- **Lettura** . legge la scheda di una persona dal disco (risorsa utilizzata disco)
- **Elaborazione** . effettua delle elaborazioni su questa scheda (risorsa utilizzata CPU)
- **Stampa** . Stampa I risultati (risorsa utilizzata stampante).

Supponiamo di dover eseguire questo task per un numero elevato di persone, diciamo  $n$ .

Il processamento delle  $n$  schede può essere eseguito da  $n$  processi differenti.

## Vantaggi del Multitasking – esempio leggi/elabora/stampa

Il processamento inizia dalla lettura della scheda **s1** da parte del processo **p1**.

Disco	CPU	Stampante
p1		

**p1**

**leggi s1**

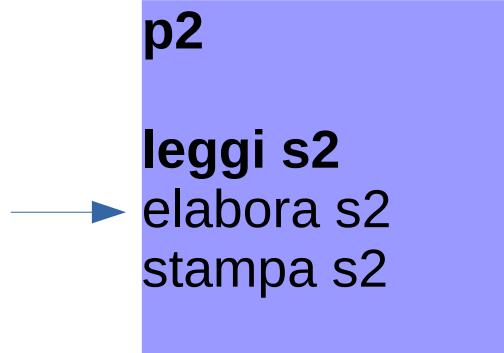
elabora s1  
stampa s1



## Vantaggi del Multitasking – esempio leggi/elabora/stampa

Terminata la lettura di s1 si procede alla sua elaborazione. Nel frattempo il disco è libero e può essere iniziata la lettura di s2.

Disco	CPU	Stampante
p2	p1	



## Vantaggi del Multitasking – esempio leggi/elabora/stampa

Terminata la elaborazione di s1 si può mandarlo in stampa. Nel frattempo può essere avviata l'elaborazione di s2 e la lettura di s3.

Disco	CPU	Stampante
p3	p2	p1

**p1**  
leggi s1  
elabora s1  
**stampa s1**

**p2**  
leggi s2  
**elabora s2**  
stampa s2

**p3**  
**leggi s3**  
elabora s3  
stampa s3



## Vantaggi del Multitasking – esempio leggi/elabora/stampa

Terminata la stampa di s1 il processo p1 conclude la sua esecuzione, gli altri proseguono e nuovi processi entreranno in gioco per s4, s5, ..., sn.

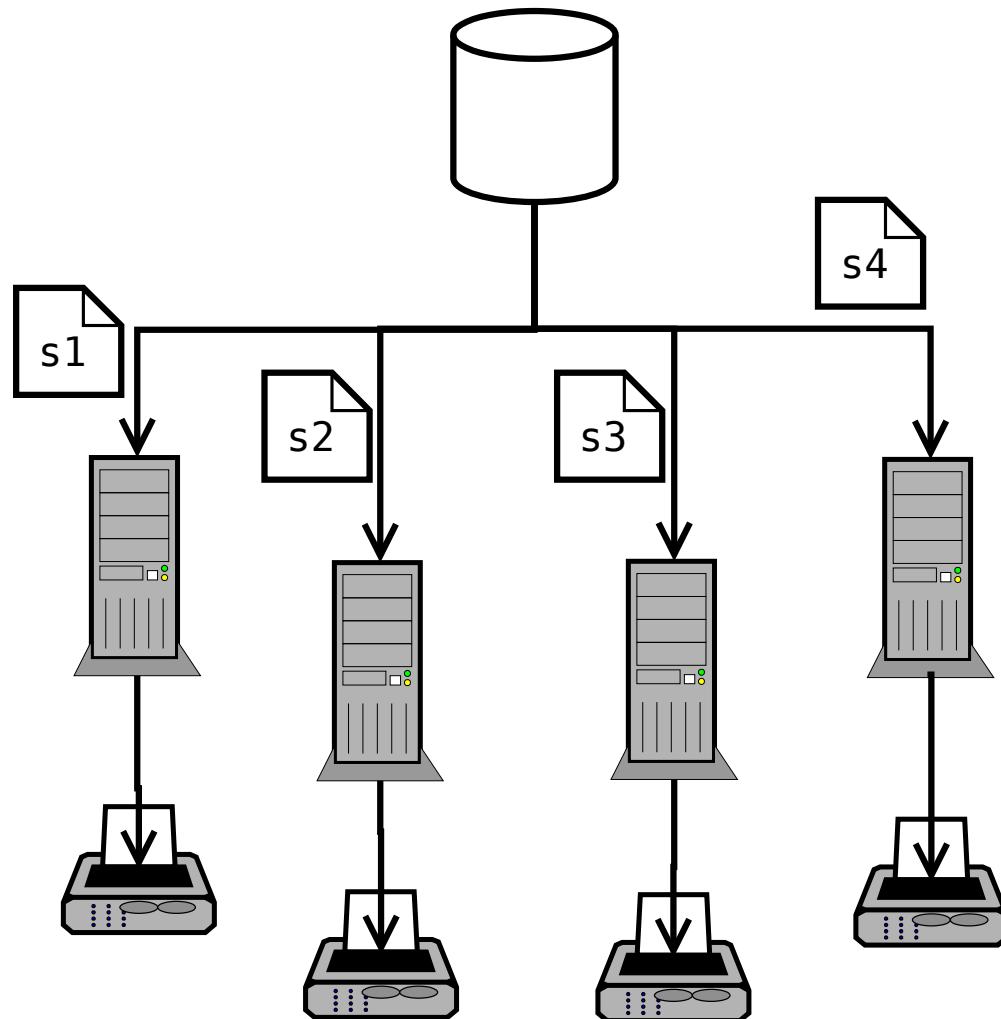
Disco	CPU	Stampante
p4	p3	p2

**p2**  
leggi s2  
elabora s2  
**stampa s2**

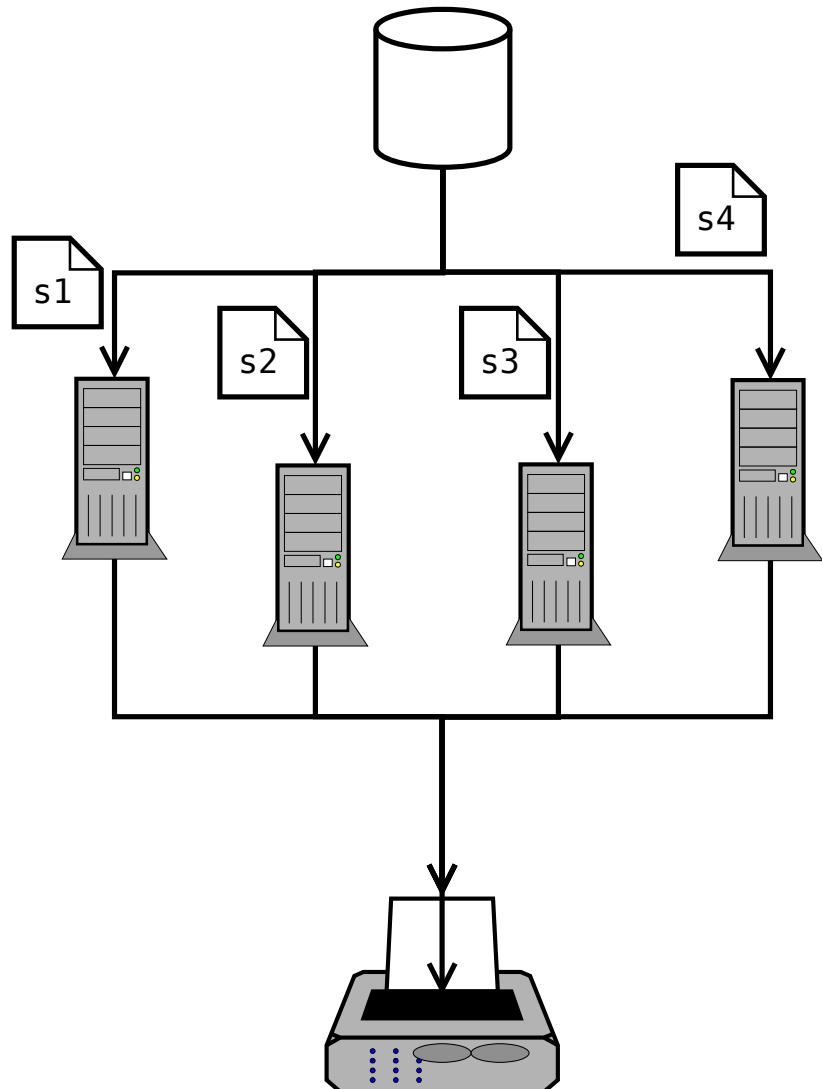
**p3**  
leggi s3  
**elabora s3**  
stampa s3

→  
**p4**  
**leggi s4**  
elabora s4  
stampa s4

In presenza di architetture multiprocessore (ad esempio GUP) o distribuite (Cloud, grid, ...) è fisicamente possibile eseguire diversi processi contemporaneamente.



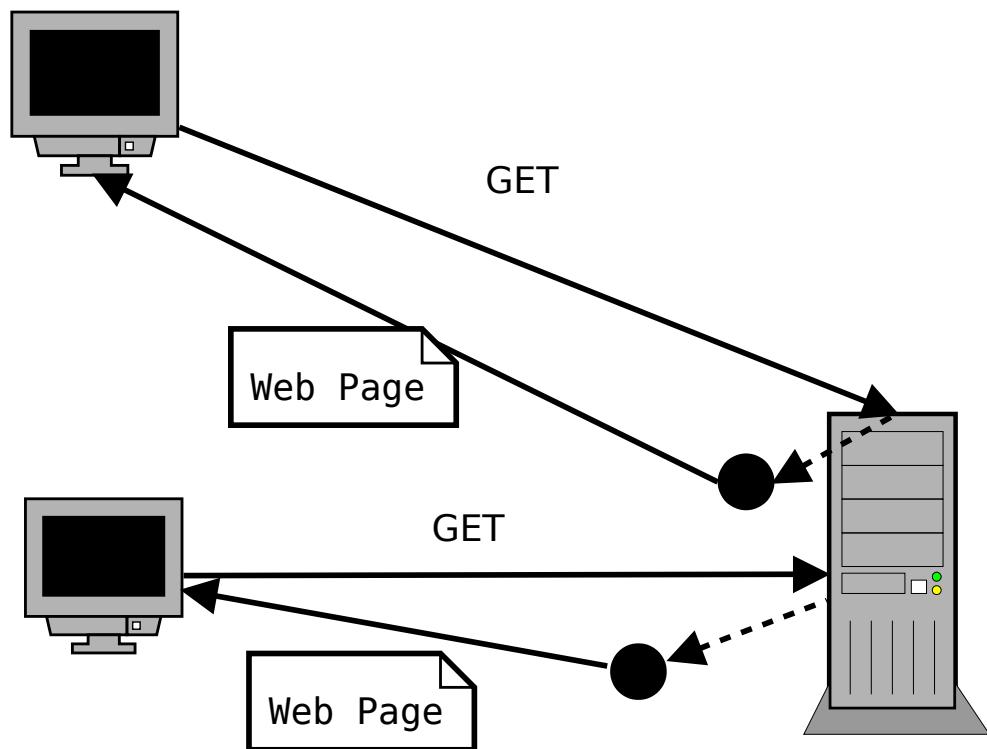
Sia in caso di multitasking effettivo che simulato, l'accesso a risorse condivise può essere problematico. Ad esempio, se si usa un unico terminale di stampa l'output dei vari processi potrebbe accavallarsi.



```
s1 year of birth 1978
s1 age 38
s2 year of birth 1979
s2 age 37
s3 year of birth 1980
s4 year of birth 1981
s3 age 36
s4 age 35
```

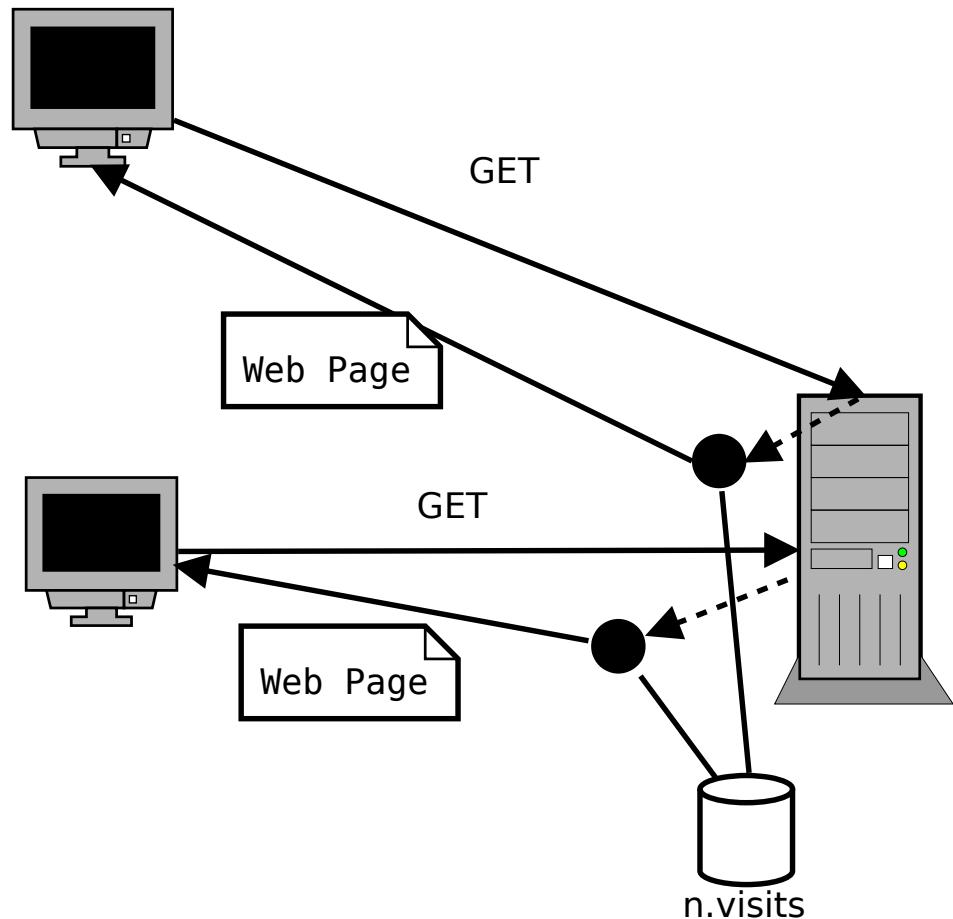
## Accesso Concorrente – Esempio : Contatore di Accessi

Un server web serve ogni richiesta su un thread dedicato.



## Accesso Concorrente – Esempio : Contatore di Accessi

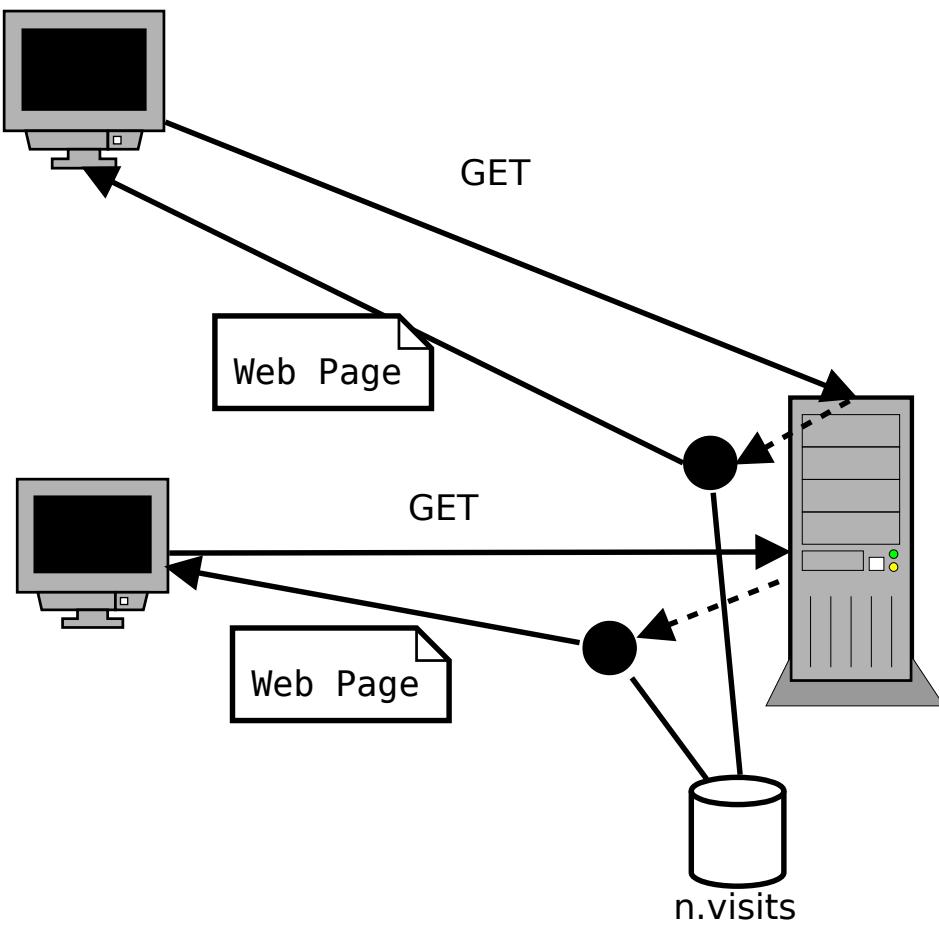
Un server web serve ogni richiesta su un thread dedicato. Supponiamo che per ogni richiesta si voglia aumentare il valore di un contatore salvato su uno storage.



## Accesso Concorrente – Esempio : Contatore di Accessi

Un server web serve ogni richiesta su un thread dedicato. Supponiamo che per ogni richiesta si voglia aumentare il valore di un contatore salvato su uno storage.

