

Elementi di Object Oriented e Web Programming

**Cristiano Longo, 2016
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”.

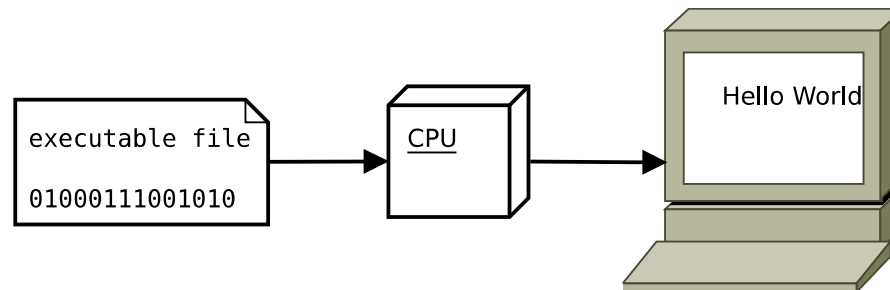
Altrimenti, per ogni parametro:

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

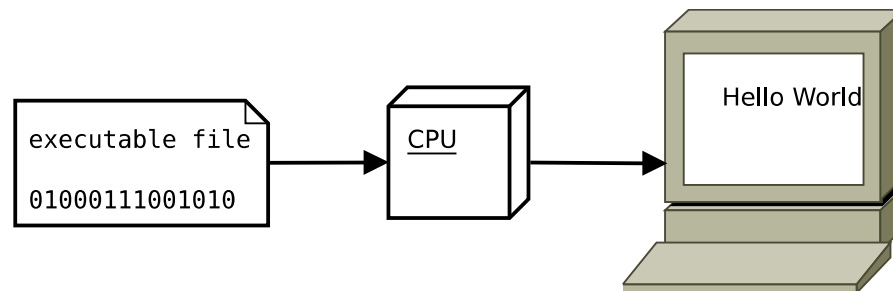
Vedi

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

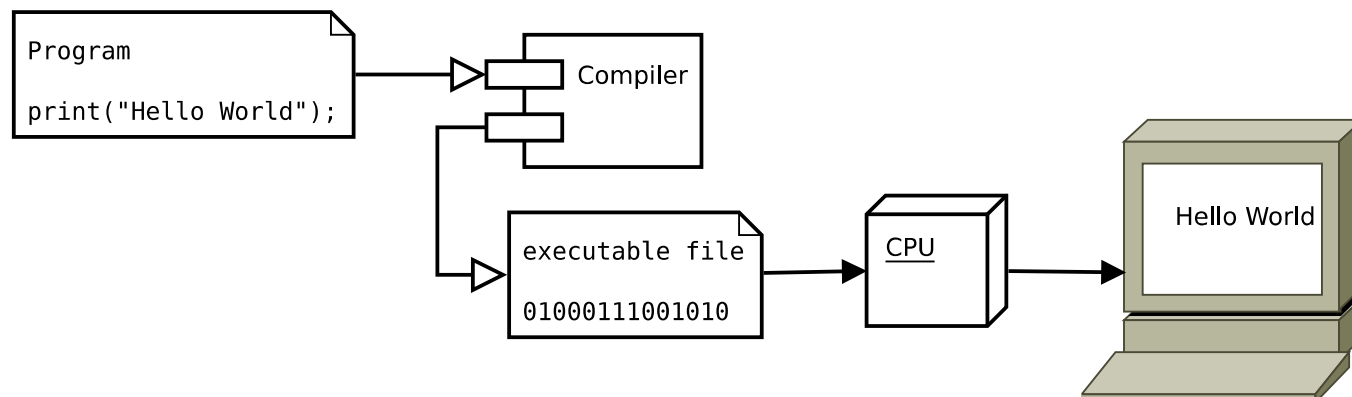
Le unità di processamento in un computer eseguono sequenze di istruzioni scritte in linguaggio specifico per l'architettura (ad. Esempio Assembly x86), chiamati **linguaggi macchina**. I file contenenti le sequenze di istruzioni sono detti *eseguibili*.



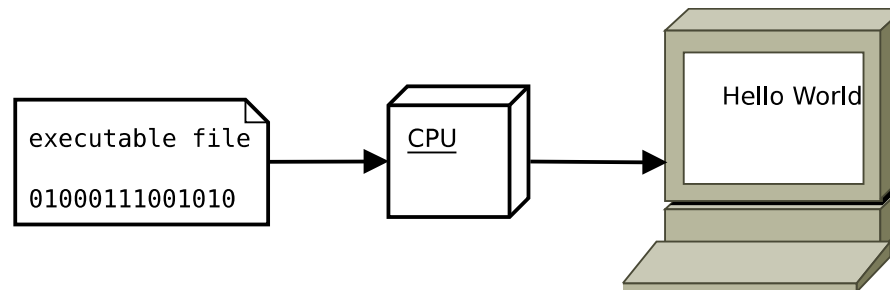
Le unità di processamento in un computer eseguono sequenze di istruzioni scritte in linguaggio specifico per l'architettura (ad. Esempio Assembly x86), chiamati **linguaggi macchina**. I file contenenti le sequenze di istruzioni sono detti *eseguibili*.



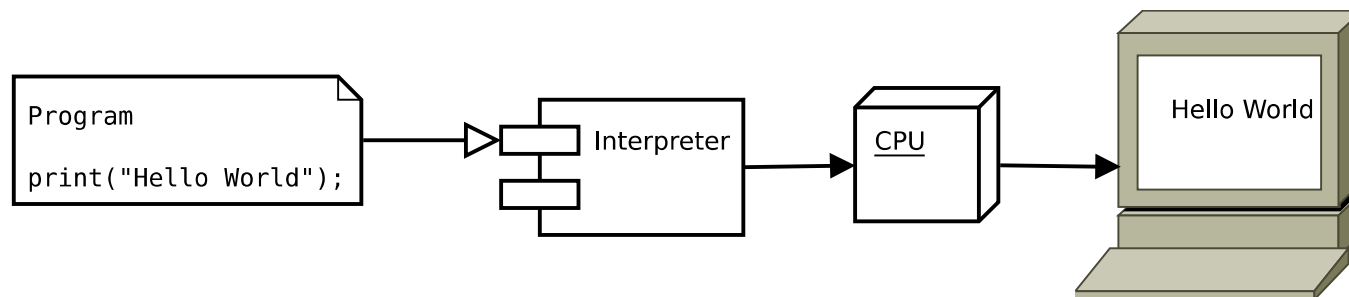
Un **compilatore** è un eseguibile che traduce un programma scritto in un linguaggio di *alto livello* in un eseguibile in linguaggio macchina.



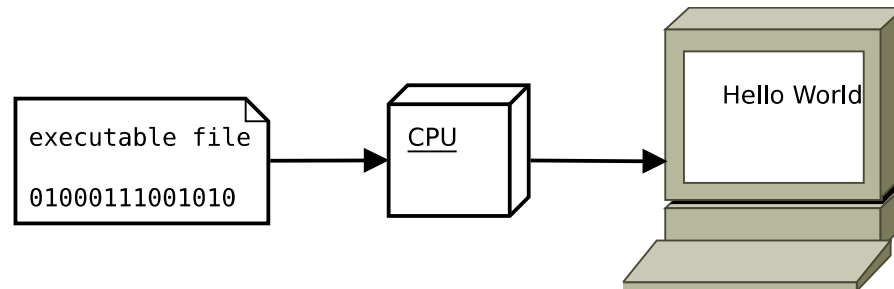
Le unità di processamento in un computer eseguono sequenze di istruzioni scritte in linguaggio specifico per l'architettura (ad. Esempio Assembly x86), chiamati **linguaggi macchina**. I file contenenti le sequenze di istruzioni sono detti *eseguibili*.