Ogni thread di servizio legge il contatore, lo incrementa e salva il valore incrementato. Supponiamo che due richieste siano servite simultaneamente.



leggi c  
incrementa c  
scrivi c

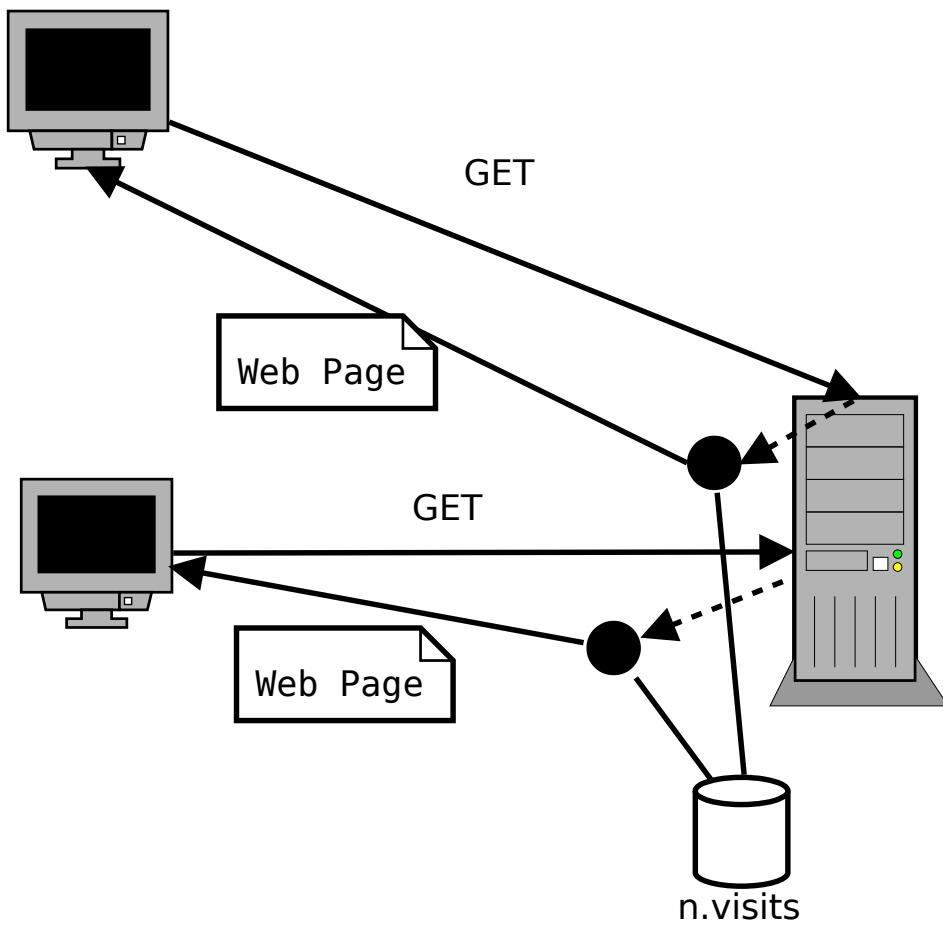
leggi c  
incrementa c  
scrivi c

**Contatore su file = n**

## Accesso Concorrente – Esempio : Contatore di Accessi

Un server web serve ogni richiesta su un thread dedicato. Supponiamo che per ogni richiesta si voglia aumentare il valore di un contatore salvato su uno storage.

Ogni thread di servizio legge il contatore, lo incrementa e salva il valore incrementato. Supponiamo che due richieste siano servite simultaneamente.



leggi c  
incrementa c  
scrivi c  
  
c=n

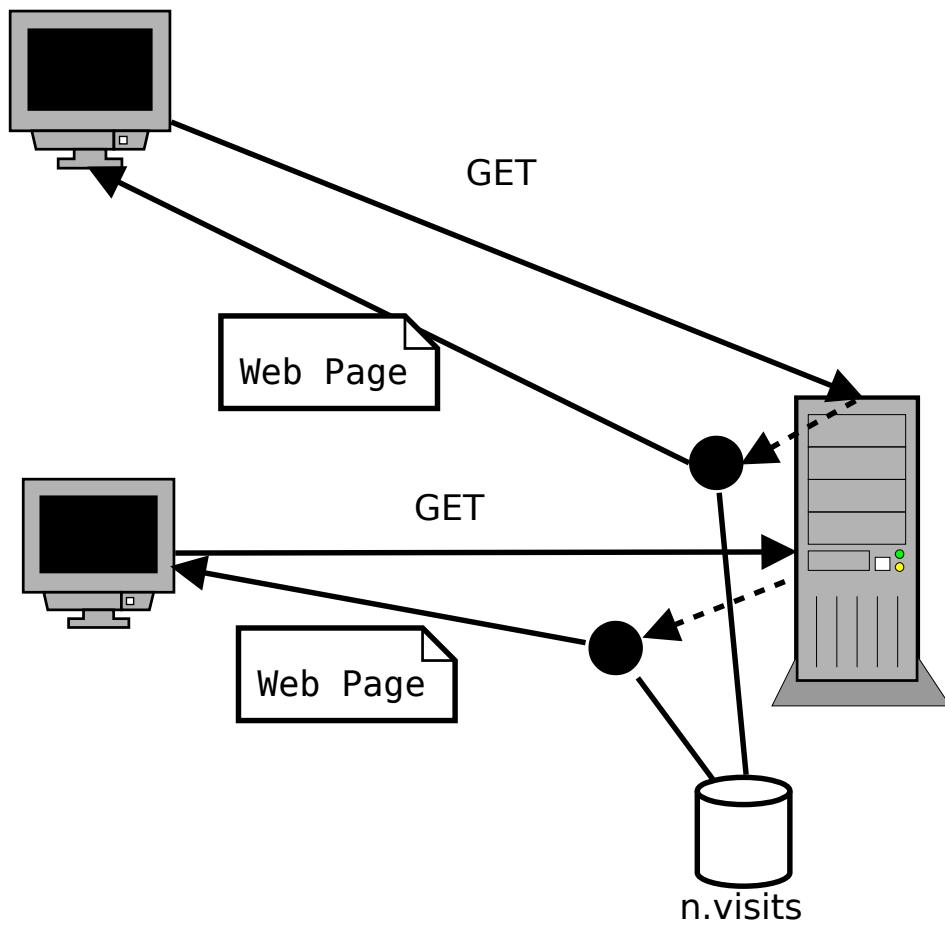
leggi c  
incrementa c  
scrivi c

**Contatore su file = n**

## Accesso Concorrente – Esempio : Contatore di Accessi

Un server web serve ogni richiesta su un thread dedicato. Supponiamo che per ogni richiesta si voglia aumentare il valore di un contatore salvato su uno storage.

Ogni thread di servizio legge il contatore, lo incrementa e salva il valore incrementato. Supponiamo che due richieste siano servite simultaneamente.

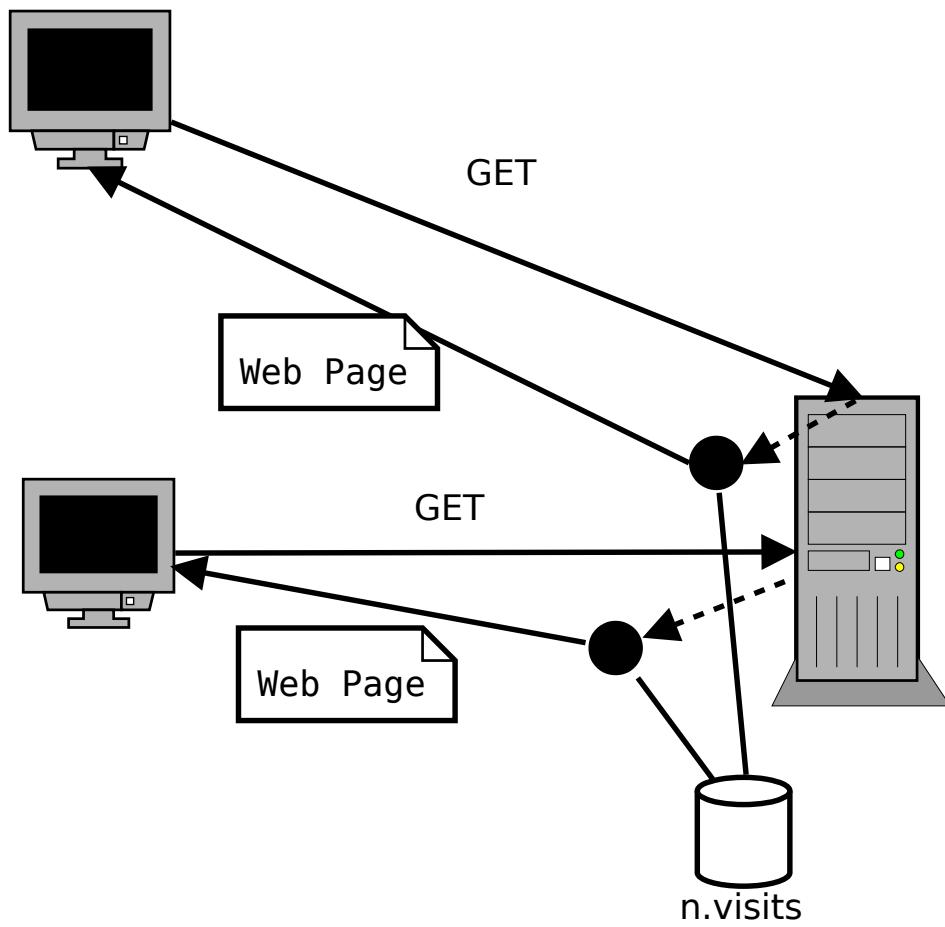


**Contatore su file = n**

## Accesso Concorrente – Esempio : Contatore di Accessi

Un server web serve ogni richiesta su un thread dedicato. Supponiamo che per ogni richiesta si voglia aumentare il valore di un contatore salvato su uno storage.

Ogni thread di servizio legge il contatore, lo incrementa e salva il valore incrementato. Supponiamo che due richieste siano servite simultaneamente.



leggi c  
**incrementa c**  
scrivi c  
 $c=n+1$

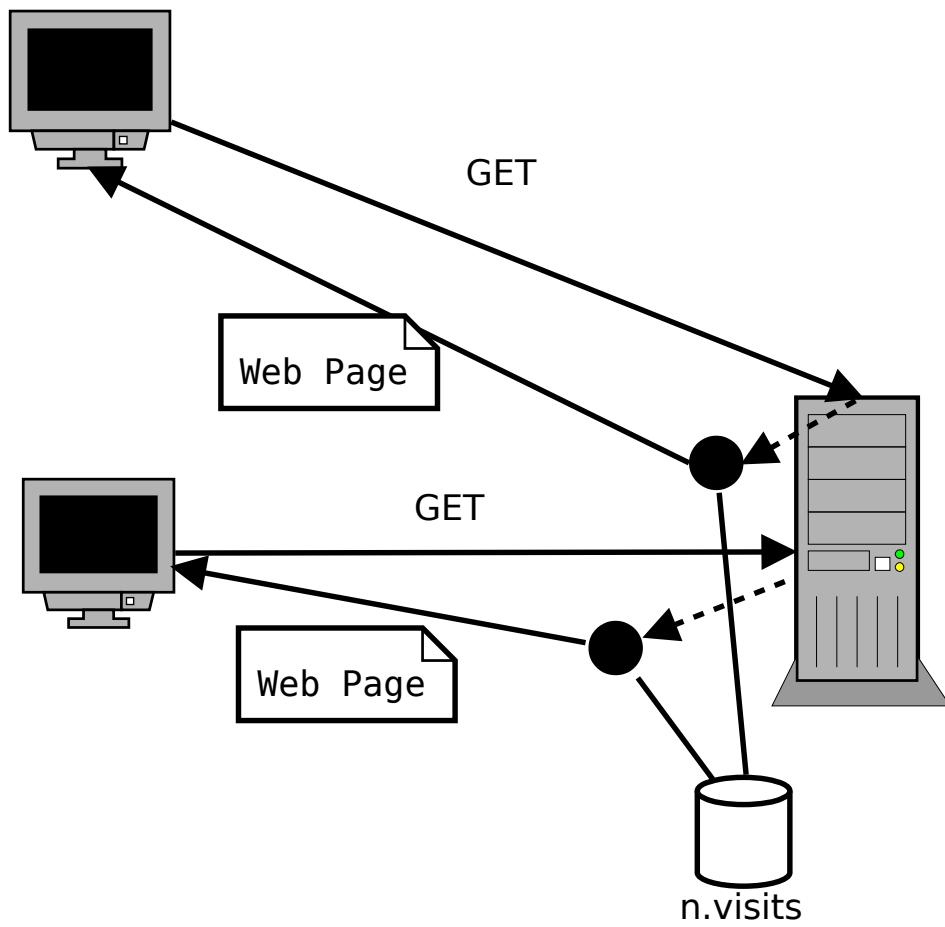
leggi c  
incrementa c  
scrivi c  
 $c=n$

**Contatore su file = n**

## Accesso Concorrente – Esempio : Contatore di Accessi

Un server web serve ogni richiesta su un thread dedicato. Supponiamo che per ogni richiesta si voglia aumentare il valore di un contatore salvato su uno storage.

Ogni thread di servizio legge il contatore, lo incrementa e salva il valore incrementato. Supponiamo che due richieste siano servite simultaneamente.



leggi c  
incrementa c  
scrivi c  
 $c=n+1$

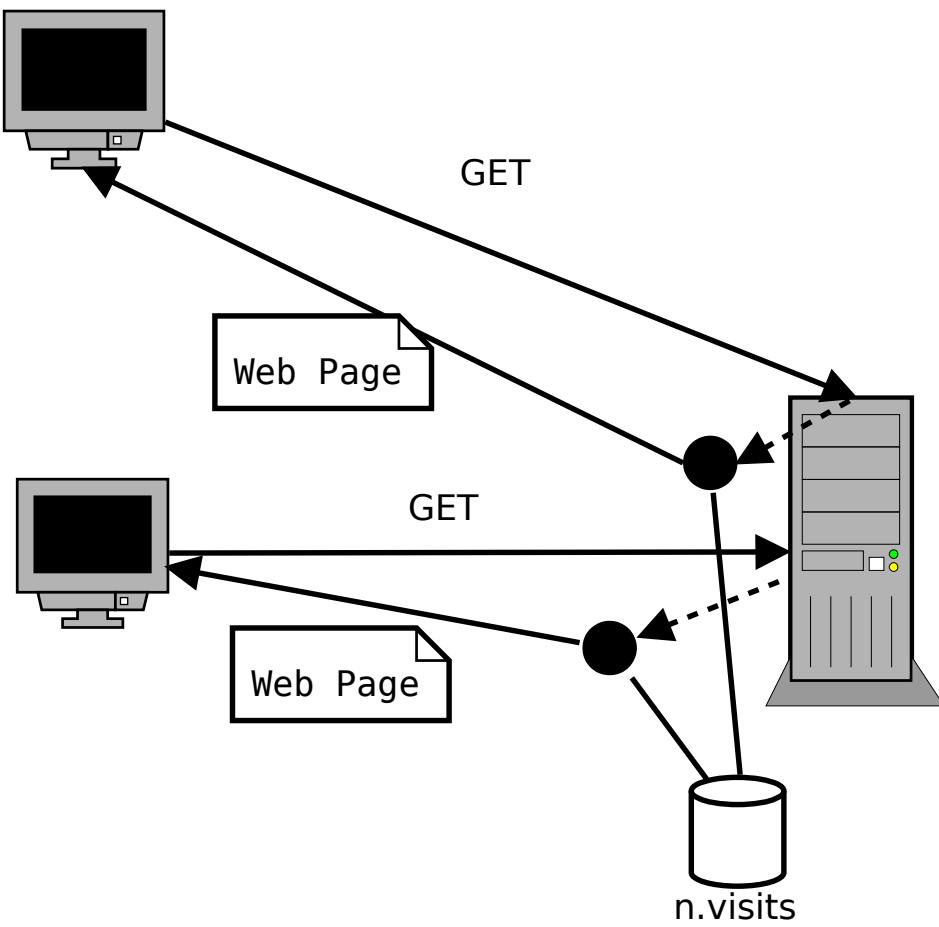
leggi c  
**incrementa c**  
scrivi c  
 $c=n+1$

**Contatore su file = n**

## Accesso Concorrente – Esempio : Contatore di Accessi

Un server web serve ogni richiesta su un thread dedicato. Supponiamo che per ogni richiesta si voglia aumentare il valore di un contatore salvato su uno storage.

Ogni thread di servizio legge il contatore, lo incrementa e salva il valore incrementato. Supponiamo che due richieste siano servite simultaneamente.

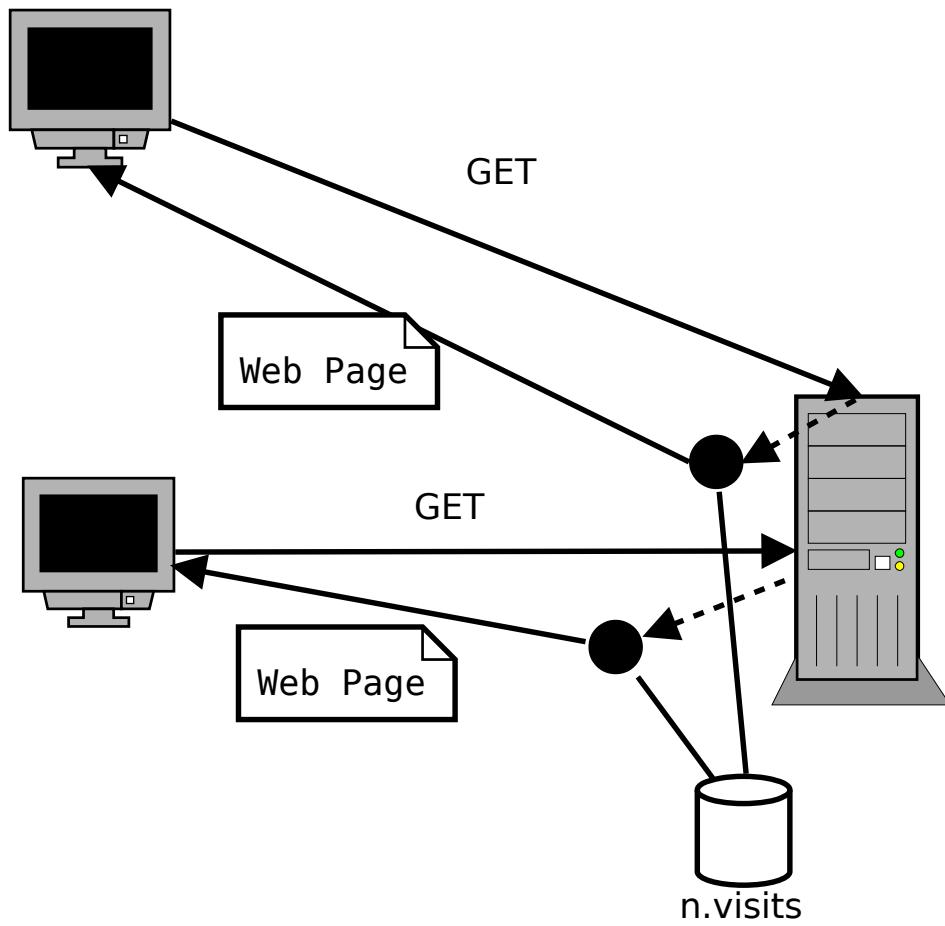


**Contatore su file =  $n + 1$**

## Accesso Concorrente – Esempio : Contatore di Accessi

Un server web serve ogni richiesta su un thread dedicato. Supponiamo che per ogni richiesta si voglia aumentare il valore di un contatore salvato su uno storage.

Ogni thread di servizio legge il contatore, lo incrementa e salva il valore incrementato. Supponiamo che due richieste siano servite simultaneamente.



leggi c  
incrementa c  
scrivi c  
 $c=n+1$

leggi c  
incrementa c  
**scrivi c**  
 $c=n+1$

**Contatore su file = n + 1**

I problemi di accesso concorrente alle risorse non possono essere risolti a livello applicativo ma richiedono che vengano forniti strumenti appropriati a livello di scheduler: lock, semafori, monitor, ...

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.



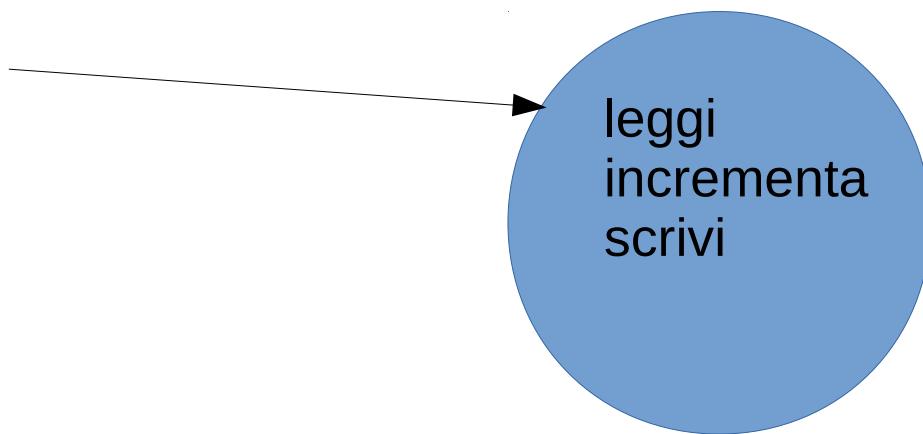
**Contatore su file = n**

Vedi [https://en.wikipedia.org/wiki/Monitor\\_\(synchronization\)](https://en.wikipedia.org/wiki/Monitor_(synchronization)) .

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.

Arriva la prima richiesta. Il thread corrispondente acquisisce il monitor perchè lo trova libero.

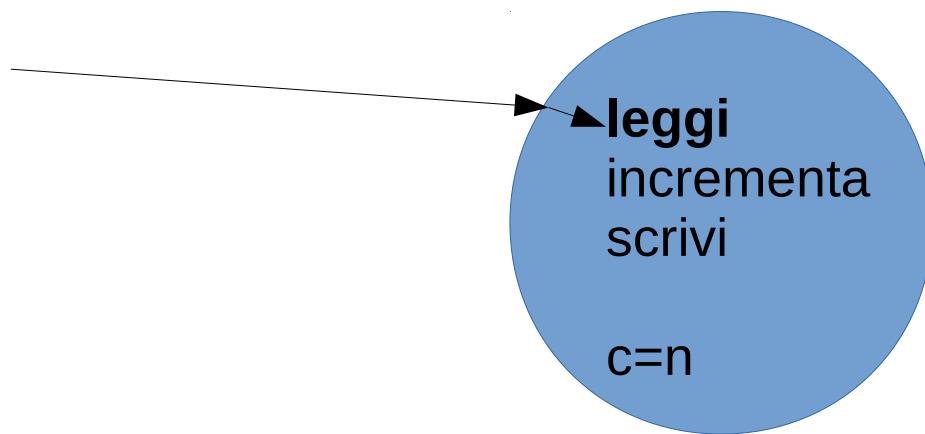


**Contatore su file = n**

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.

Arriva la prima richiesta. Il thread corrispondente acquisisce il monitor perchè lo trova libero. Inizia quindi ad eseguire.

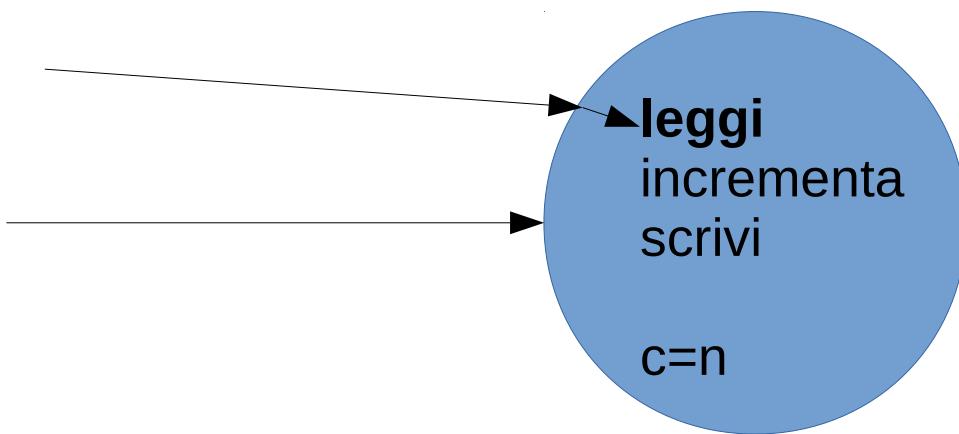


**Contatore su file = n**

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.

Arriva la seconda richiesta. Il thread corrispondente trova il monitor occupato e quindi viene sospeso.

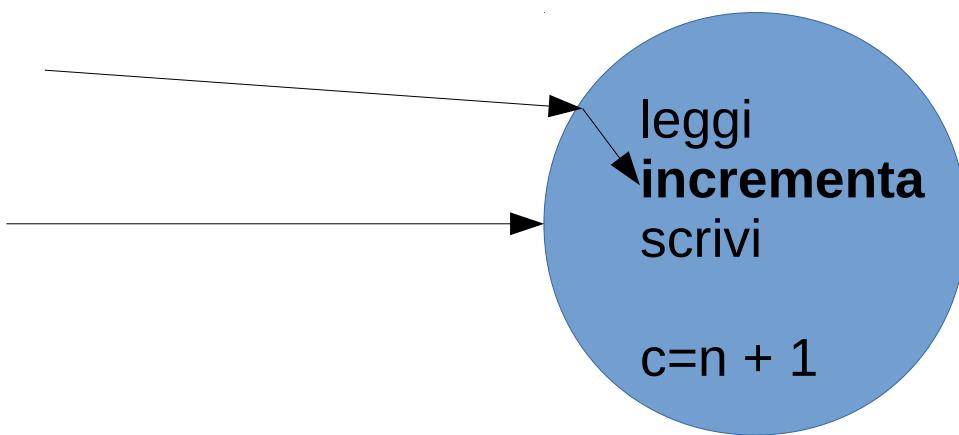


**Contatore su file = n**

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.

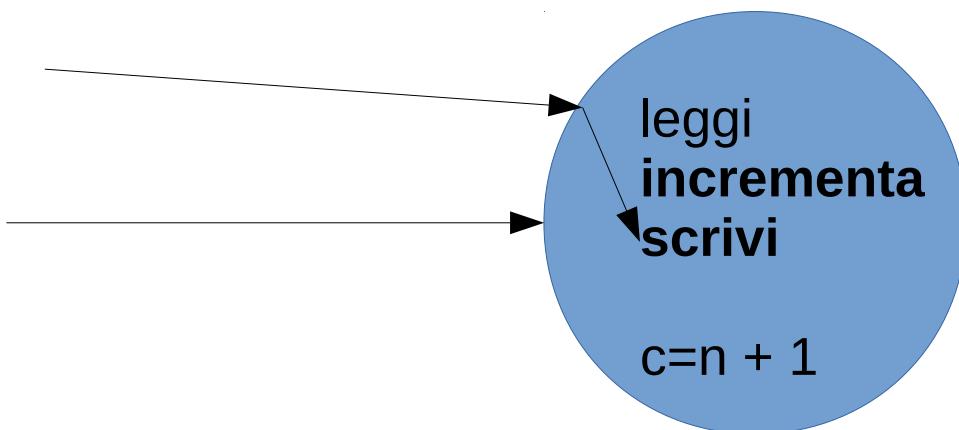
L'esecuzione della richiesta del primo thread continua.



**Contatore su file = n**

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

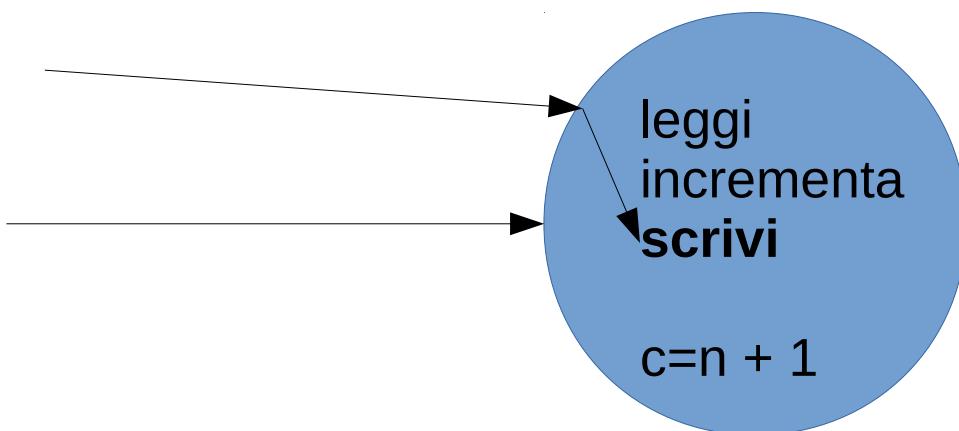
Esempio: contatore di accessi. Il contatore è implementato con un monitor.



**Contatore su file = n + 1**

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.



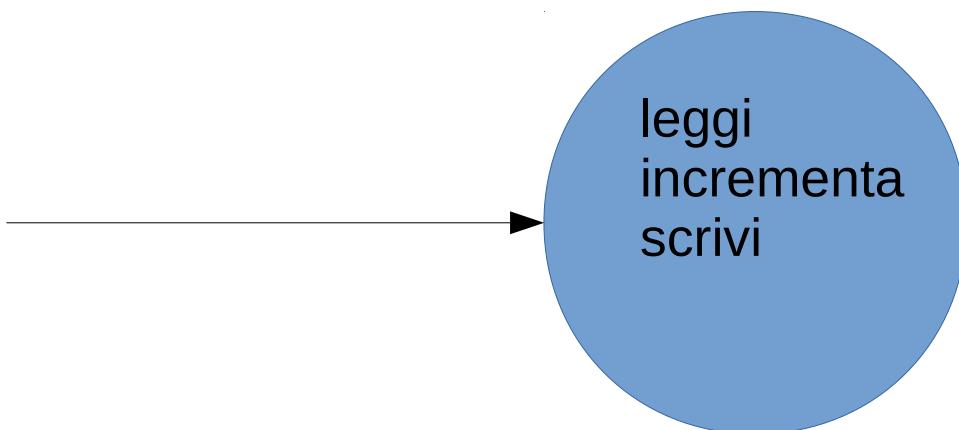
**Contatore su file = n + 1**

Vedi [https://en.wikipedia.org/wiki/Monitor\\_\(synchronization\)](https://en.wikipedia.org/wiki/Monitor_(synchronization)) .

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.

L'esecuzione della richiesta del primo thread termina. Ora il secondo thread può entrare nel monitor.

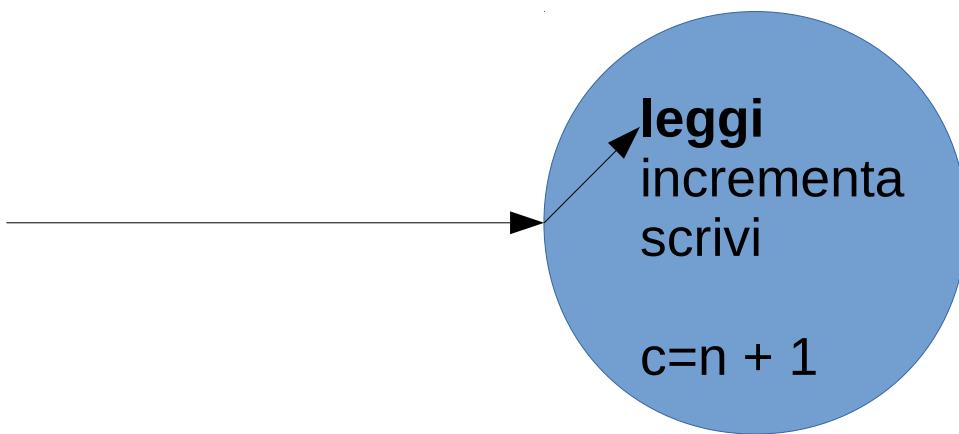


**Contatore su file = n + 1**

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.

La richiesta del secondo thread verrà portata a compimento.

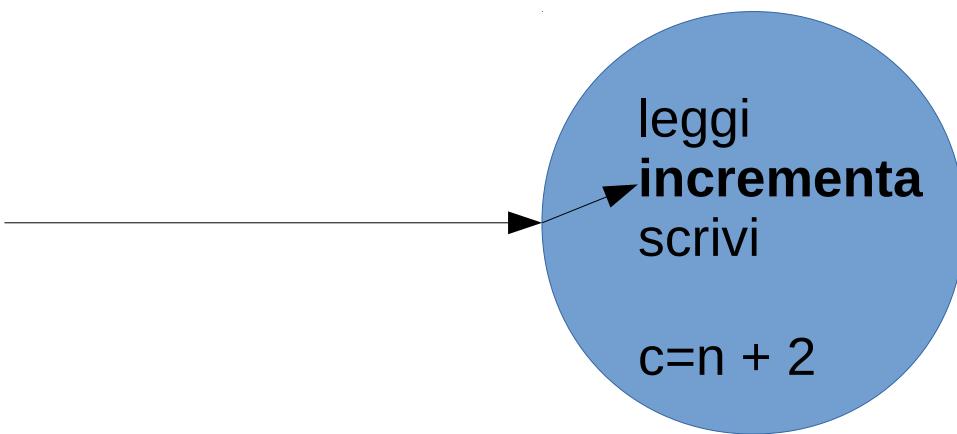


**Contatore su file = n + 1**

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.

La richiesta del secondo thread verrà portata a compimento.

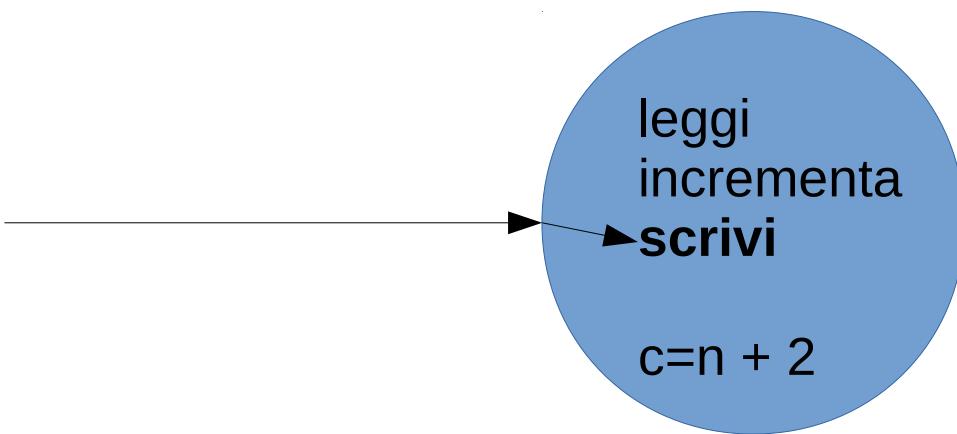


**Contatore su file = n + 1**

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.

La richiesta del secondo thread verrà portata a compimento.



**Contatore su file = n + 2**

Vedi [https://en.wikipedia.org/wiki/Monitor\\_\(synchronization\)](https://en.wikipedia.org/wiki/Monitor_(synchronization)) .

Un *monitor* è un oggetto al quale solo un thread per volta può accedere.

Esempio: contatore di accessi. Il contatore è implementato con un monitor.



**Contatore su file = n + 2**

Vedi [https://en.wikipedia.org/wiki/Monitor\\_\(synchronization\)](https://en.wikipedia.org/wiki/Monitor_(synchronization)) .

In Java ogni oggetto è fornito di un lock implicito, che si acquisisce entrando in un blocco *synchronized*

```
final Counter c = new Counter();
final Runnable incCounter = new Runnable() { // anon class

    @Override
    public void run() {
        synchronized(c) { //acquisisce il lock su c
            c.inc();
        }
    }
};
```

Anche i metodi possono essere dichiarati synchronized. In questo caso il lock è quello relativo all'istanza e riguarda tutto il corpo del metodo.

```
class Counter{  
...  
    public synchronized inc() {  
        //do something  
    }  
}
```

## Sospendere l'esecuzione dei Thread

Quando un thread detiene il lock per un qualche oggetto o può sospendere la propria esecuzione con

- `o.wait()` sospende l'esecuzione del thread corrente a tempo indefinito
- `o.wait(timeInMillis)` sospende l'esecuzione del thread corrente per *timeInMillis* e poi tenta di riacquisire il lock.

```
class Consumer implements Runnable{
    private final Queue buffer;
...
    while(true) {
        if (this.buffer.isEmpty())
            synchronized(this.buffer) {
                try {
                    this.buffer.wait();
                } catch (final InterruptedException e) {
                    System.out.println("Interrupted "+e.getMessage());
                }
            } else
                consume(this.buffer.poll());
    }
}
```

Un thread la cui esecuzione è sospesa su un determinato monitor o può essere riattivato da un altro thread nei seguenti modi:

- `o.notify()` risveglia un unico thread sospeso su `o`, scelto casualmente
- `o.notifyAll()` risveglia tutti i thread sospesi su `o`.

Un thread la cui esecuzione è sospesa su un determinato monitor o può essere riattivato da un altro thread nei seguenti modi:

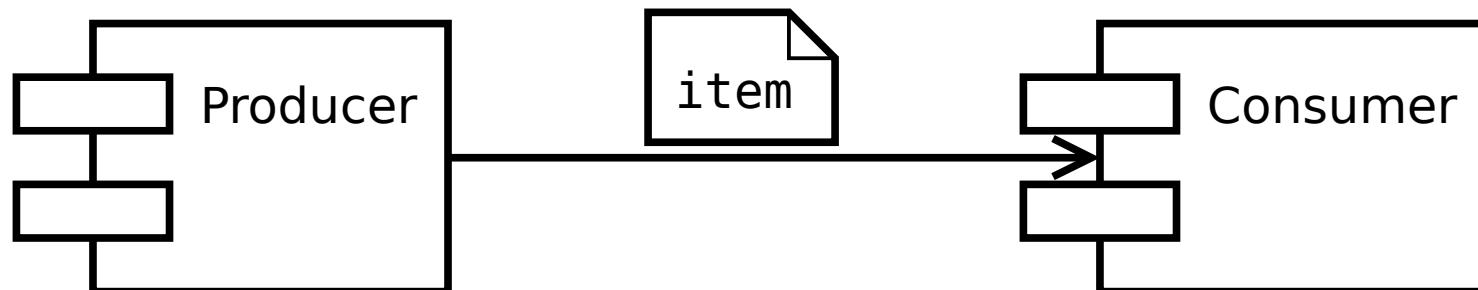
- `o.notify()` risveglia un unico thread sospeso su `o`, scelto casualmente
- `o.notifyAll()` risveglia tutti i thread sospesi su `o`.

Le chiamate `o.notify()` e `o.notifyAll()` possono essere eseguite solo se si detiene il lock su `o`.

Il thread *notificato* per continuare la sua esecuzione deve riottenere il lock su `o`

Il problema *produttore-consumatore* è un esempio classico di problema di sincronizzazione di processi.