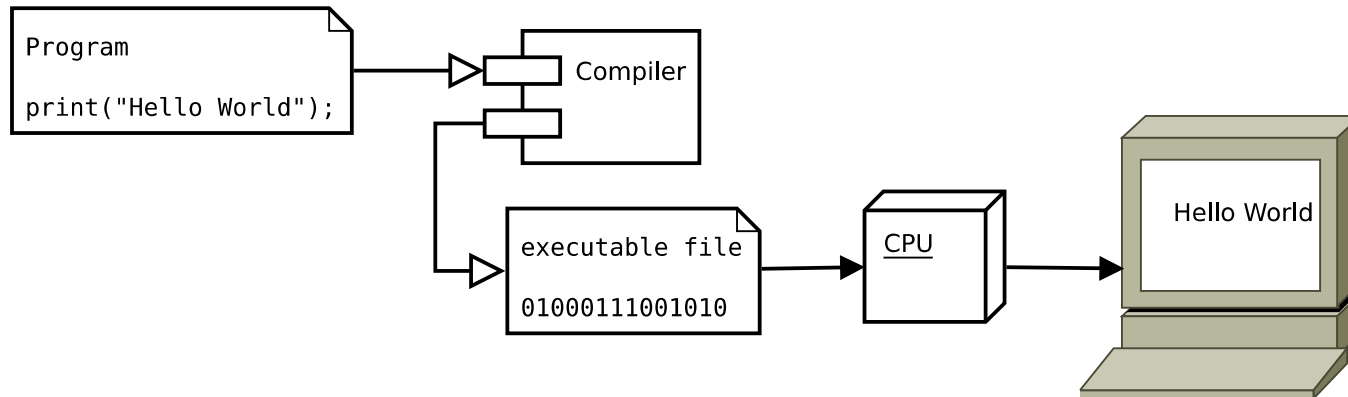
Un **interprete** è un programma eseguibile che interpreta ed *esegue* un programma scritto in un linguaggio di *alto livello*



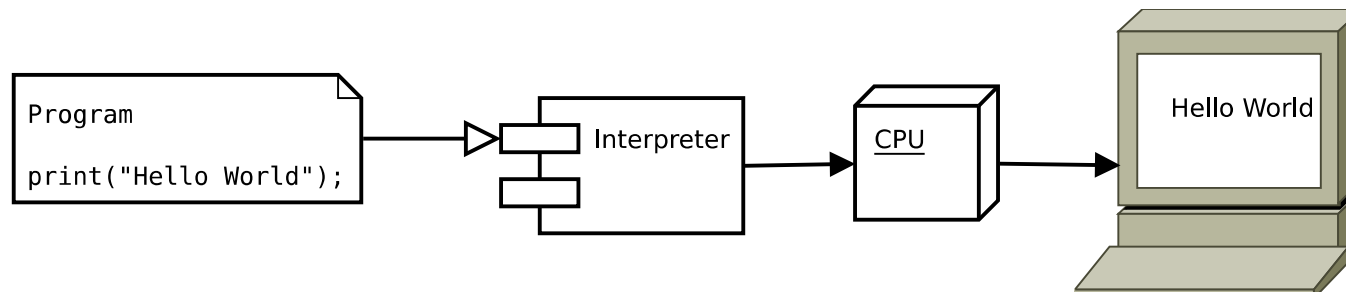
File eseguibili



Programmi Compilati

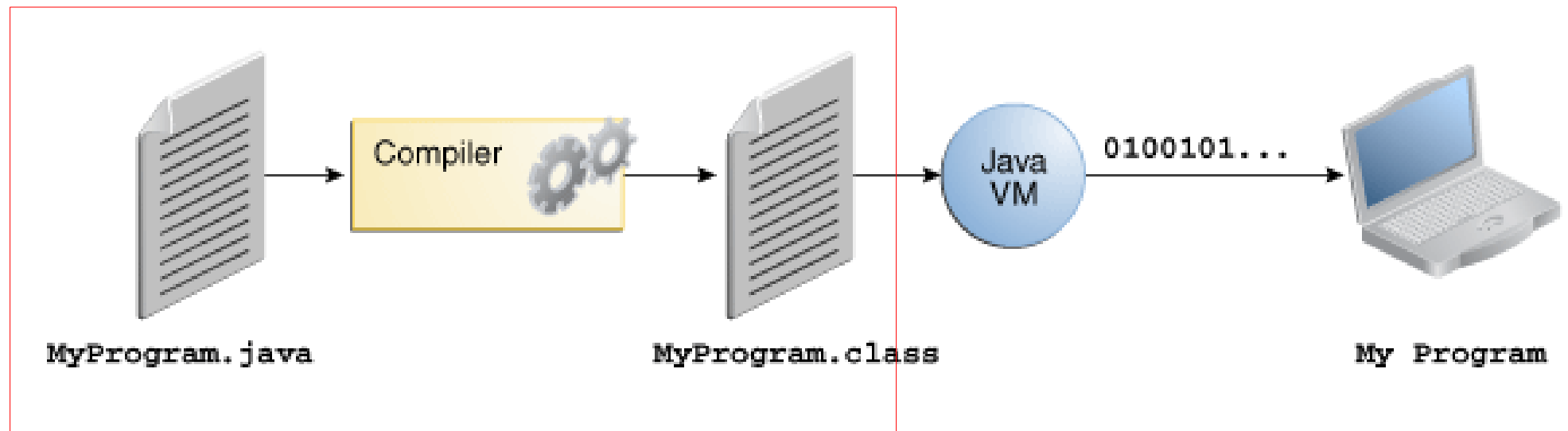


Programmi Interpretati



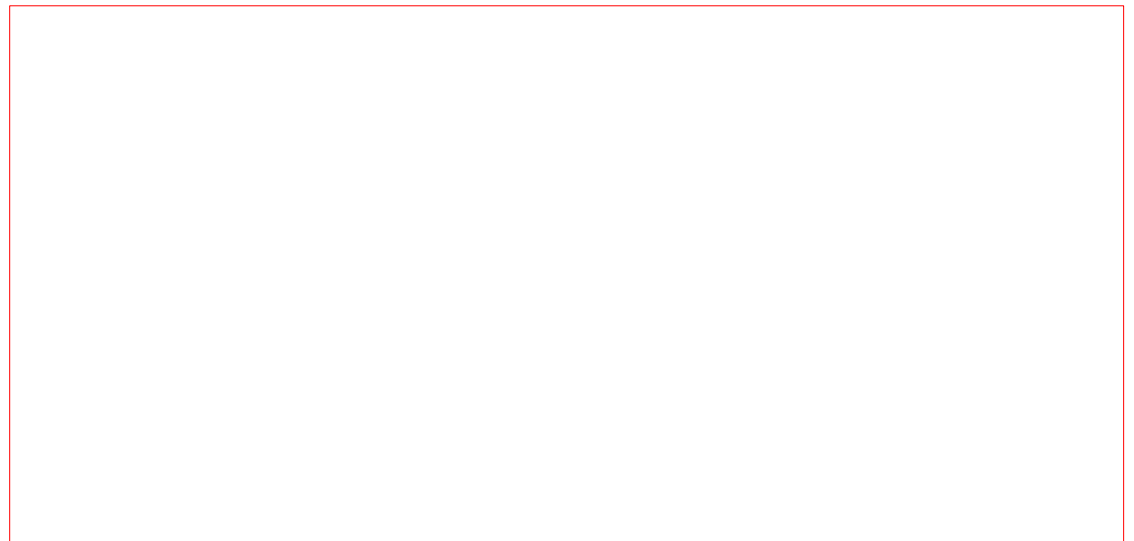
L'approccio delle tecnologie Java è *ibrido*:

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



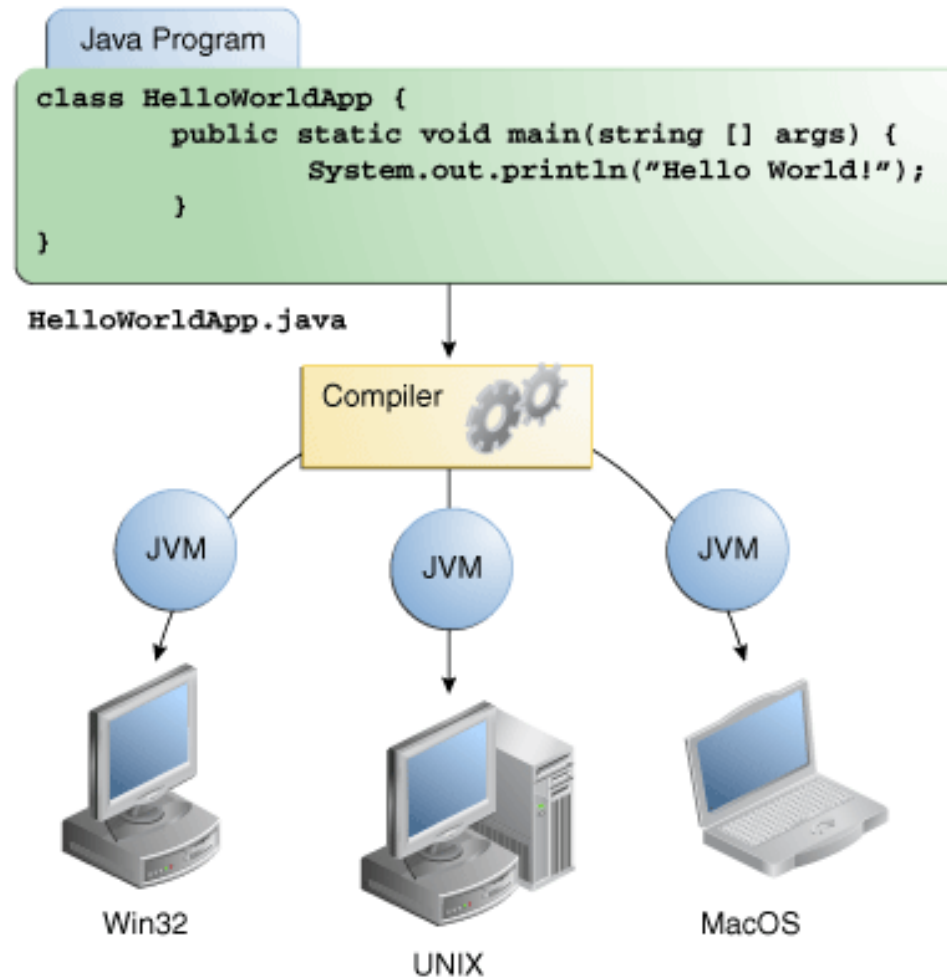
L'approccio delle tecnologie Java è *ibrido*:

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



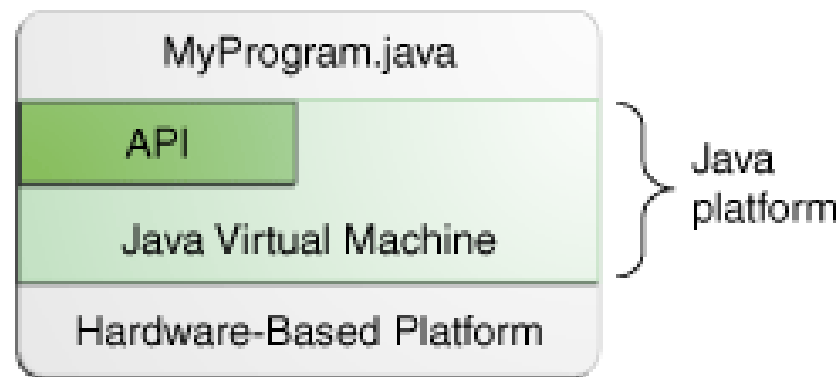
La JVM è disponibile per diversi sistemi operativi

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

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.

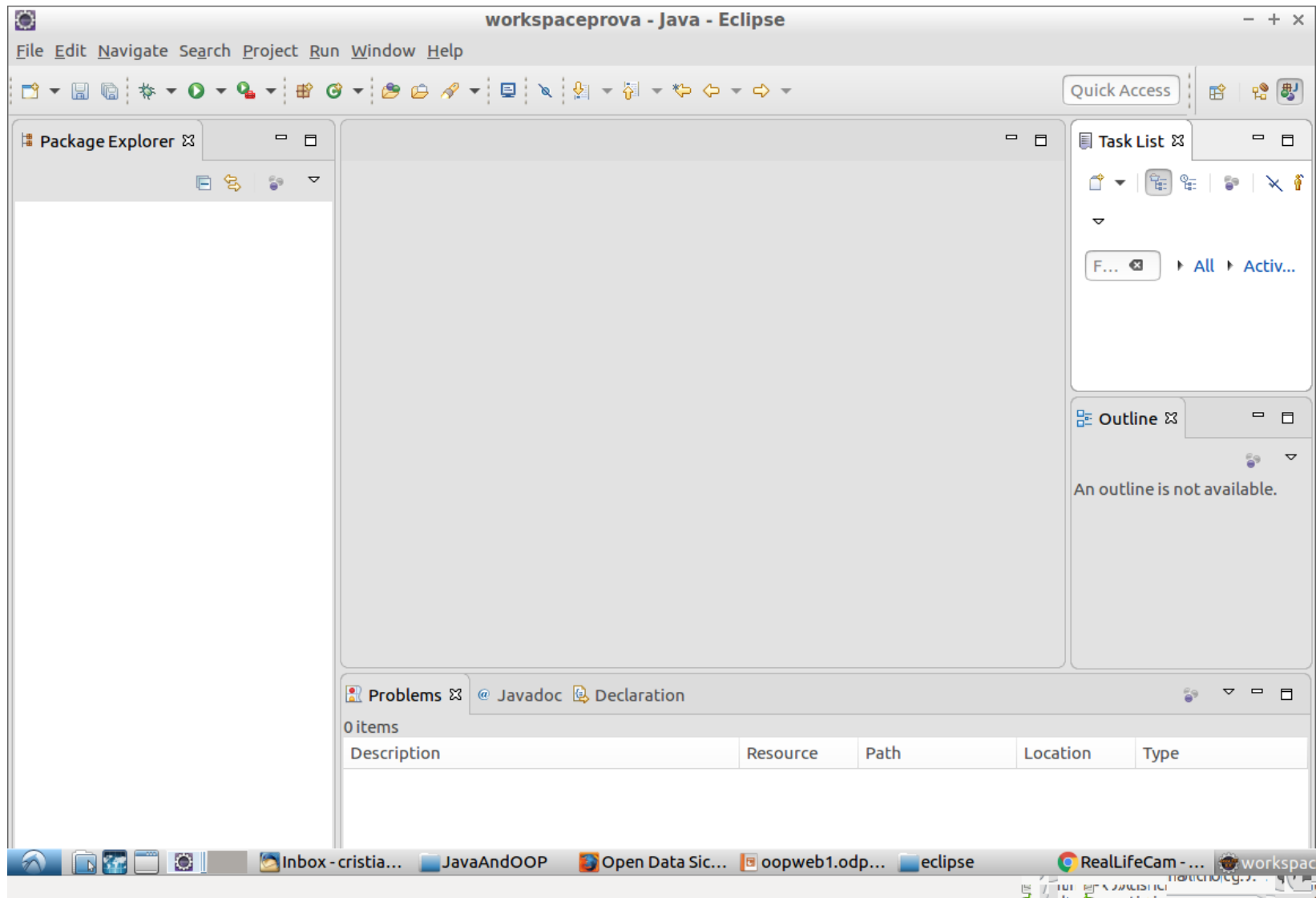
I progetti veri e propri possono essere posizionati in qualsiasi locazione del filesystem, non necessariamente all'interno del workspace.