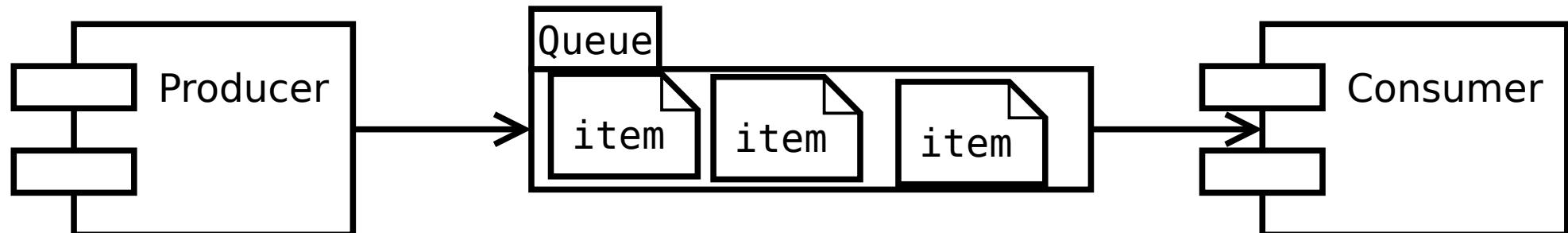
Il produttore *produce* degli elementi e li passa al *consumatore* per farci qualcosa.



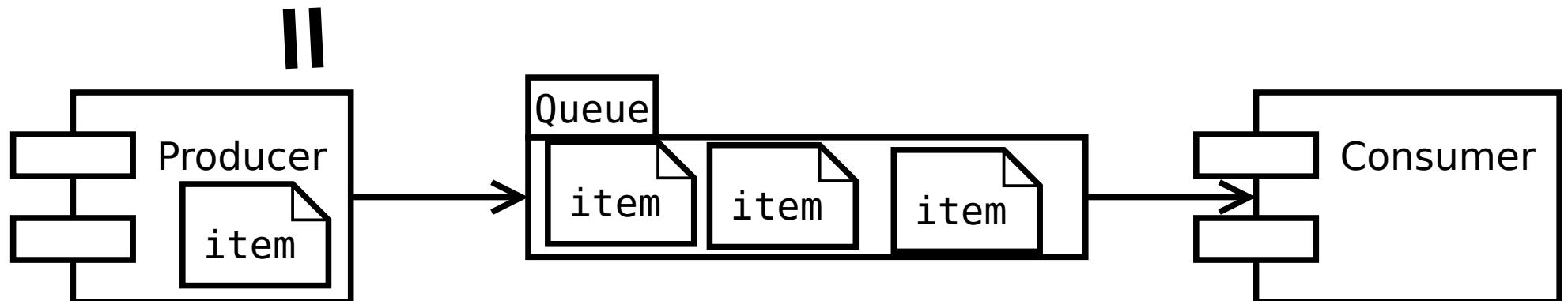
Il problema *produttore-consumatore* è un esempio classico di problema di sincronizzazione di processi.

Il produttore *produce* degli elementi e li passa al *consumatore* per farci qualcosa.

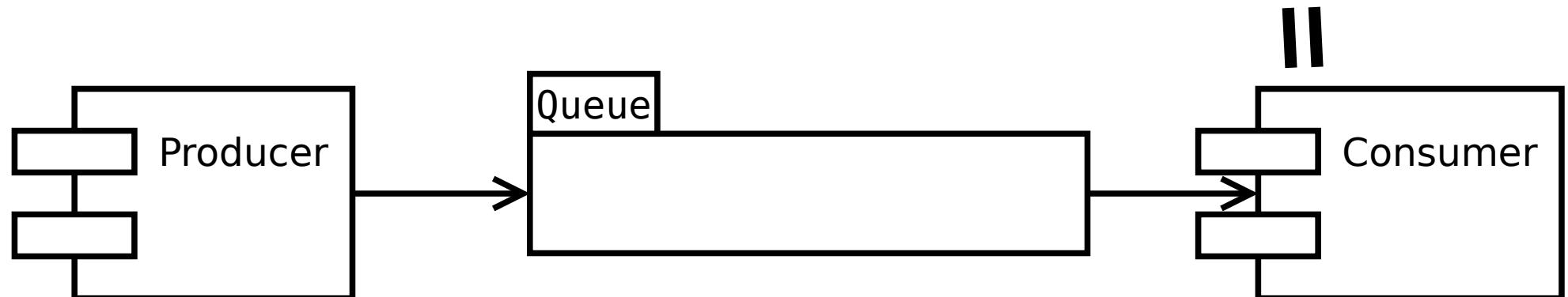
Produttore e consumatore hanno *tempi* diversi. Per ottimizzare vengono eseguiti su thread separati e gli elementi vengono scambiati su una coda.



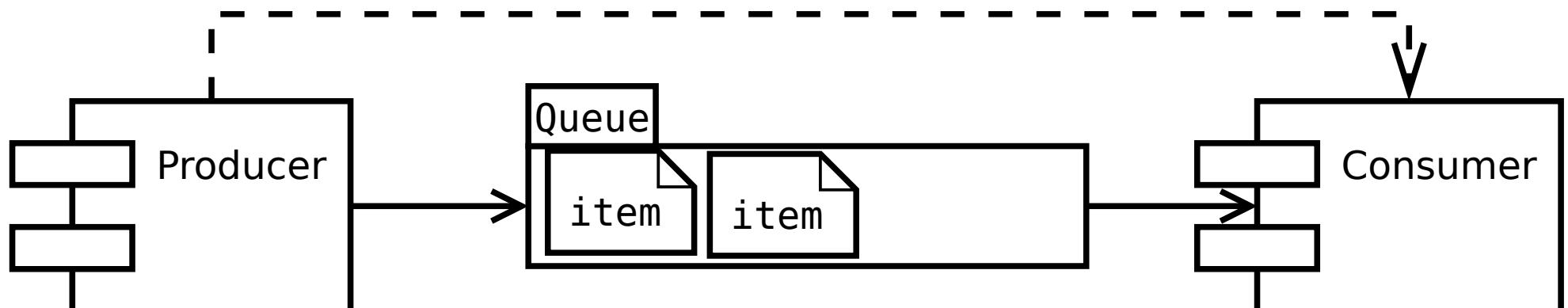
Quando trova la coda piena, il produttore si sospende in attesa che si liberi uno spazio.



Analogamente, se la coda si svuota è il consumatore ad andare in pausa.



Finchè il produttore non inserisce nuovi elementi in coda e sveglia il consumatore che può riprendere a lavorare.

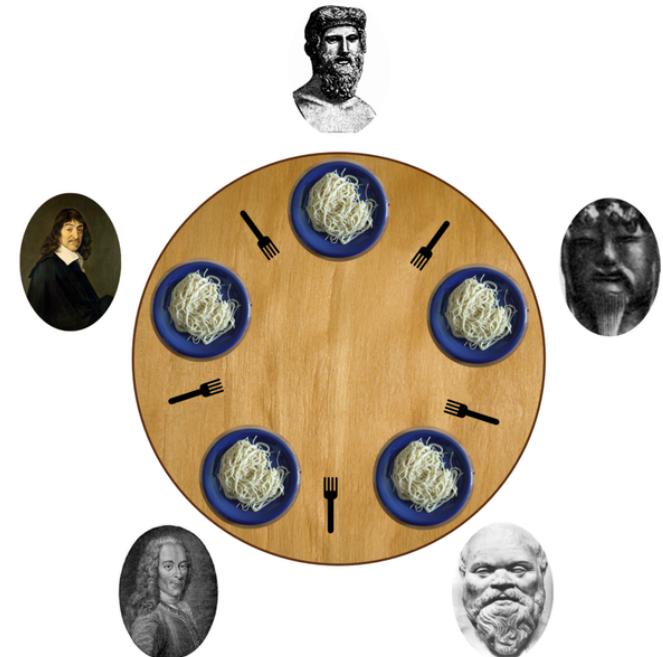


Un deadlock si verifica quando due o più processi sono bloccati per sempre in attesa l'uno dell'altro.

### Esempio: I Filosofi a Cena

Cinque filosofi sono seduti a tavola, un piatto di spaghetti e posto di fronte ad ogni filosofo e tra ogni piatto di spaghetti è posta una forchetta, per un totale di 5 forchette.

I filosofi pensano ma ogni tanto a qualcuno viene fame. Per mangiare ogni filosofo prende prima la forchetta alla sua destra e poi quella alla sua sinistra, e poi inizia a mangiare.



Benjamin D. Esham / Wikimedia Commons

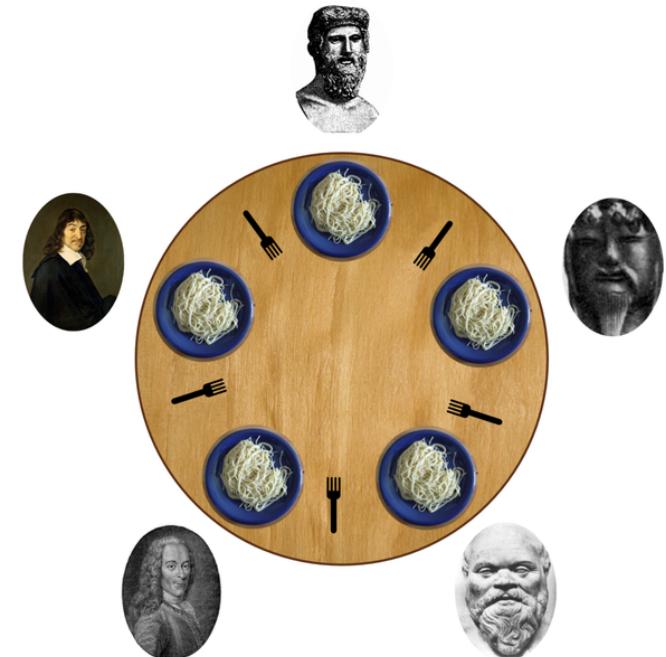
Un deadlock si verifica quando due o più processi sono bloccati per sempre in attesa l'uno dell'altro.

### Esempio: I Filosofi a Cena

Cinque filosofi sono seduti a tavola, un piatto di spaghetti e posto di fronte ad ogni filosofo e tra ogni piatto di spaghetti è posta una forchetta, per un totale di 5 forchette.

I filosofi pensano ma ogni tanto a qualcuno viene fame. Per mangiare ogni filosofo prende prima la forchetta alla sua destra e poi quella alla sua sinistra, e poi inizia a mangiare.

**Deadlock** – se a tutti i filosofi viene fame contemporaneamente ognuno di essi prenderà la forchetta alla propria destra, ma resterà per sempre in attesa che la forchetta alla propria sinistra venga rilasciata dal vicino.



Benjamin D. Esham / Wikimedia Commons

# Database Relazionali

Un **Database** è una collezione di dati organizzata (secondo qualche modello).

Col termine **Database Management System** si indica un applicativo software che permette di interagire con uno o più database.

Un **Database Relazionale** è un database nel quale i dati sono organizzati secondo il *modello relazionale* (vedi Algebra Relazionale).

Codd, E.F (1970). "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM. Classics. 13 (6): 377–87.  
doi:10.1145/362384.362685.

L'algebra relazionale è un modello di rappresentazione dei dati basato sulla logica del primo ordine.

Dati n (n intero positivo) insiemi disgiunti  $S_1, \dots, S_n$  una relazione  $R$  su  $S_1, \dots, S_n$  è un insieme di n-uple  $[v_1, \dots, v_n]$  con  $v_1$  in  $S_1$ ,  $v_2$  in  $S_2, \dots, v_n$  in  $S_n$ .  $R$  è detta essere una relazione n-aria.

## Esempio1

Siano  $P$  l'insieme di tutte le persone fisiche e  $CF$  l'insieme di tutti i possibili codici fiscali.

$$P = \{\text{alice, bob, charlie, ...}\} \quad CF = \{ \text{ALCALC111N, BBBB112N, ...} \}$$

La relazione  $R_1$  associa ad alcune persone il proprio codice fiscale

$$R_1 = \{ [\text{alice, ALCALC111N}], [\text{bob, BBBB112N}] \}$$

L'algebra relazionale è un modello di rappresentazione dei dati basato sulla logica del primo ordine.

Dati n (n intero positivo) insiemi disgiunti  $S_1, \dots, S_n$  una relazione  $R$  su  $S_1, \dots, S_n$  è un insieme di n-uple  $[v_1, \dots, v_n]$  con  $v_1$  in  $S_1$ ,  $v_2$  in  $S_2$ , ...,  $v_n$  in  $S_n$ .  $R$  è detta essere una relazione n-aria.

## Esempio2

Siano  $P$  l'insieme di tutte le persone fisiche e  $CF$  l'insieme di tutti i possibili codici fiscali e  $D$  l'insieme dei cani.

$$\begin{aligned} P &= \{\text{alice, bob, charlie, ...}\} & CF &= \{\text{ALCALC111N, BBBB112N, ...}\} \\ D &= \{\text{fuffy, doggy, pluto, ...}\} \end{aligned}$$

La relazione  $R_2$  associa ad alcune persone il proprio codice fiscale ed i propri cani.

$$R_2 = \{ [\text{alice}, \text{ALCALC111N}, \text{fuffy}], [\text{charlie}, \text{CRLCRL113N}, \text{doggy}], [\text{charlie}, \text{CRLCRL113N}, \text{pluto}] \}$$

L'algebra relazionale è un modello di rappresentazione dei dati basato sulla logica del primo ordine.

Dati n (n intero positivo) insiemi disgiunti S<sub>1</sub>, ..., S<sub>n</sub> una relazione R su S<sub>1</sub>, ..., S<sub>n</sub> è un insieme di n-uple [v<sub>1</sub>, ..., v<sub>n</sub>] con v<sub>1</sub> in S<sub>1</sub>, v<sub>2</sub> in S<sub>2</sub>, ..., v<sub>n</sub> in S<sub>n</sub>. R è detta essere una relazione n-aria.

NOTA: essendo insiemi, l'ordine in cui vengono presentate le tuple nelle relazioni è irrilevanti.

Ad esempio le due seguenti relazioni sono in realtà la stessa relazione

R<sub>2</sub> = { [alice, ALCALC111N , fuzzy],  
[charlie, CRLCRL113N, doggy],  
[charlie, CRLCRL113N, pluto] }

R<sub>4</sub> = { [charlie, CRLCRL113N, pluto],  
[charlie, CRLCRL113N, doggy],  
[alice, ALCALC111N , fuzzy] }

E' possibile estendere la nozione di relazione associando delle *etichette* ad ogni posizione nelle tuple.

Dato un intero positivo  $n$ ,  $n$  insiemi disgiunti  $S_1, \dots, S_n$  e  $n$  etichette  $I_1, \dots, I_n$ , una relazione con etichette  $R$  su  $S_1, \dots, S_n$  è costituita da un insieme di  $n$ -uple  $[v_1, \dots, v_n]$  con  $v_1$  in  $S_1$ ,  $v_2$  in  $S_2, \dots, v_n$  in  $S_n$  ed  $n$  etichette  $I_1, \dots, I_n$ .

Tali etichette danno indicazioni sul *ruolo* che svolgono gli elementi nella relazione.

### Esempio1

Siano  $P$  l'insieme di tutte le persone fisiche,  $S$  l'insieme di tutte le possibili stringhe di caratteri e  $CF$  l'insieme di tutti i possibili codici fiscali. Definiamo una relazione su  $P, S, S$  e  $CF$  (in grassetto le etichette).

<b>user</b>	<b>Name</b>	<b>Surname</b>	<b>CF</b>
alice	Alice	Rossi	ALC...
bob	Roberto	Verdi	RBR..

E' possibile estendere la nozione di relazione associando delle *etichette* ad ogni posizione nelle tuple.

Dato un intero positivo n, n insiemi disgiunti S<sub>1</sub>, ..., S<sub>n</sub> e n etichette l<sub>1</sub>, ..., l<sub>n</sub>, una relazione con etichette R su S<sub>1</sub>, ..., S<sub>n</sub> è costituita da un insieme di n-uple [v<sub>1</sub>, ..., v<sub>n</sub>] con v<sub>1</sub> in S<sub>1</sub>, v<sub>2</sub> in S<sub>2</sub>, ..., v<sub>n</sub> in S<sub>n</sub> ed n etichette l<sub>1</sub>, ..., l<sub>n</sub>.

Tali etichette danno indicazioni sul *ruolo* che svolgono gli elementi nella relazione.

**NOTA:** due relazioni possono differenziarsi anche solo per le etichette.

Utente	Comune di Nascita
alice	Roma
bob	Milano

Utente	Comune di Residenza
alice	Roma
bob	Milano

SQL (Structured Query Language, 1986) è un linguaggio per interrogare e modificare database relazionali. Vedi

- <https://en.wikipedia.org/wiki/SQL> e
- <http://www.w3schools.com/SQL/>

Esistono svariati DBMS che supportano questo linguaggio: Oracle, SQLServer, MySQL e MariaDB, Postgres, ...

MySQL workbench (multipiattaforma) e HeidiSQL sono due frontend grafici per la gestione di database relazionali che supportano SQL.

La creazione di un database in SQL avviene con la seguente sintassi

`CREATE DATABASE dbname;`

Nel caso in cui siano presenti diversi database, è necessario selezionare quello di lavoro con

`USE dbname;`

Per definire nuove relazioni (tabelle nel gergo SQL) invece è possibile usare comandi con la seguente sintassi

```
CREATE TABLE table_name
(
column_name1 data_type1(size1) ,
column_name2 data_type2(size2) ,
column_name3 data_type3(size3) ,
.....
);
```

### Esempio

```
CREATE TABLE UserDescription (user varchar(16), name varchar(255),
surname varchar(255), CF char(16));
```

NOTA: in SQL le etichette delle relazioni vengono denominate colonne.

Vedi [http://www.w3schools.com/SQL/sql\\_create\\_table.asp](http://www.w3schools.com/SQL/sql_create_table.asp)

Per popolare le relazioni invece la sintassi è la seguente

```
INSERT INTO table_name (column1,column2,column3,...)  
VALUES (value1,value2,value3,...);
```

### Esempio

```
INSERT INTO UserDescription (name, surname, CF)  
VALUES ('alice', 'Alice', 'Rossi', 'LCARSS81A01C351L');
```

Tra le etichette è possibile identificarne una o più che identificano delle posizioni nelle tuple i cui valori identificano univocamente le tuple.

### Esempio1

Nella relazione che segue il campo user può fungere da chiave primaria.

user 	Name	Surname	CF
alice	Alice	Rossi	ALC...
bob	Roberto	Verdi	RBR..

Tra le etichette è possibile identificarne una o più che identificano delle posizioni nelle tuple i cui valori identificano univocamente le tuple.

### Esempio2

Nella relazione che segue il campo user può fungere da chiave primaria, Ma anche il campo CF.

user	Name	Surname	CF 
alice	Alice	Rossi	ALC...
bob	Roberto	Verdi	RBR..

Le chiave primaria di una relazione in SQL può essere dichiarata al momento della creazione della relazione stessa specificando il *constraint PRIMARY KEY*.

```
CREATE TABLE table_name
(
column_name1 data_type1(size1) ,
column_name2 data_type2(size2) ,
column_name3 data_type3(size3) ,
...
PRIMARY KEY (column_namei, ..., column_namei+m)
);
```

I tipi di dato sono specifici per DBMS. Una lista dei tipi non ufficiale più di dato per i maggiori DBMS è disponibile al seguente indirizzo

[http://www.w3schools.com/SQL/sql\\_datatypes.asp](http://www.w3schools.com/SQL/sql_datatypes.asp)

I tipi di dato sono specifici per DBMS. Una lista non ufficiale dei tipi di dato per i maggiori DBMS è disponibile al seguente indirizzo

[http://www.w3schools.com/SQL/sql\\_datatypes.asp](http://www.w3schools.com/SQL/sql_datatypes.asp)

**Esempio**

```
CREATE TABLE UserDescription (user varchar(16), name varchar(255),  
surname varchar(255), CF char(16), PRIMARY KEY (user));
```

user 	Name	Surname	CF
alice	Alice	Rossi	ALC...
bob	Roberto	Verdi	RBR..