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

Particolarmente rilevante tra la 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.

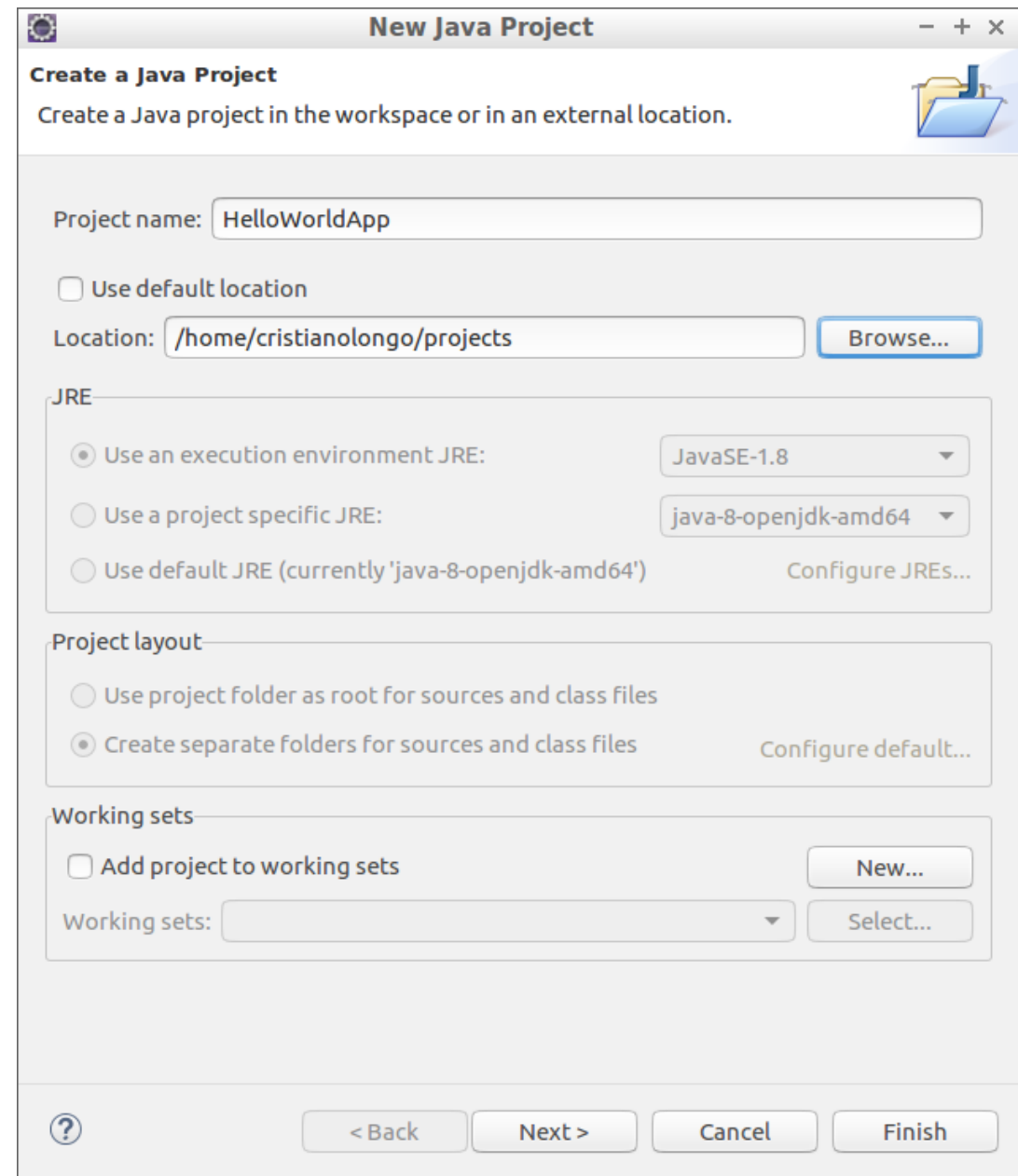
La *perspective* Java (menù – window – perspectives) si presenta con un pannello a sinistra per i progetti, un *editor* al centro e varie altre *view* e *tabs* a destra e in basso.