Tra le etichette è possibile identificare una o più che identificano delle posizioni nelle tuple i cui valori identificano univocamente le tuple.

### Esempio3

Nella relazione che associa ad ogni utente il proprio cane è necessario invece selezionare almeno due campi come chiave primaria.

Utente 	Nome	CF	Cane 
alice	Alice Rossi	ALCALC111N	fuffy
charlie	Carlo Verdi	CRLCRL121M	doggy
charlie	Carlo Verdi	CRLCRL121M	pluto
bob	Roberto Verdi	RBTRBT137M	fuffy

## Chiavi Primarie in SQL – Esempio Chiave Multipla

## Esempio

```
CREATE TABLE UserDescription (Utente varchar(16), Nome varchar(255),  
CF char(16), Cane varchar(16), PRIMARY KEY (Utente, Cane));
```

Utente 	Nome	CF	Cane 
alice	Alice Rossi	ALCALC111N	fuffy
charlie	Carlo Verdi	CRLCRL121M	doggy
charlie	Carlo Verdi	CRLCRL121M	pluto
bob	Roberto Verdi	RBTRBT137M	fuffy

E' possibile definire delle *chiavi esterne* che collegano n-uple di relazioni diverse.

Una label  $k$  in una relazione  $R$  è detta essere una chiave esterna verso un'altra relazione  $R'$  se tutti i valori in  $R$  corrispondenti a  $k$  sono valori della chiave primaria di  $R'$ .

### Esempio

Il campo Utente nella relazione Utenti\_Cani è una chiave esterna verso la relazione Utenti.

Utente 	Nome	CF
alice	Alice Rossi	ALCALC111N
charlie	Carlo Verdi	CRLCRL121M
bob	Roberto Verdi	RBTRBT137M

Utente 	Cane 
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

Analogamente alle chiavi primarie, le chiavi esterne vanno dichiarate in SQL con il constraint FOREIGN KEY

```
CREATE TABLE table_name
(
column_name1 data_type1(size1) ,
column_name2 data_type2(size2) ,
column_name3 data_type3(size3) ,
...
FOREIGN KEY (column_name) REFERENCES target_table(target_column)
);
```

## Esempio

```
CREATE TABLE Utente_Cane (Utente varchar(16), Cane varchar(255),
    PRIMARY KEY (Utente, Cane),
    FOREIGN KEY Utente REFERENCES Utente(Utente));
```

The diagram illustrates a primary key constraint. A blue arrow points from the primary key column 'Utente' in the first table to the primary key columns 'Utente' and 'Cane' in the second table. Both tables have a yellow key icon next to their primary key columns.

Utente	Nome	CF	Utente	Cane
alice	Alice Rossi	ALCALC111N	alice	fuffy
charlie	Carlo Verdi	CRLCRL121M	charlie	doggy
bob	Roberto Verdi	RBTRBT137M	charlie	pluto
			bob	fuffy

Oltre a dare indicazioni sulla semantica degli attributi (colonne), i constraint impediscono la creazione di inconsistenze (secondo la semantica che abbiamo inteso dare).

### Esempio

```
>CREATE TABLE UserDescription (user varchar(16), name varchar(255),  
    surname varchar(255), CF char(16), PRIMARY KEY CF);
```

```
>INSERT INTO UserDescription (user, name, surname, CF)  
    VALUES ('alice', 'Alice', 'Rossi', 'LCARSS81A01C351L');
```

```
>INSERT INTO UserDescription (user, name, surname, CF)  
    VALUES ('bob', 'Roberto', 'Verdi', 'LCARSS81A01C351L');
```

*Error Code: 1062. Duplicate entry 'LCARSS81A01C351L' for key  
'PRIMARY'*

L'operatore dell'algebra relazionale di proiezione  $\pi$  permette di ottenere da una relazione  $R$  una relazione  $R'$  che contiene solo alcune delle colonne di  $R$ .

$R$

User	Name	Surname	CF
alice	Alice	Rossi	ALC...
bob	Roberto	Verdi	RBR..

$\pi_{\text{Surname}, \text{CF}}(R)$

Surname	CF
Rossi	ALC...
Verdi	RBR..

Per ottenere una proiezione in linguaggio SQL è necessario usare il comando SELECT

```
SELECT column_name, column_name  
FROM table_name;
```

### Esempio

```
SELECT Surname, CF FROM UserDescription;
```

Il risultato di una proiezione ottenuta mediante una SELECT potrebbe contenere righe duplicate.

Utente	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

```
SELECT Utente FROM Utente_Cane;
```

Utente
alice
charlie
charlie
bob

Il risultato di una proiezione ottenuta mediante una SELECT potrebbe contenere righe duplicate. Per ovviare a questo inconveniente è possibile usare la clausola DISTINCT

Utente	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

```
SELECT Utente FROM Utente_Cane;
```

Utente
alice
charlie
bob

L'operatore dell'algebra relazionale di selezione  $\sigma$  permette di ottenere da una relazione  $R$  un sottoinsieme di  $R'$  che consiste in tutte le righe di  $R$  che soddisfano una determinata condizione.

## Utente\_Cane

Utente	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

$\sigma_{\text{Cane}=\text{fuffy}}(\text{Utente}_\text{Cane})$

Utente	Cane
alice	fuffy
bob	fuffy

## Selezione in SQL – la Clausola WHERE

L'operatore di selezione può essere espresso in SQL usando il comando SELECT e indicando il criterio di selezione in una clausula WHERE.

Utente	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

```
SELECT * FROM Utente_Cane WHERE Cane='fuffy';
```

Utente	Cane
alice	fuffy
bob	fuffy

Il *wildcard* \* nella SELECT sta ad indicare *tutti i campi* della relazione.

## Selezione in SQL – Condizioni nel WHERE

Nello specificare il criterio di selezione possono essere usati svariati operatori: =, <, >, <>, <=, >=, BETWEEN, LIKE, ...

Un elenco di operatori è disponibile alla seguente pagina

<https://en.wikipedia.org/wiki/SQL#Operators>

**Persona**

Nome	Cognome	Anno
Alice	Rossi	1985
Aldo	Bianchi	2003
Roberto	Rossi	1993

```
SELECT Nome, Cognome FROM Persona WHERE Nome LIKE 'A%';
```

Nome	Cognome	Anno
Alice	Rossi	1985
Aldo	Bianchi	2003

## Selezione in SQL – Operatori Booleani WHERE

Gli operatori appena visti possono essere ulteriormente combinati usando gli operatori booleani AND, OR e NOT

**Persona**

Nome	Cognome	Anno
Alice	Rossi	1985
Aldo	Bianchi	2003
Roberto	Rossi	1993

```
SELECT Nome,Cognome FROM Persona WHERE Nome LIKE 'A%' AND ANNO<2000;
```

Nome	Cognome	Anno
Alice	Rossi	1985

## Selezione in SQL – Condizioni nel WHERE

L'operatore di selezione può essere espresso in SQL usando il comando SELECT e indicando il criterio di selezione in una clausola WHERE.

Utente	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

```
SELECT * FROM Utente_Cane WHERE Cane='fuffy';
```

Utente	Cane
alice	fuffy
bob	fuffy

Il *wildcard* \* nella SELECT sta ad indicare *tutti i campi* della relazione.

L'operatore dell'algebra relazionale di rinominazione  $\rho$  permette di ottenere da una relazione  $R$  una relazione che si differenzia da  $R$  solo per una etichetta.

## Utente\_Cane

Utente	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

$P_{\text{Utente}/\text{UtentId}}(\text{Utente}_\text{Cane})$

UtentId	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

La rinominazione dei campi in SQL avviene postponendo all'etichetta da rinominare *as newlabel*

### Utente\_Cane

Utente	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

```
SELECT Utente as UserId, Cane FROM Utente_Cane;
```

UserId	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

A differenza dell'operatore di rinominazione nell'algebra relazionale, in una SELECT possono essere rinominati molteplici campi contemporaneamente.

### Utente\_Cane

Utente	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

```
SELECT Utente as UserId, Cane as DogId FROM Utente_Cane;
```

UserId	DogId
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

Proiezione, selezione e ridenominazione possono essere specificati contemporaneamente nella stessa SELECT (anche solo una coppia di questi operatori).

### Utente\_Cane

Utente	Cane
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

```
SELECT Utente as UserId FROM Utente_Cane WHERE Cane='fuffy' ;
```

UserId
alice
bob

L'operatore binario dell'algebra relazionale di giunzione (naturale)  $\bowtie$  serve per *combinare* le n-uple di due diverse relazioni. Il risultato della giunzione  $R \bowtie S$  è l'insieme di tutte le combinazioni di tuple di R ed S che sono uguali rispetto agli attributi che hanno in comune.

## Utente

UtentId	Nome	CF
alice	Alice Rossi	ALCALC111N
charlie	Carlo Verdi	CRLCRL121M
bob	Roberto Verdi	RBTRBT137M

## Utente\_Cane

UtentId	CanId
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

## Utente $\bowtie$ Utente\_Cane

UtentId	Nome	CF	CanId
alice	Alice Rossi	ALCALC111N	fuffy
charlie	Carlo Verdi	CRLCRL121M	doggy
charlie	Carlo Verdi	CRLCRL121M	pluto
bob	Roberto Verdi	RBTRBT137M	fuffy

Vedi [https://en.wikipedia.org/wiki/Relational\\_algebra#Natural\\_join\\_.28.E2.8B.88.29](https://en.wikipedia.org/wiki/Relational_algebra#Natural_join_.28.E2.8B.88.29) e

## Operatore di Giunzione Naturale in SQL

La giunzione naturale si esprime in SQL sempre attraverso il SELECT ma attraverso le parole chiave INNER JOIN

```
SELECT col1, ...,coln FROM table1 INNER JOIN table2
```

**Utente**

<b>UtentId</b>	<b>Nome</b>	<b>CF</b>
alice	Alice Rossi	ALCALC111N
charlie	Carlo Verdi	CRLCRL121M
bob	Roberto Verdi	RBTRBT137M

**Utente\_Cane**

<b>UtentId</b>	<b>CanId</b>
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

```
SELECT * FROM Utente INNER JOIN Utente_Cane;
```

<b>UtentId</b>	<b>Nome</b>	<b>CF</b>	<b>CanId</b>
alice	Alice Rossi	ALCALC111N	fuffy
charlie	Carlo Verdi	CRLCRL121M	doggy
charlie	Carlo Verdi	CRLCRL121M	pluto
bob	Roberto Verdi	RBTRBT137M	fuffy

Vedi [http://www.w3schools.com/sql/sql\\_join.asp](http://www.w3schools.com/sql/sql_join.asp)

## Operatore di Giunzione Naturale in SQL

In SQL la giunzione può avvenire anche su campi che non abbiano la stessa etichetta specificando questi ultimi nella clausola ON.

```
SELECT t1.c1,...,t1.cn, ...,t2.cm   FROM t1 INNER JOIN t2
ON t1.ci=t2.ch
```

**Utente**

<b>Id</b>	<b>Nome</b>	<b>CF</b>
alice	Alice Rossi	ALCALC111N
charlie	Carlo Verdi	CRLCRL121M
bob	Roberto Verdi	RBTRBT137M

**Utente\_Cane**

<b>UtentId</b>	<b>Cane</b>
alice	fuffy
charlie	doggy
charlie	pluto
bob	fuffy

```
SELECT * FROM Utente INNER JOIN Utente_Cane ON
Utente.Id=Utente_Cane.UtenteId;
```

<b>UtentId</b>	<b>Nome</b>	<b>CF</b>	<b>CanId</b>
alice	Alice Rossi	ALCALC111N	fuffy
charlie	Carlo Verdi	CRLCRL121M	doggy
charlie	Carlo Verdi	CRLCRL121M	pluto
bob	Roberto Verdi	RBTRBT137M	fuffy

Per accedere ad un database relazionale in Java è necessario ottenere un oggetto che implementi l'interfaccia `java.sql.Connection`.

### Esempio

```
final Connection conn = DriverManager.getConnection(  
        "jdbc:mysql://localhost/");  
conn.close();
```

Vedi <https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-usagenotes-connect-drivermanager.html>

TODO transazioni

# Design Pattern

Un *software design pattern* è una soluzione (o un template di soluzione) generale e riusabile all'interno di specifici contesti per un problema di design che occorre di frequente.

Il concetto di design pattern viene esplicitato nel 1995 nel libro “Design Patterns: Elements of Reusable Object-Oriented Software” scritto dalla cosiddetta *Gang of Four* : Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

I design pattern vengono classificati in 4 categorie, di seguito elencate assieme ad alcuni pattern che andremo a esaminare:

*Creazionali* – riguardano la creazione di nuovi oggetti;

*Strutturali* – riguardano la definizione di classi e strutture complesse di oggetti;

*Comportamentali* – riguardano le comunicazioni tra oggetti e le reazioni degli oggetti agli eventi.

Un *Factory method* è un metodo che genera nuove istanze di oggetti.

```
...
Person createPerson(final String name, final String surname,
    final String cf, boolean isMale) {
    Return new Person(name, surname, cf, isMale);
}
...
```

Una factory è un oggetto che contiene dei factory method.

Una factory è un oggetto che contiene dei factory method. Normalmente le factory vengono definite come interfacce e le implementazioni di queste generano oggetti di differenti classi.

```
public interface PersonFactory{  
    Person createPerson(String name, String surname,  
                        String cf, boolean isMale);  
}  
  
/**  
 * persons in a persistent storage  
 */  
public class PersistentPersonFactory implement PersonFactory{  
    final Connection conn;  
    public Person createPerson(String name, String surname,  
                              String cf, boolean isMale) {  
        return new PersistentPerson(conn, name, surname, cf, is  
Male);  
    }  
}  
  
public class VolatilePersonFactory implement PersonFactory{  
...  
}
```

Questo permette di creare classi cui viene passata una factory come parametro senza specificare che tipi di oggetti verranno creati dalla factory.

```
public class PersonRegistry{  
    private final PersonFactory factory;  
  
    public PersonRegistry(final PersonFactory factory) {  
        this.factory=factory;  
    }  
  
    public Person add(final String name, final String surname,  
        final String cf, final boolean isMale) {  
        final Person p = factory.create(...);  
    }  
}  
...
```

Una factory può essere usata per definire in maniera più chiara diversi costruttori.

```
public interface PersonFactory{  
    Person createPerson(String name, String surname,  
                        String cf, boolean isMale);  
    Person createMan(String name, String surname, String cf);  
    Person createWoman(String name, String surname, String cf);  
}  
...
```

Una factory può essere usata anche per creare diverse istanze di oggetti da usare insieme.

```
public interface PersonsFactory{  
    Person createMan(String name, String surname, String cf);  
    Person createWoman(String name, String surname, String cf);  
  
    //persons must be created with the other two methods  
    Family createFamily(Collection<Person> persons);  
}  
  
....
```

Un singleton è un oggetto che può essere istanziato una sola volta. E' da molti considerato un anti-pattern

```
public class Logger{  
    public static final Logger INSTANCE = new Logger();  
    ...  
}  
...  
  
public class Example class{  
    void aMethod() {  
        ...  
        Logger.INSTANCE.log("executing aMethod");  
    }  
}
```

Nel caso di strutture ad albero, il pattern composite prevede la definizione di una interfaccia unica per trattare in maniera uniforme sia gli oggetti singoli (foglie) che le composizioni di oggetti (alberi).

```
public interface Graphic{
    void print();
}

public class Ellipse implements Graphic{
    public void print() {
        //draw the ellipse
    }
}

public class CompositeGraphic implements Graphic{
    public void add(Graphic g) {
        //add g to this composite
    }
    public void print() {
        //draw all the added graphics
    }
}
```

Il pattern decorator permette di estendere le funzionalità di un oggetto *delegato* che implementa la stessa interfaccia del decoratore.

```
public interface Graphic{
    void print();
}

public class GraphicWithLog implements Graphic{
    private final Graphic delegate;
    GraphicWithLog(final Graphic delegate) {
        this.delegate=delegate;
    }
    public void print() {
        this.delegate.print();
        Logger.INSTANCE.log("print performed");
    }
}
```

Il pattern decorator permette di decorare un oggetto già decorato

```
public interface Graphic{
    void print();
}

public class GraphicWithLog implements Graphic{
    ...
    public void print(){
        this.delegate.print();
        Logger.INSTANCE.log("print performed");
    }
}

public class GraphicWithSound implements Graphic{
    ...
    public void print(){
        this.delegate.print();
        Beeper.INSTANCE.beep();
    }
}
...
Graphic graphicGWithLogAndBeep =
    new GraphicWithLog(new GraphicWithBeep(G));
```

Vedi [https://en.wikipedia.org/wiki/Decorator\\_pattern](https://en.wikipedia.org/wiki/Decorator_pattern)

Un oggetto *facade* fornisce una interfaccia semplificata ad un corpus esteso e/o complesso di codice, ad esempio una libreria.

```
class ComputerFacade {  
    private CPU processor;  
    private Memory ram;  
    private HardDrive hd;  
  
    public ComputerFacade() {  
        this.processor = new CPU();  
        this.ram = new Memory();  
        this.hd = new HardDrive();  
    }  
  
    public void start() {  
        processor.freeze();  
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));  
        processor.jump(BOOT_ADDRESS);  
        processor.execute();  
    }  
}
```

Vedi [https://en.wikipedia.org/wiki/Facade\\_pattern](https://en.wikipedia.org/wiki/Facade_pattern)

Un *proxy* è un oggetto che funge da interfaccia verso qualcosaltro al fine ad esempio di semplificare l'accesso alla risorsa.

```
class DBProxy {  
  
    public DBProxy(final String dburl) {  
        ...  
    }  
  
    public void executeUpdate(String query) {  
        conn.executeUpdate(query);  
    }  
  
    public void close() {  
        conn.close();  
    }  
}
```

Nel proxy può essere specificata anche logica di business ulteriore al fine ad esempio di restringere l'accesso alla risorsa.

```
class DBProxy {  
  
    public DBProxy(final String dburl) {  
        ...  
    }  
  
    public void executeUpdate(String query, final Credentials userCred) {  
        if (check(userCred)) {  
            conn.executeUpdate(query);  
            conn.commit();  
        }  
    }  
  
    public void close() {  
        conn.close();  
    }  
  
    private boolean check(final Credentials userCred) { ... }  
}
```

## Pattern Comportamentali - Catena di Responsabilità

Una catena di responsabilità serve a far processare ogni elemento dal primo elemento della catena che è in grado di farlo.

```
class interface MedicalDoctor {  
    boolean canHandle(Patient p);  
    void attend(p);  
}  
  
class Pediatra implements MedicalDoctor{ ... }  
  
class Internista implements MedicalDoctor{ ... }  
  
class Andrologo implements MedicalDoctor{ ... }  
  
class Ginecologo implements MedicalDoctor{ ... }  
  
class Hospital{  
    final List<MedicalDoctor> chain;  
    Hospital(final List<MedicalDoctor> chain) { ... }  
    void attend(Patient p) {  
        for(MedicalDoctor d : chain)  
            if (d.canHandle(p))  
                d.attend(p);  
    }  
}
```

Vedi [https://en.wikipedia.org/wiki/Facade\\_pattern](https://en.wikipedia.org/wiki/Facade_pattern)

## Pattern Comportamentali - Command

Il Command Pattern prevede di encapsulare le azioni da eseguire e tutti i dati necessari all'interno di un oggetto (il command appunto) da passare ad un esecutore.