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

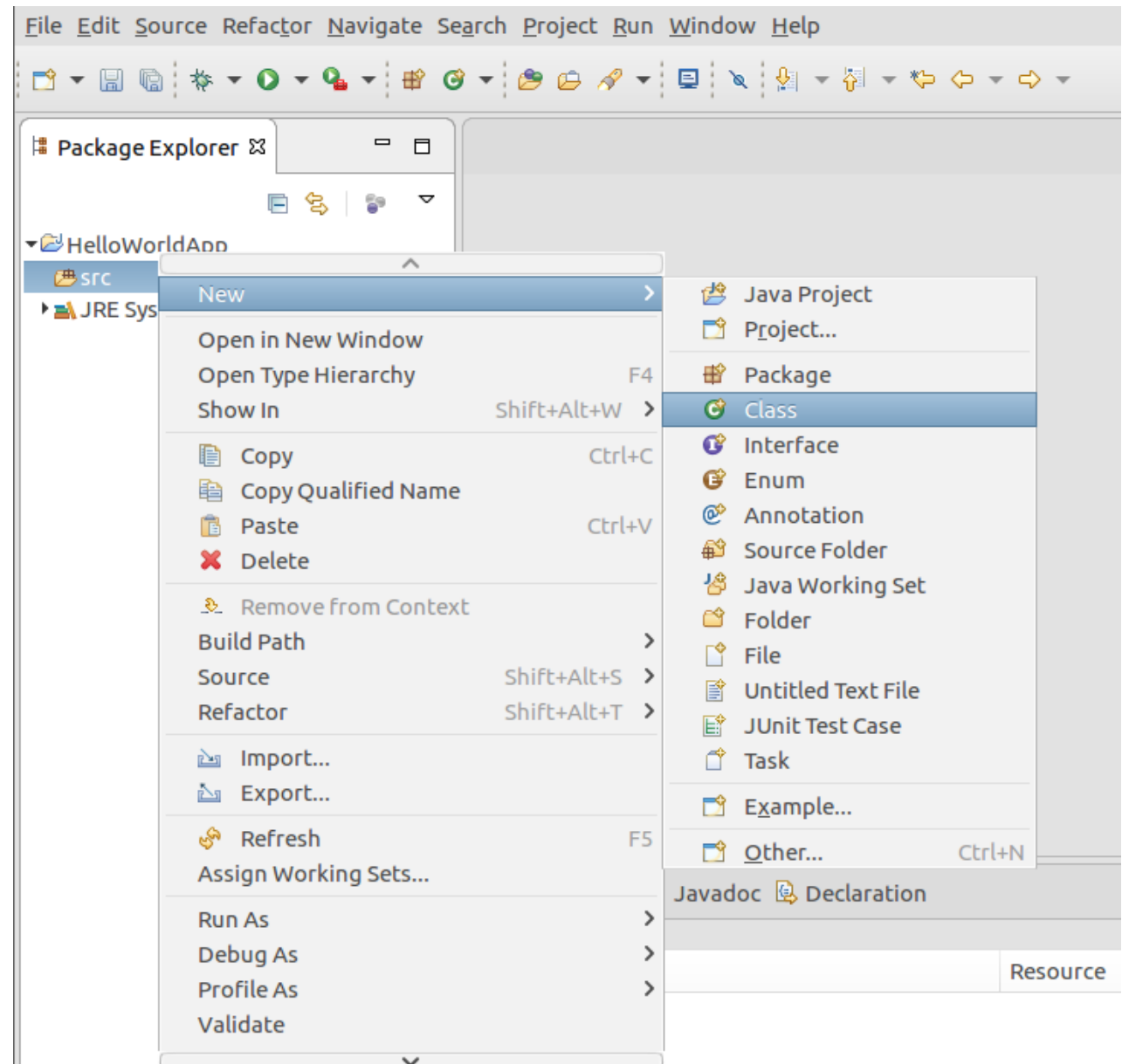
Il campo location permette di specificare una posizione esterna al workspace.

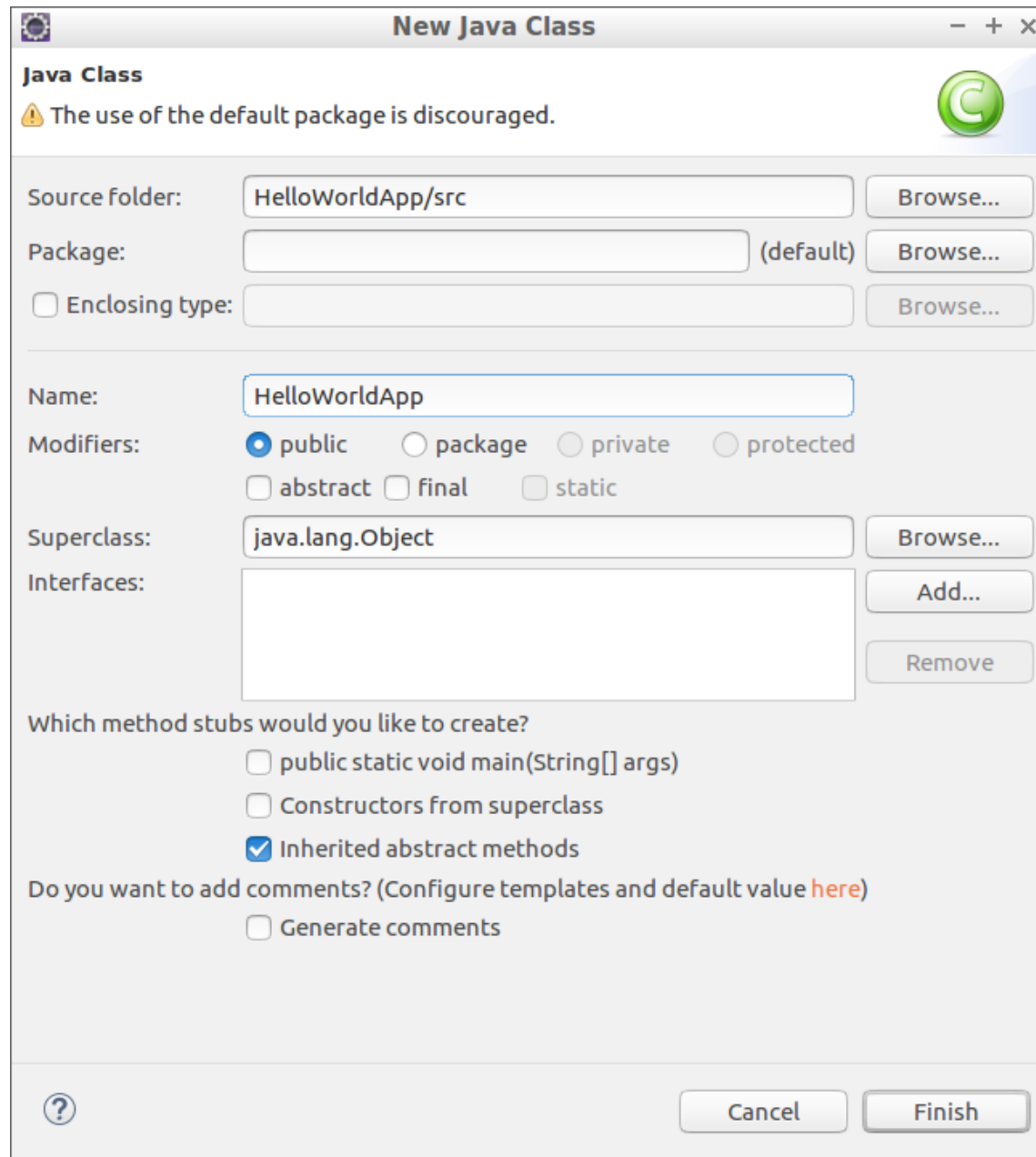
E' possibile specificare la JRE nel caso in cui ne siano disponibili più di una (questo parametro si può cambiare successivamente)



Il progetto sarà visibile nel pannello di sinistra.

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



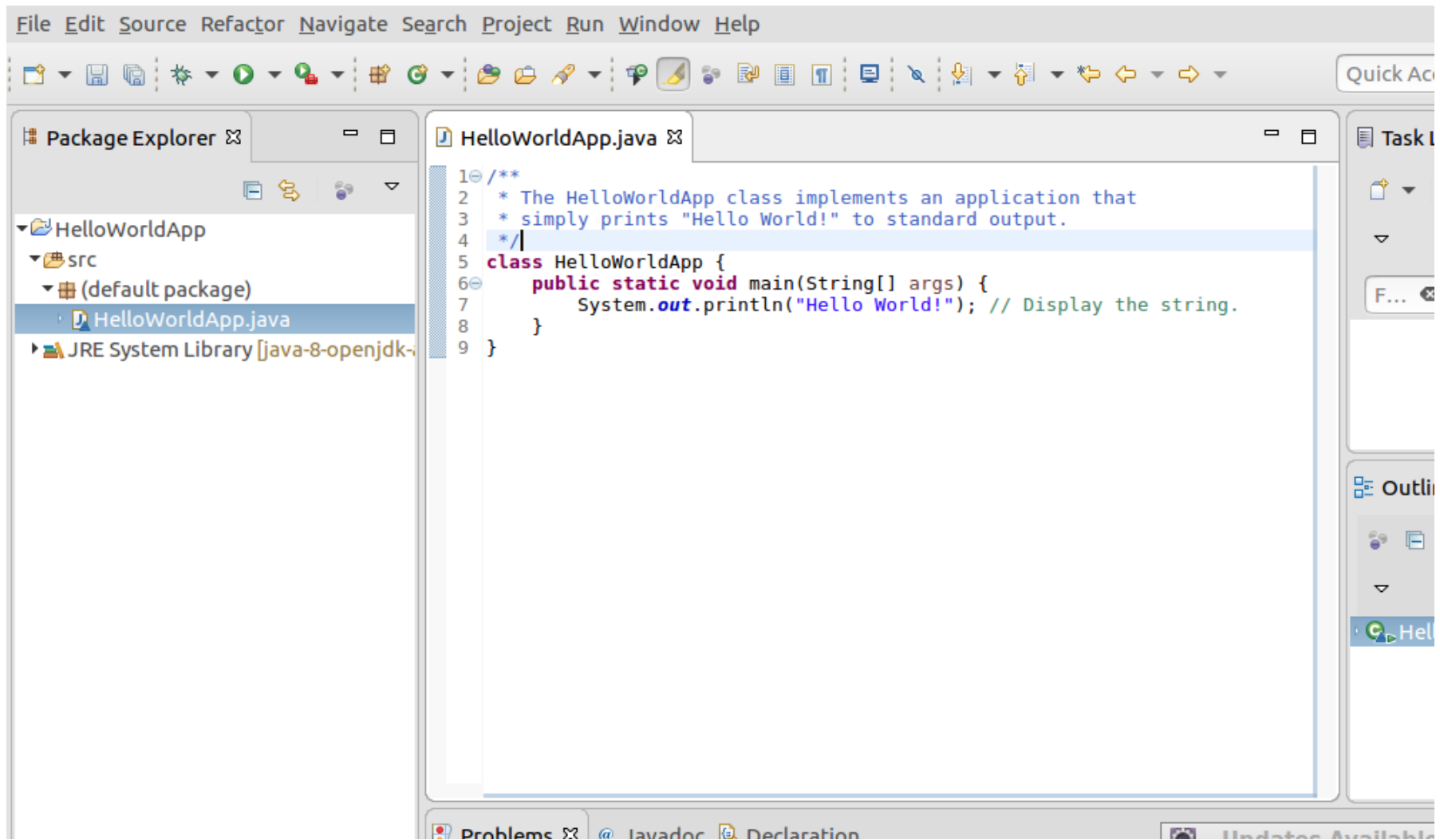


The screenshot shows the 'New Java Class' dialog box in the Eclipse IDE. The dialog has a title bar with the Eclipse logo and window controls. Below the title bar, there's a section labeled 'Java Class' with a warning icon and the text 'The use of the default package is discouraged.' and a green 'C' icon. The main area contains several input fields and buttons:

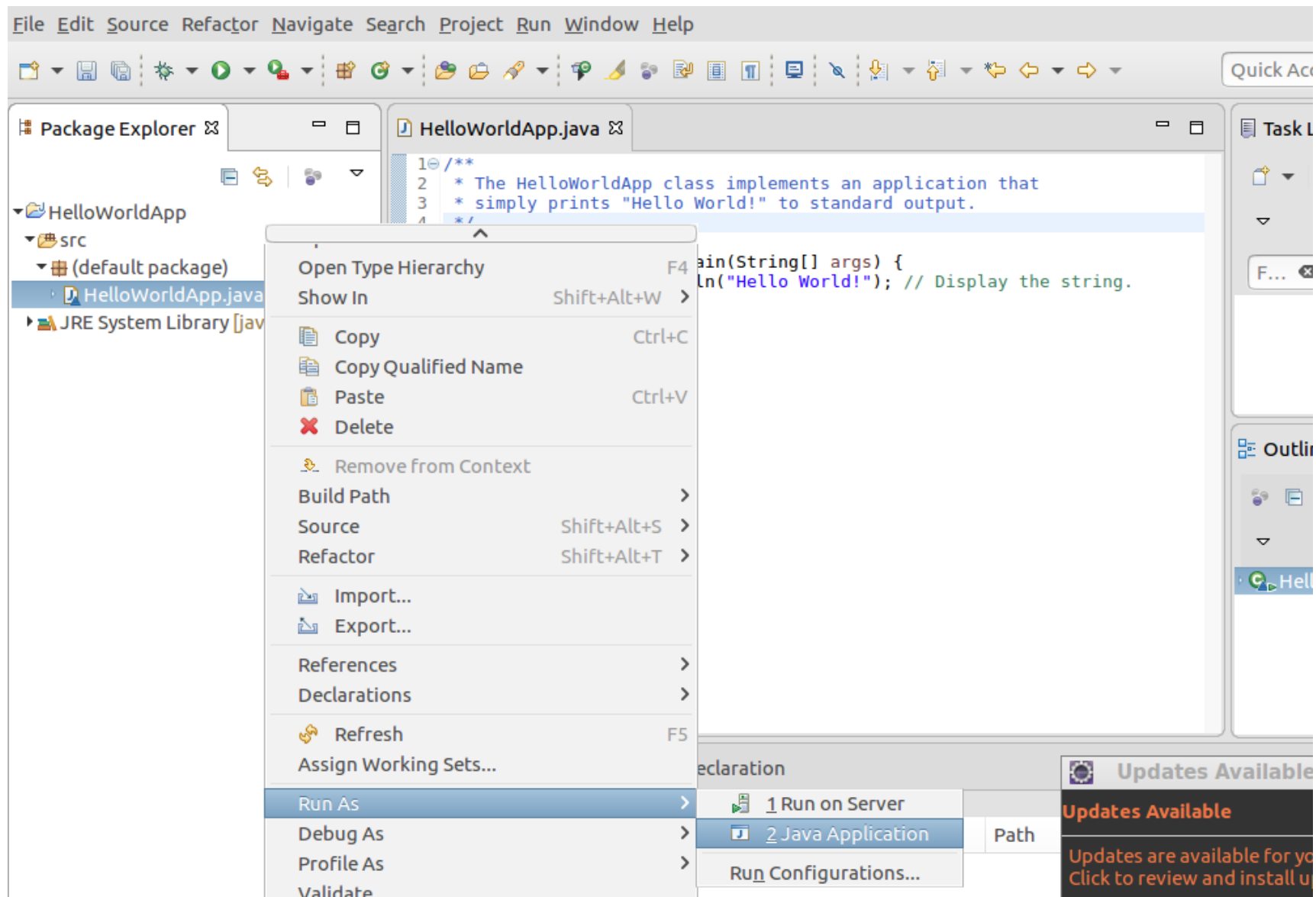
- Source folder:** A text field containing 'HelloWorldApp/src' and a 'Browse...' button.
- Package:** A text field (empty) followed by '(default)' and a 'Browse...' button.
- Enclosing type:** A checkbox labeled 'Enclosing type:' followed by an empty text field and a 'Browse...' button.
- Name:** A text field containing 'HelloWorldApp'.
- Modifiers:** A group of radio buttons for 'public' (selected), 'package', 'private', and 'protected'. Below them are checkboxes for 'abstract', 'final', and 'static'.
- Superclass:** A text field containing 'java.lang.Object' and a 'Browse...' button.
- Interfaces:** An empty text area with 'Add...' and 'Remove' buttons.
- Which method stubs would you like to create?** A section with three checkboxes: 'public static void main(String[] args)' (unchecked), 'Constructors from superclass' (unchecked), and 'Inherited abstract methods' (checked).
- Do you want to add comments?** A section with the text '(Configure templates and default value [here](#))' and a checkbox for 'Generate comments' (unchecked).