Un esempio classico è un *serializzatore* di task. Diversi processi concorrenti devono accedere ad un pool di risorse condivise, ma è opportuno che su questo pool le operazioni vengano eseguite una alla volta.

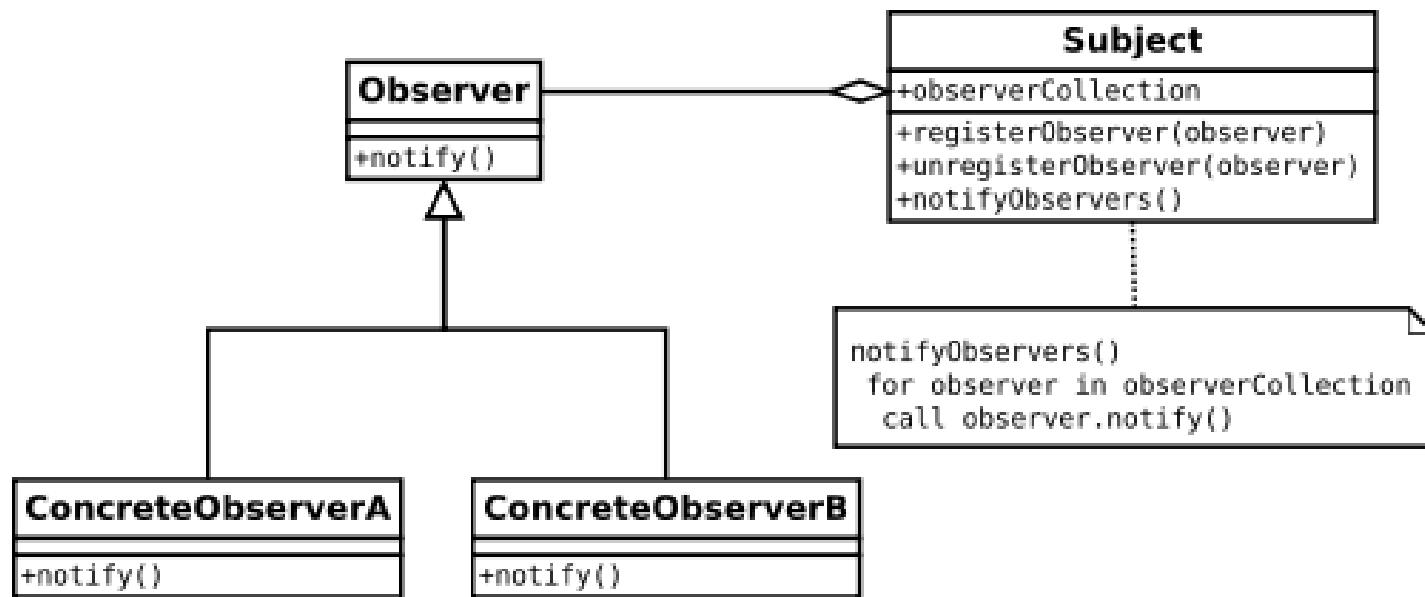
I vari thread allora encapsulano le operazioni in oggetti che implementano la stessa interfaccia e le inseriscono in una coda di esecuzione.

Il serializzatore preleva le operazioni dalla coda e le esegue una alla volta.

## Pattern Comportamentali - Observer

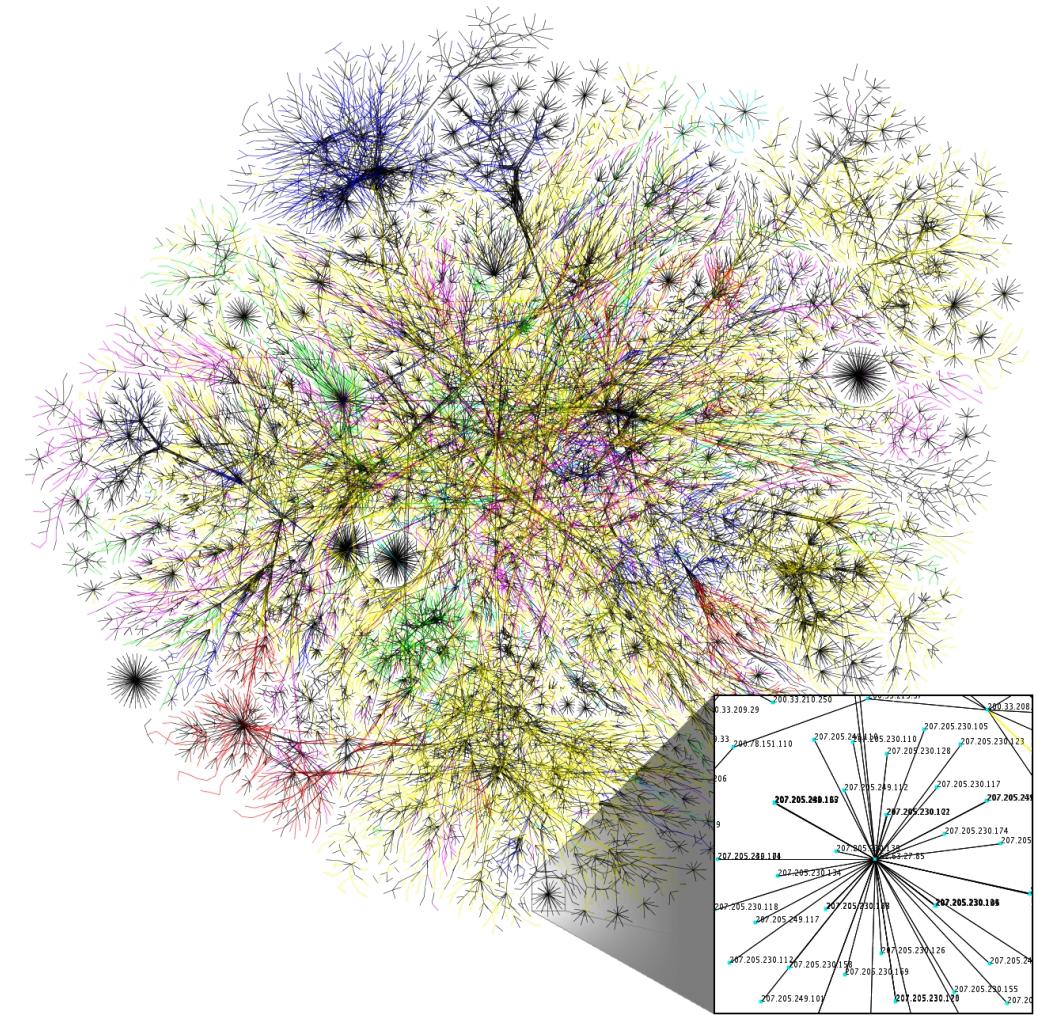
Il pattern *observer* (detto anche *listener*) prevede la possibilità da parte di un oggetto (*Observer*) di ricevere notifiche sui cambi di stato di vari oggetti *Observable*.

Nell'esempio precedente, i thread che inviano i task all'esecutore potrebbero osservare i command per essere notificati della loro esecuzione.



Vedi [https://en.wikipedia.org/wiki/Command\\_pattern](https://en.wikipedia.org/wiki/Command_pattern)

# Programmazione Web



“The Internet is the global system of interconnected computer networks that use the Internet protocol suite (TCP/IP) to link billions of devices worldwide. It is a network of networks that consists of millions of private, public, academic, business, and government networks of local to global scope, linked by a broad array of electronic, wireless, and optical networking technologies.”

# Da Wikipedia

By The Opte Project [CC BY 2.5 (<http://creativecommons.org/licenses/by/2.5>) or CC BY 2.5 (<http://creativecommons.org/licenses/by/2.5>)], via Wikimedia Commons

Vedi <https://en.wikipedia.org/wiki/Internet>

1961 - primo articolo sul packet switching: L. Kleinrock, "Information Flow in Large Communication Nets", RLE Quarterly Progress Report;

1969 - ARPANET: prima connessione e scambio di pacchetti tra due computer rispettivamente all'Università della California (UCLA) e allo Stanford Research Institute (SRI);

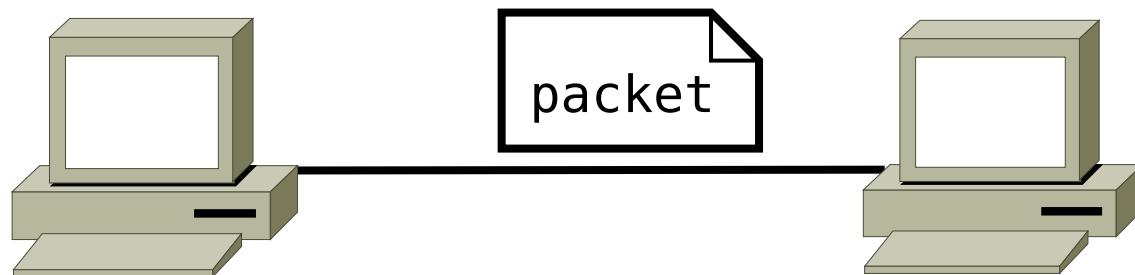
1972 - viene presentata ufficialmente ARPANET e la prima applicazione: e-mail;

1974 - nasce il protocollo TCP/IP : V. G. Cerf and R. E. Kahn, "A protocol for packet network interconnection" IEEE Trans. Comm. Tech., vol. COM-22, V 5, pp. 627-641;

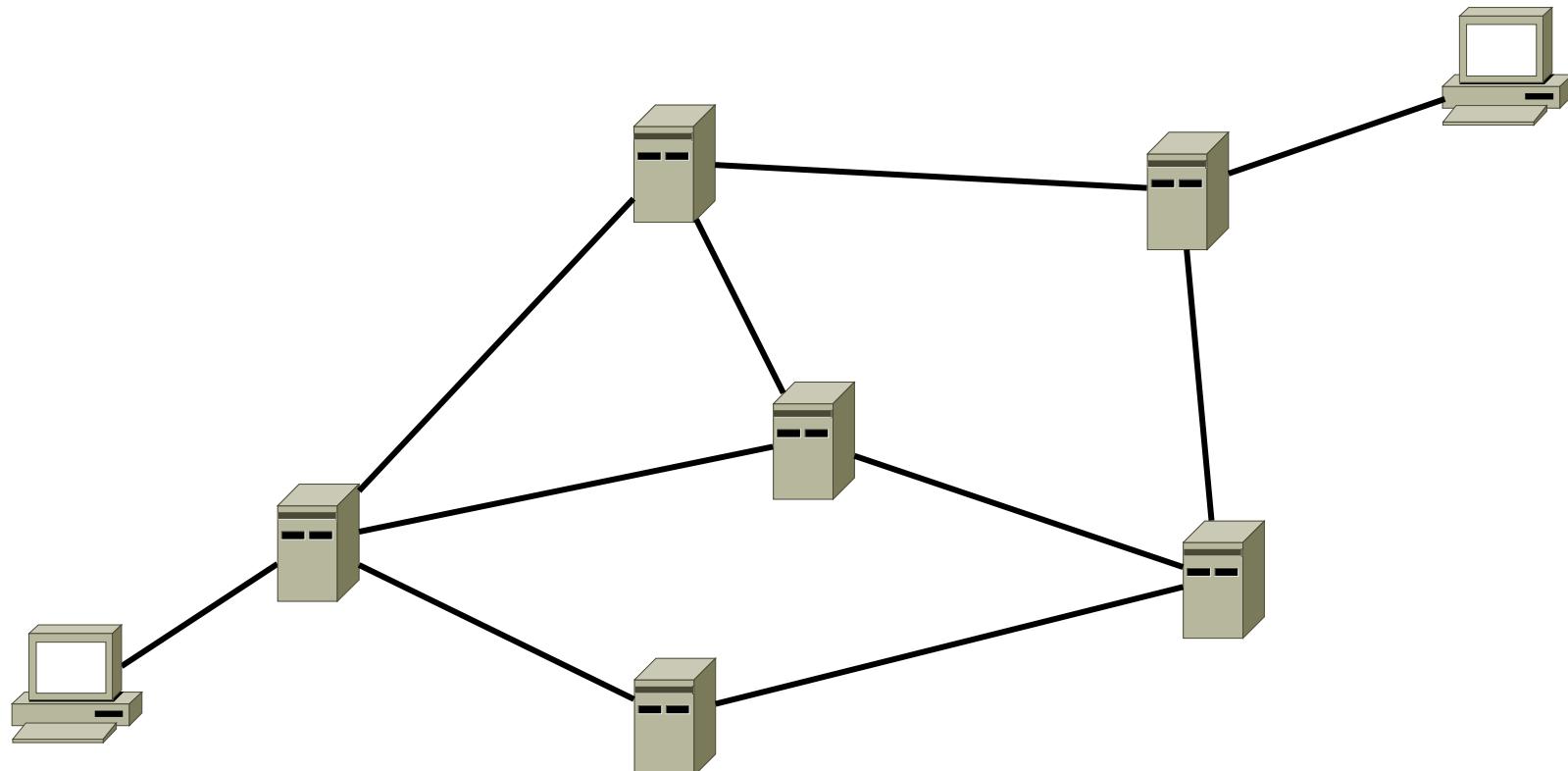
1993 - il CERN rilascerà di pubblico dominio i primi software per il world wide web, tra cui il primo browser chiamato per l'appunto World Wide Web.

L. Kleinrock, "Information Flow in Large Communication Nets", RLE Quarterly Progress Report;

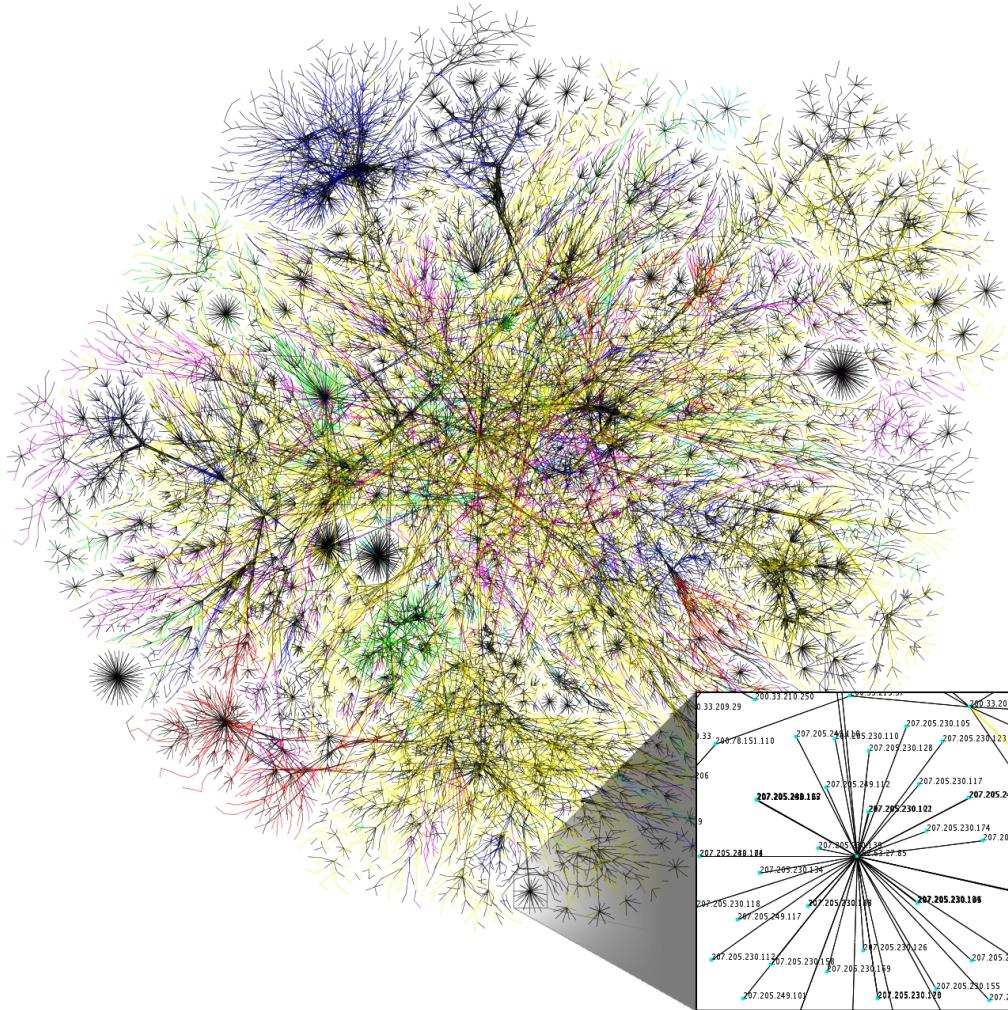
Gli elaboratori si scambiano informazioni raggruppate in unità di dimensione fissata chiamate *pacchetti*.



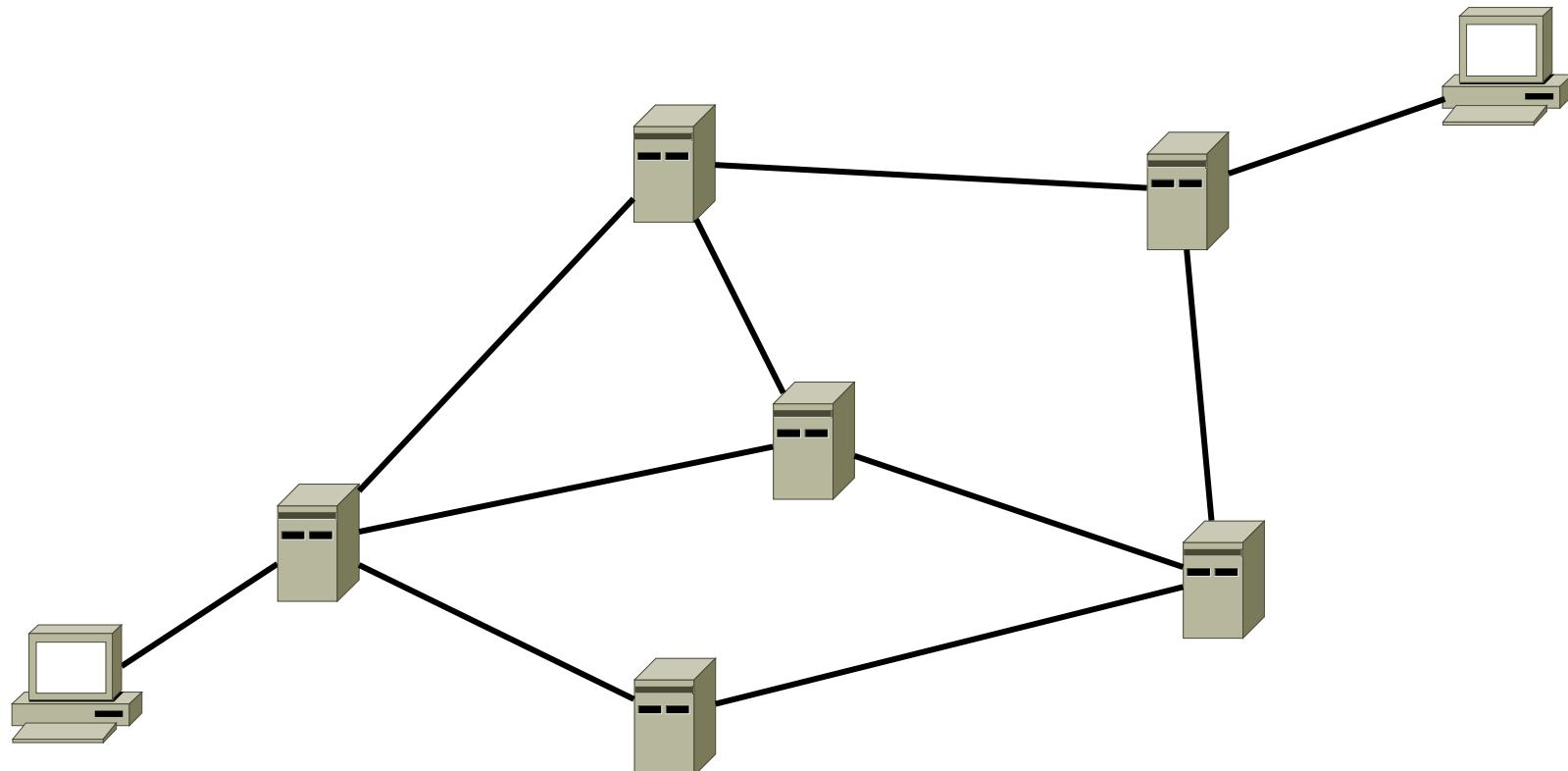
Una rete di computer connette diversi elaboratori.



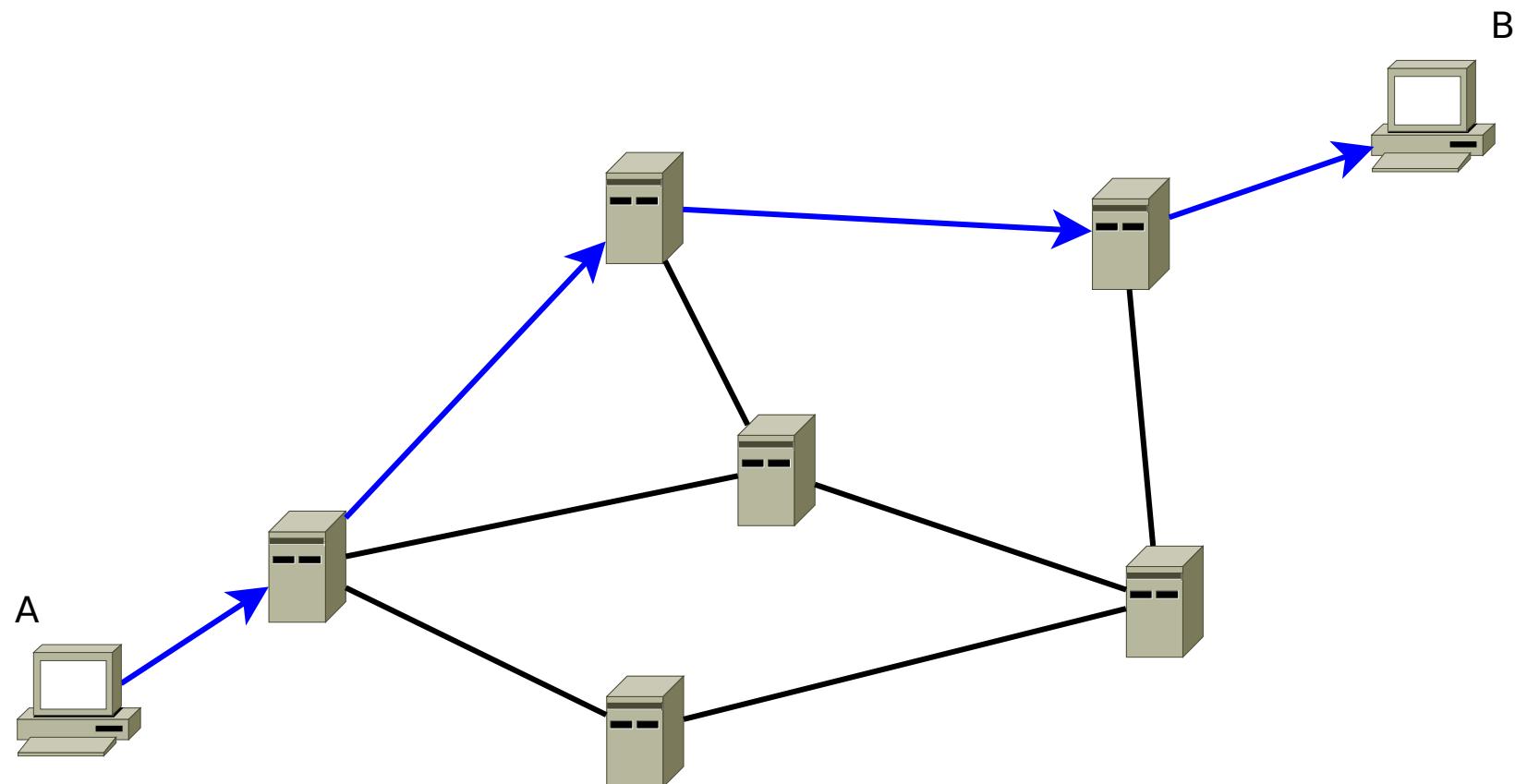
**Internet è la rete delle reti.**



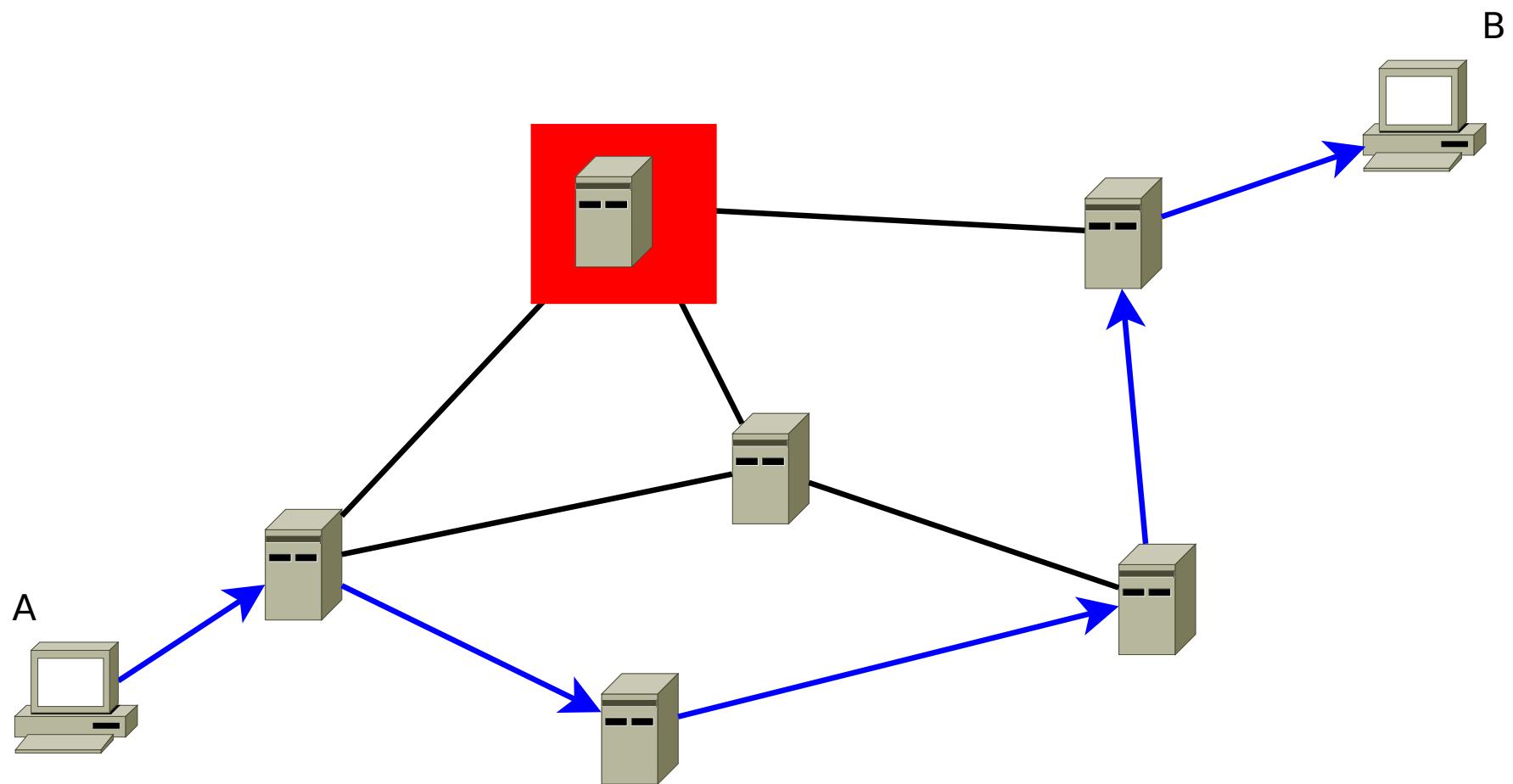
I *nodi* su internet sono identificati da indirizzi IP univoci. In IPv4 (vedi RFC791) gli indirizzi sono di 4 byte, ad esempio 151.97.240.5.



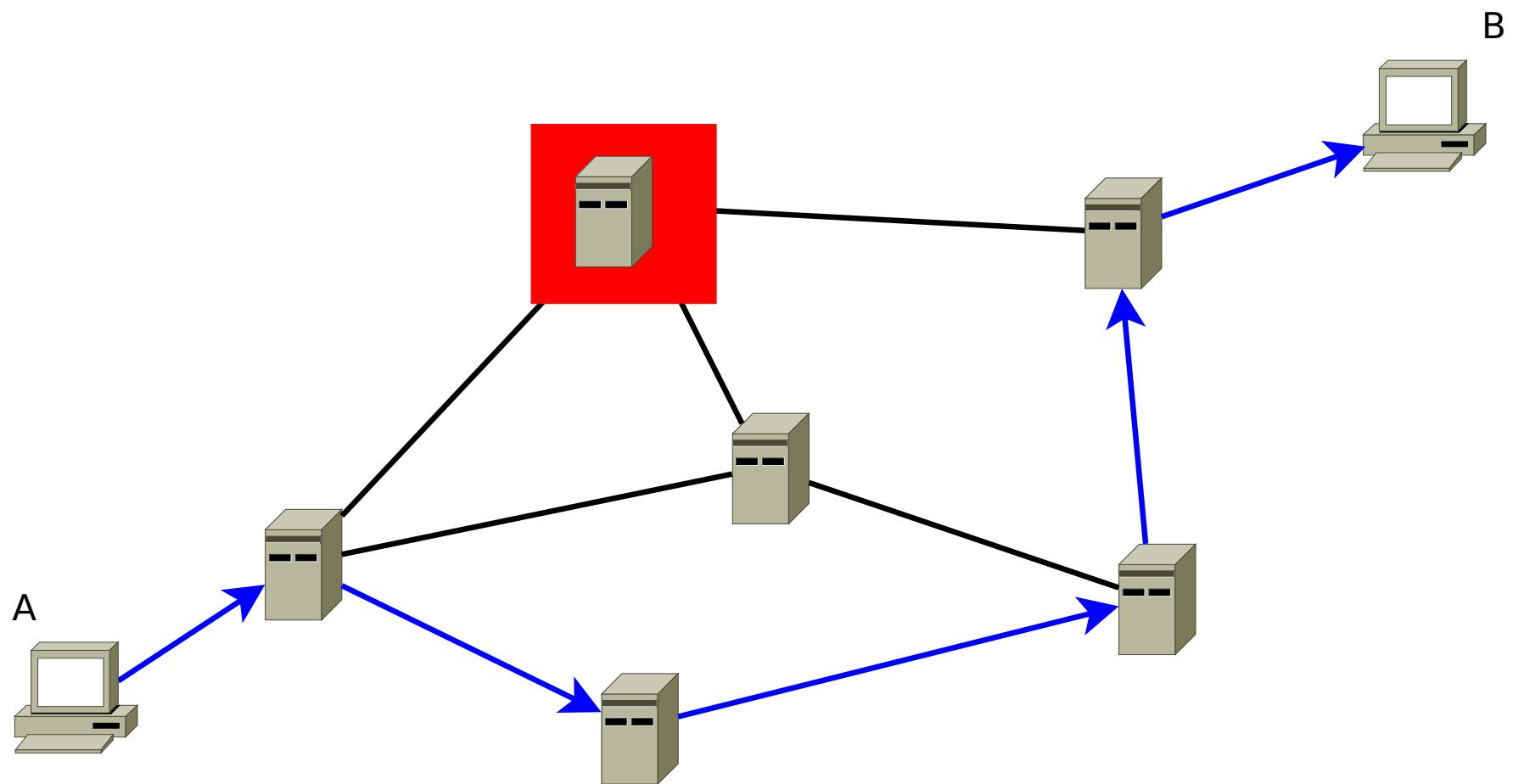
Un *messaggio* inviato da A verso B attraversa un insieme di nodi. Gli algoritmi che indicano ad ogni nodo come smistare il pacchetto si chiamano algoritmi di *routing*.



Se uno dei nodi sul percorso è sovraccarico o non funziona bene, viene utilizzato un percorso alternativo.



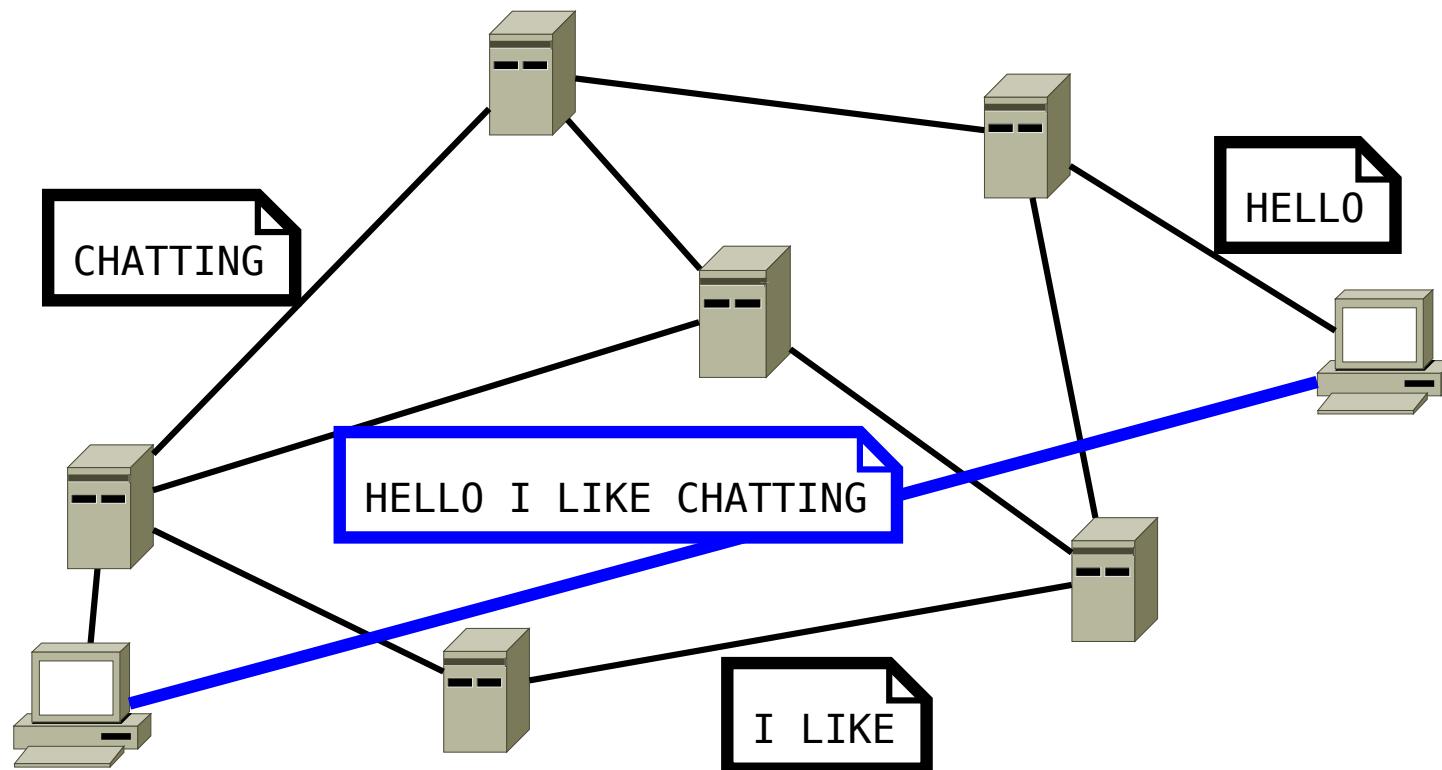
Se uno dei nodi sul percorso è sovraccarico o non funziona bene, viene utilizzato un percorso alternativo. Questa caratteristica è rafforzata se si usa il protocollo TCP/IP



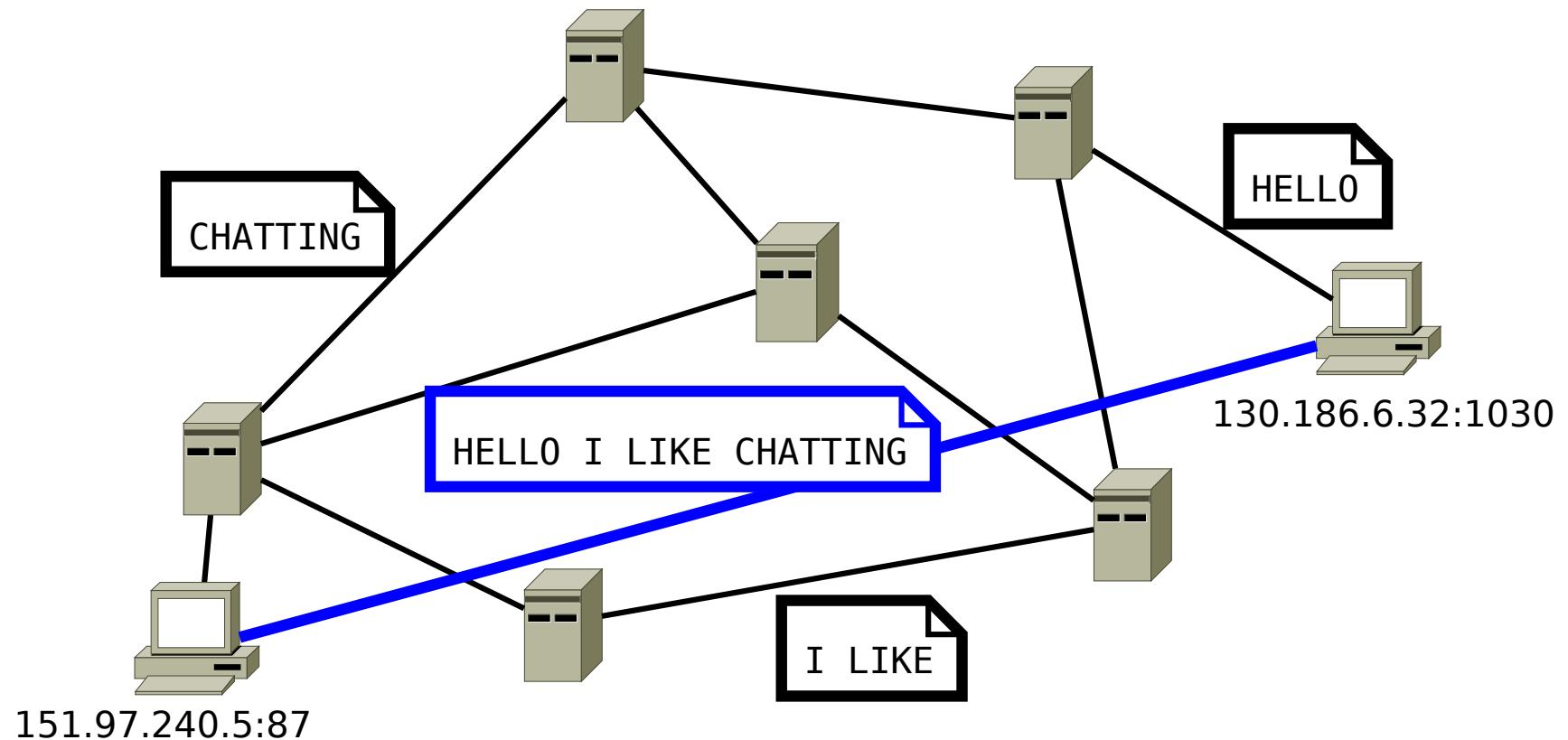
Il protocollo TCP/IP lavora usando IP per l'indirizzamento ma aggiunge le seguenti funzionalità:

- **affidabilità** : vengono implementati meccanismi basati su ACK per ritrasmettere pacchetti eventualmente persi;
- **integrità** : attraverso dei checksum è possibile verificare se un pacchetto è arrivato integro a destinazione;
- **ordinamento** : è possibile ricostruire l'ordine in cui i pacchetti sono stati inviati.