At the bottom, there is a question mark icon, a 'Cancel' button, and a 'Finish' button.

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
- <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 $\langle \text{value}_i \rangle$ è vero, per qualche i compreso tra 1 ed n , esegue $\langle \text{body}_i \rangle$, $\langle \text{body}_{(i+1)} \rangle$, ..., $\langle \text{body}_n \rangle$ e $\langle \text{bodyDefault} \rangle$, se presente. Altrimenti esegue solo $\langle \text{bodyDefault} \rangle$, 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”

Vedi

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

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));
```

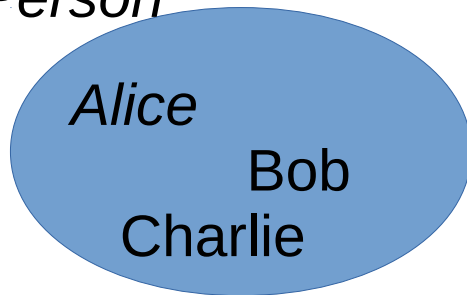
Java è *class-based*: gli oggetti vengono suddivisi in *Classi*. Ogni classe definisce gli attributi e i metodi di tutte le proprie *istanze* (oggetti).

```
/**
 * A class with attributes and methods.
 *
 * @author Cristiano Longo
 */
public class ExampleClass {
    int a = 1; //an attribute

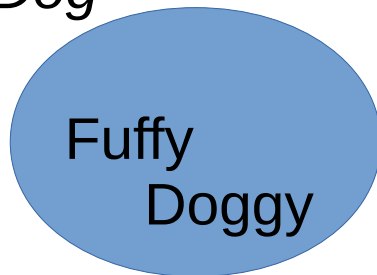
    /**
     * Increment the value of the attribute.
     */
    void inc() {
        ++a;
    }
}
```

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.

Person



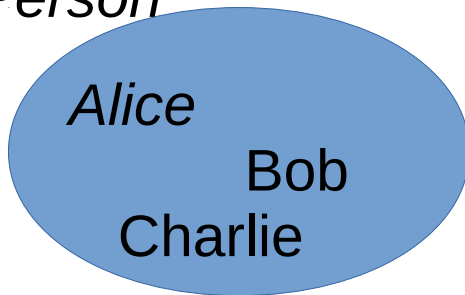
Dog



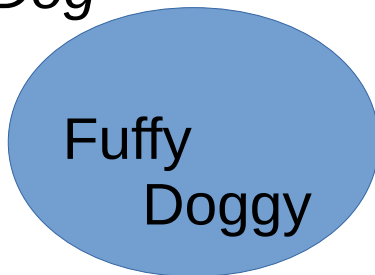
Intuitivamente, le classi rappresentano insiemi di oggetti (concetti) del mondo reale mentre le istanze di una classe rappresentano gli oggetti veri e propri nell'insieme. Le classi sono *tipi* a tutti gli effetti.

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

Person



Dog



```
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;  
    }  
}
```

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 incremeted "+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*).

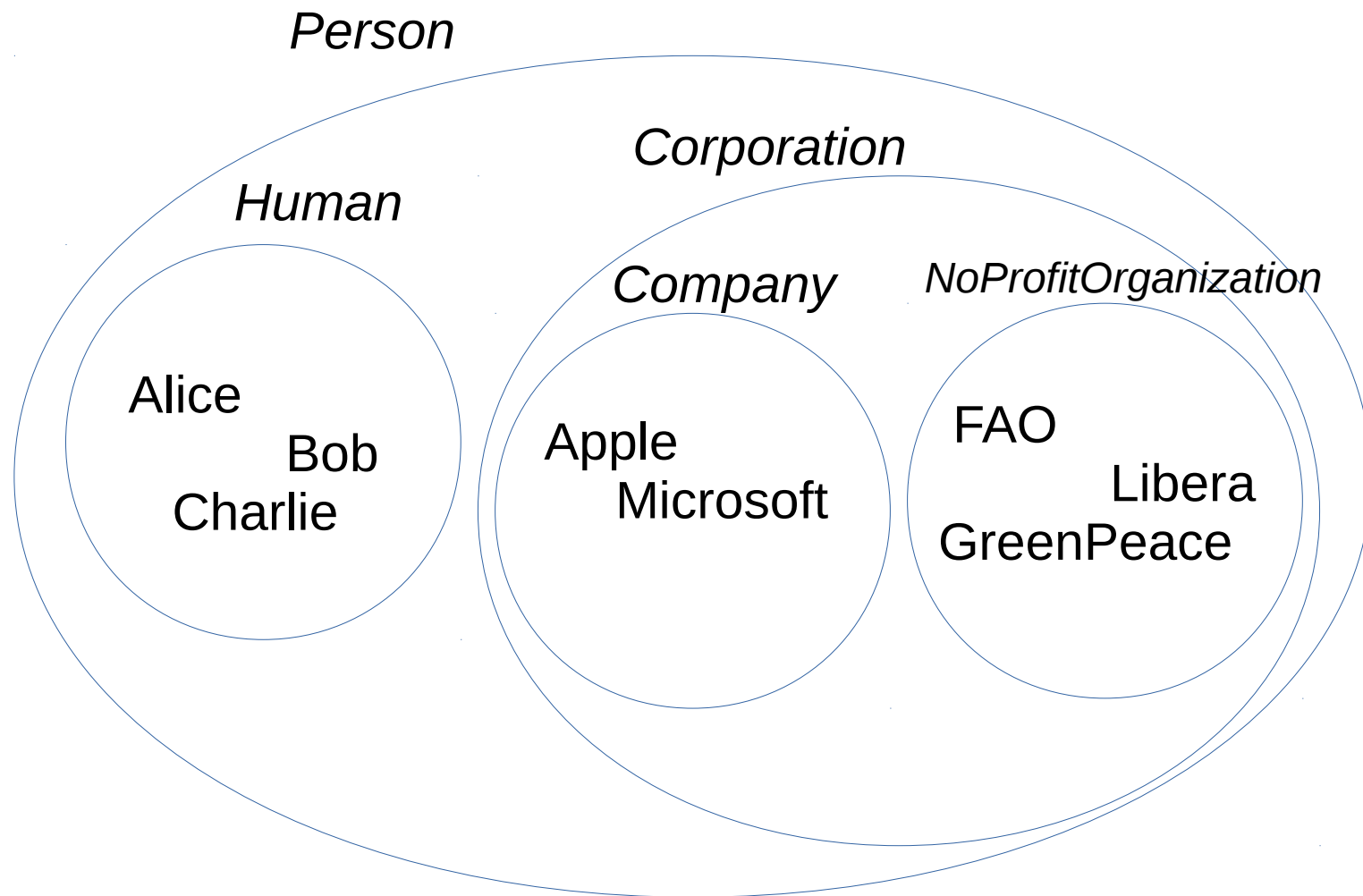
Gli attributi *statici* sono collegati direttamente alla classe. Di conseguenza hanno lo stesso valore in tutte le istanze della stessa classe.

```
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.