Il protocollo TCP permette ai software che lo usano di astrarsi dal concetto di pacchetto per usare quello di *connessione* vista come uno *stream* bidirezionale di dati.



Per stabilire una connessione TCP/IP oltre all'IP è necessario stabilire una porta (numero intero). Normalmente ogni servizio su un elaboratore resta in ascolto su una porta attraverso la quale contattarlo.



Per stabilire una connessione TCP/IP oltre all'IP è necessario stabilire una porta (numero intero). Normalmente ogni servizio su un elaboratore resta in ascolto su una porta attraverso la quale contattarlo.

Alcune porte standard sono

20 - FTP

22 - SSH

25 - SMTP

80 - HTTP

190 - POP

443 - HTTPS

E' possibile stabilire una connessione TCP (justo per prova) con un servizio remoto in ascolto attraverso il comando

```
telnet <ip> <port>
```

*“Imagine, then, the references in this document all being associated with the network address of the thing to which they referred, so that while reading this document you could skip to them with a click of the mouse.” Tim Berners Lee, 1989*

*“Imagine, then, the references in this document all being associated with the network address of the thing to which they referred, so that while reading this document you could skip to them with a click of the mouse.” Tim Berners Lee, 1989*

Nel 1993 il CERN rilascerà di pubblico dominio i primi software per il world wide web, tra cui il primo browser chiamato per l'appunto World Wide Web.

*“Imagine, then, the references in this document all being associated with the network address of the thing to which they referred, so that while reading this document you could skip to them with a click of the mouse.” Tim Berners Lee, 1989*

Nel 1993 il CERN rilascerà di pubblico dominio i primi software per il world wide web, tra cui il primo browser chiamato per l'appunto World Wide Web.

*“The World Wide Web (WWW, or simply Web) is an information space in which the items of interest, referred to as resources, are identified by global identifiers called Uniform Resource Identifiers (URI).”*

da Architecture of the World Wide Web, Volume I

Le prime specifiche rilasciate furono:

- Uniform Resource Locators (URLs),
- Hypertext Transfer Protocol (HTTP),
- Hypertext Markup Language (HTML).

Uniform Resource Locator (URL, vedi RFC1738) è un formato di *nomi* per indicare risorse sul web.

Esempi di URL sono <http://google.com>, <https://www.ietf.org/rfc/rfc1738.txt>

Uniform Resource Locator (URL, vedi RFC1738) è un formato di *nomi* per indicare risorse sul web.

Esempi di URL sono `http://google.com`, `https://www.ietf.org/rfc/rfc1738.txt`

In generale una URL ha il seguente formato

`<scheme>:<scheme-specific-part>`

dove la sintassi della scheme-specific-part deve rispettare le condizioni indicate nel documento di specifica dello scheme.

Tra gli schemi più usati troviamo `http`, `https`, `mailto`, `ftp`, ... (gli schemi solitamente indicano il protocollo)

Gli schemi più comuni adottano la Common Internet Scheme Syntax

<scheme>:// [<user> [:<password>]@] <host> [:<port>] [/<url-path>]

url-path, se presente, è una parte gerarchica nella quale tutti i livelli sono separati da /

<https://www.ietf.org/rfc/rfc1738.txt>

Gli schemi più comuni adottano la Common Internet Scheme Syntax

<scheme>:// [<user> [:<password>]@] <host> [:<port>] [/<url-path>]

url-path, se presente, è una parte gerarchica nella quale tutti i livelli sono separati da /

<https://www.ietf.org/rfc/rfc1738.txt>

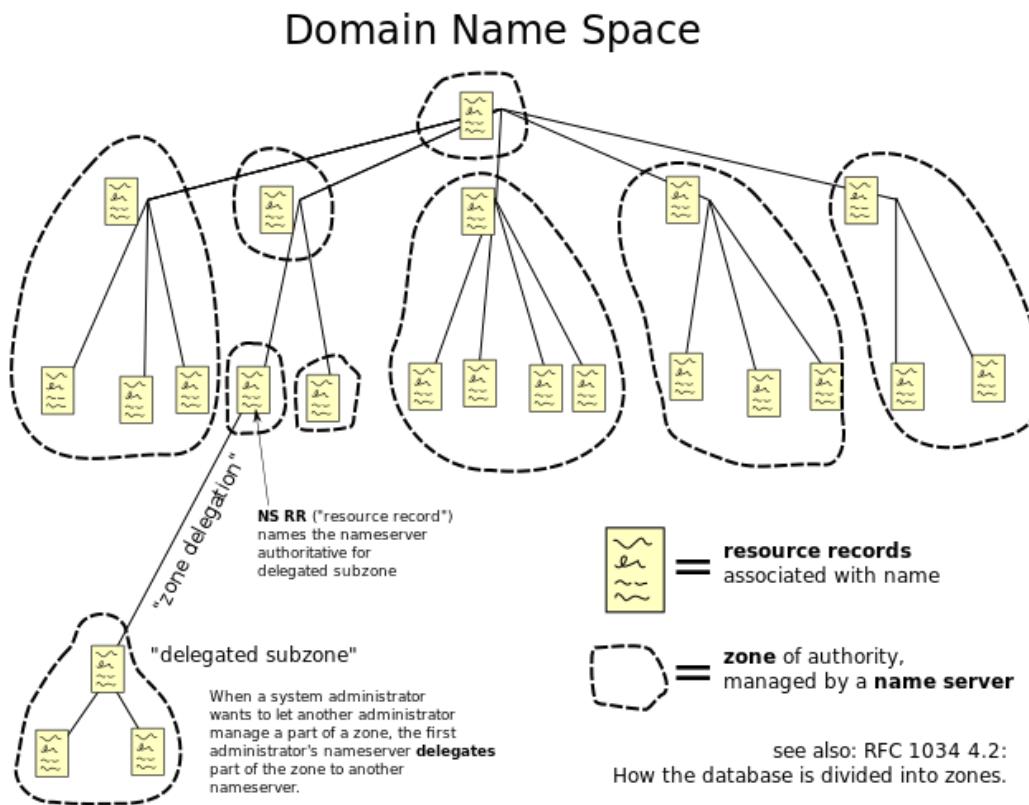
Dopo al termine del path alcuni URL Scheme prevedono la presenza di una *query-part*

<http://example.org?id=1024&var=v>

La nozione di URL viene successivamente estesa e superata da quella di URI (Uniform Resource Identifier, RFC3986) e poi da quella di IRI (Internationalized Resource Identifiers, RFC3987).

*Domain Name System* (DNS, vedi RFC1034) è un sistema gerarchico decentralizzato che permette di associare a *nomi di dominio* delle risorse disponibili su Internet.

Nella sua versione più basilare associa ad un nome di dominio un indirizzo IP. Queste associazioni risiedono, in una organizzazione gerarchica, all'interno dei cosiddetti sever DNS che comunicano tra loro usando il protocollo DNS.



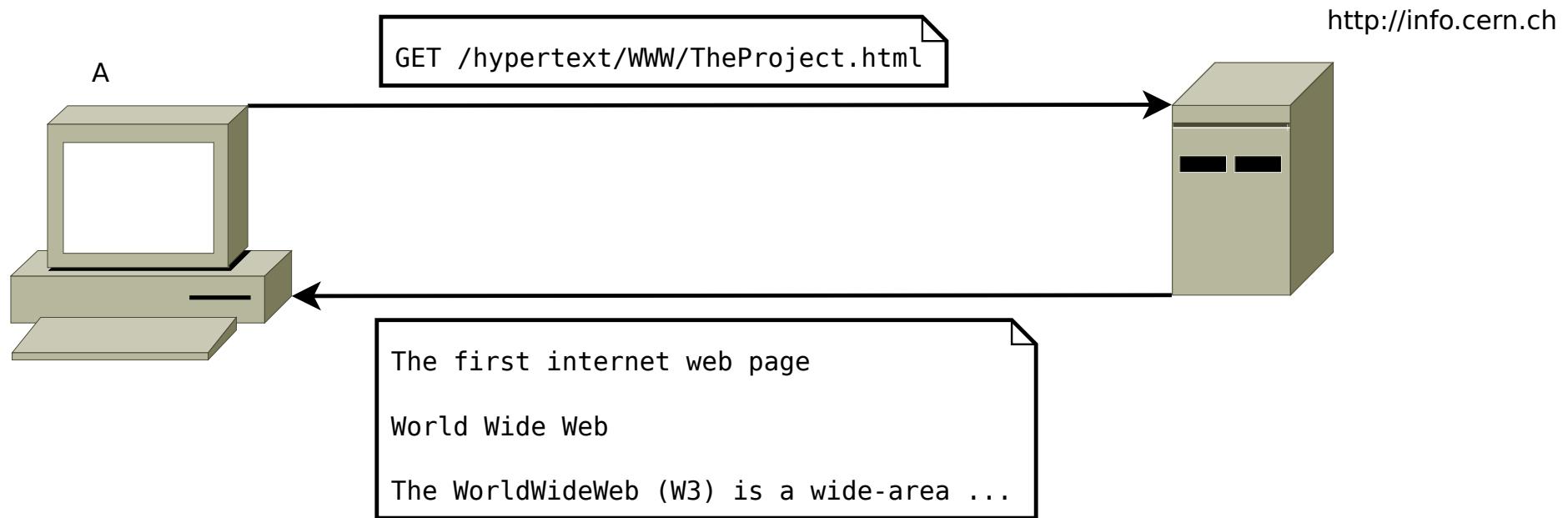
I nomi di dominio riflettono la struttura gerarchica del domain name system. I vari segmenti sono separati da “.” e vanno, da destra verso sinistra, dal livello più grande al più piccolo.

www.dmi.unict.it

Il comando nslookup <domainname> permette di ricavare l'ip associato al nome di dominio specificato.

Hyper Text Transfer Protocol (HTTP, RFC2616), è un protocollo *richiesta-risposta* per comunicazioni client-server:

- il client invia una stringa (richiesta) al server su una connessione TCP
- sulla stessa connessione il server risponde
- la connessione viene chiusa.



Una richiesta HTTP contiene il *metodo*

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

Una richiesta HTTP contiene il *metodo*, la *risorsa* cui il metodo fa riferimento

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

Una richiesta HTTP contiene il *metodo*, la *risorsa* cui il metodo fa riferimento, la *versione* del protocollo HTTP

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

Una richiesta HTTP contiene il *metodo*, la *risorsa* cui il metodo fa riferimento, la *versione* del protocollo HTTP ed alcune informazioni *Header*

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

Una richiesta HTTP contiene il *metodo*, la *risorsa* cui il metodo fa riferimento, la *versione* del protocollo HTTP ed alcune informazioni *Header*. Può contenere anche un *corpo*.

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

Il server risponde con la versione del protocollo ed un codice di risposta (vedi RFC 7231 sezione 6)

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

**HTTP/1.1 200 OK**

```
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

Il server risponde con la versione del protocollo ed un codice di risposta (vedi RFC 7231 sezione 6), le informazioni header

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

Il server risponde con la versione del protocollo ed un codice di risposta (vedi RFC 7231 sezione 6), le informazioni header ed eventualmente un body

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

Server response:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
```

Hello World! My payload includes a trailing CRLF.

I *metodi* forniti da HTTP 1.1 sono i seguenti:

- GET permette di ottenere una risorsa e i suoi metadati (charset encoding, lingua, data di creazione);
- HEAD permette di ottenere (solo) i metadati relativi ad una risorsa;
- POST simile a GET ma permette di inviare nella richiesta dei blocchi di dati che si presume siano processati dal server per fornire la risposta. Un sempio classico sono le form;
- PUT permette di creare una risorsa o modificare lo stato di una risorsa esistente;
- DELETE serve a rimuovere una risorsa dal server;
- CONNECT usato per i proxy e per stabilire canali di comunicazione sicuri;
- OPTIONS permette di richiedere informazioni sulle opzioni possibili di comunicazione relative ad una risorsa;
- TRACE il server risponde *rimbalzando* un messaggio identico a quello che è stato inviato dal client.

I *metodi* forniti da HTTP 1.1 sono i seguenti:

- GET permette di ottenere una risorsa e i suoi metadati (charset encoding, lingua, data di creazione);
- HEAD permette di ottenere (solo) i metadati relativi ad una risorsa;
- POST simile a GET ma permette di inviare nella richiesta dei blocchi di dati che si presume siano processati dal server per fornire la risposta. Un sempio classico sono le form;
- PUT permette di creare una risorsa o modificare lo stato di una risorsa esistente;
- DELETE serve a rimuovere una risorsa dal server;
- CONNECT usato per i proxy e per stabilire canali di comunicazione sicuri;
- OPTIONS permette di richiedere informazioni sulle opzioni possibili di comunicazione relative ad una risorsa;
- TRACE il server risponde *rimbalzando* un messaggio identico a quello che è stato inviato dal client.

La stessa risorsa può essere disponibile sul server in diversi formati, diverse lingue e diversi set di caratteri.

Esempio (di fantasia): elenco di Capitali nel mondo e relative coordinate

E' una tabella che può essere in CSV, HTML, XML, JSON ...

I nomi delle città possono essere in diverse lingue

...

La stessa risorsa può essere disponibile sul server in diversi formati, diverse lingue e diversi set di caratteri.

Esempio (di fantasia): elenco di Capitali nel mondo e relative coordinate

E' una tabella che può essere in CSV, HTML, XML, JSON ...

I nomi delle città possono essere in diverse lingue

...

Il protocollo HTTP permette al client di specificare nell'header della richiesta alcune preferenze. Il server risponderà tentando di soddisfarle al meglio

Client request:

```
GET /hello.txt HTTP/1.1
User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.71 zlib/1.2.3
Host: www.example.com
Accept-Language: en, mi
```

I campi che è possibile specificare nell'header della richiesta per la content negotiation sono i seguenti:

- **Accept** *media range* costruiti a partire dai media type IANA (vedi <https://www.iana.org/assignments/media-types/media-types.xhtml> )
- **Accept-Charset** set di caratteri (vedi <http://www.iana.org/assignments/character-sets/character-sets.xhtml> )
- **Accept-Encoding** uno di compress, deflate, gzip o identity
- **Accept-Language** lingue (vedi <https://www.iana.org/assignments/language-tags/language-tags.xhtml> per l'elenco dei language tags)

Come visto prima, il server invia nella risposta uno *status code*, che è un codice a 3 cifre decimali.

La prima cifra indica la *categoria*:

- 1xx (Informational): la richiesta è stata ricevuta, continuo il processamento;
- 2xx (Successful): la richiesta è stata ricevuta, accettata e processata con successo;
- 3xx (Redirection): ulteriori operazioni vengono richieste al client per completare la richiesta;
- 4xx (Client Error): la richiesta è malformata o non può essere eseguita;
- 5xx (Server Error): qualche problema interno al server ha impedito di soddisfare la richiesta.

Un *Web Server* è un programma che permette di fornire risorse ai client attraverso il protocollo HTTP.

Alcuni Web Server sono *Apache* (<https://httpd.apache.org/>), IIS (<https://www.iis.net/>), Jetty (<http://www.eclipse.org/jetty/>)

Nel suo funzionamento di base un web server fornisce al client i files presenti all'interno della sua directory root (il path che il client indica equivale al path del file relativamente alla directory root),

Un web server può implementare funzionalità per offrire lo stesso file in diverse lingue (normalmente si postpone il language tag della lingua al nome del file).

Funzionalità avanzate riguardano il processamento di pagine generate server side con linguaggi di scripting (php, asp, jsp).

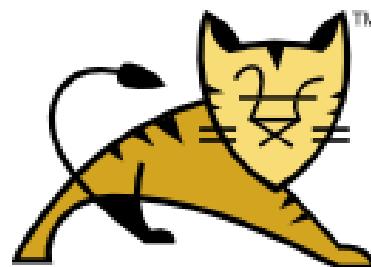
Nel framework java il meccanismo più basilare per la realizzazione di applicazioni web sono le *servlet*.

Per mettere in esercizio una Servlet è necessario installare un Servlet Container che altro non è che un web server che offre in più il supporto per il deploy e la gestione del ciclo di vita delle servlet.

Nel framework java il meccanismo più basilare per la realizzazione di applicazioni web sono le *servlet*.

Per mettere in esercizio una Servlet è necessario installare un Servlet Container che altro non è che un web server che offre in più il supporto per il deploy e la gestione del ciclo di vita delle servlet.

Alcuni web server sono Jetty e Apache Tomcat.



Entrambi possono essere utilizzati direttamente in eclipse in fase di sviluppo. Per Jetty è stato realizzato un plugin di eclipse apposito

[https://wiki.eclipse.org/Jetty\\_WTP\\_Plugin](https://wiki.eclipse.org/Jetty_WTP_Plugin)

Vedi <https://docs.oracle.com/javaee/6/tutorial/doc/bnaf.html>