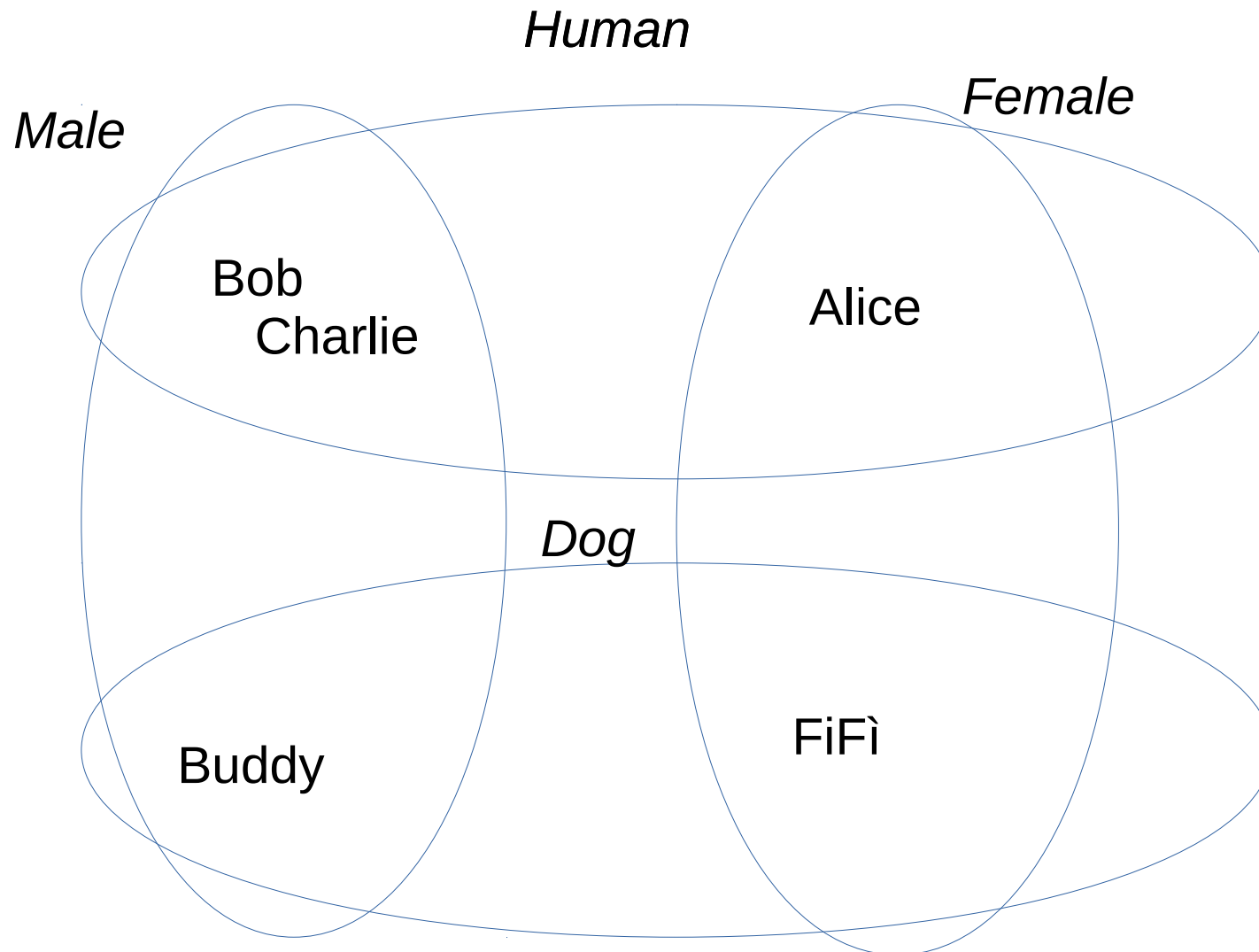
```
public class Person {
    protected String cf;

    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 interfacce 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;  
    }  
}
```

Vedi <http://docs.oracle.com/javase/tutorial/essential/exceptions/> .

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 `RstuntimeException`, che non devono essere necessariamente gestite.

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.

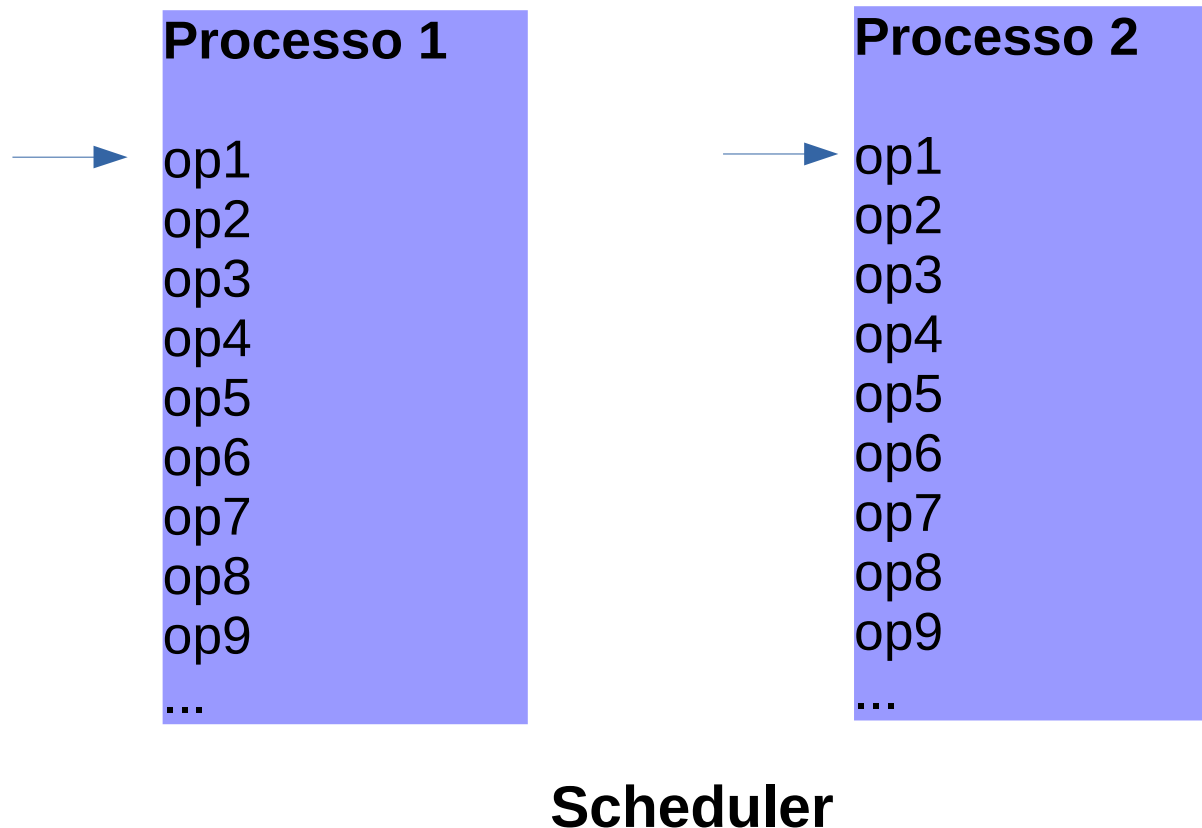
```
...
Counter c=new Counter();
try{
    c.inc(Integer.parseInt(args[0]));
}catch(final NegativeNumberException e){
    e.printStackTrace();
}catch(final ParseException e){
    e.printStackTrace();
} finally {
    System.out.println(c.value);
}
```

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.

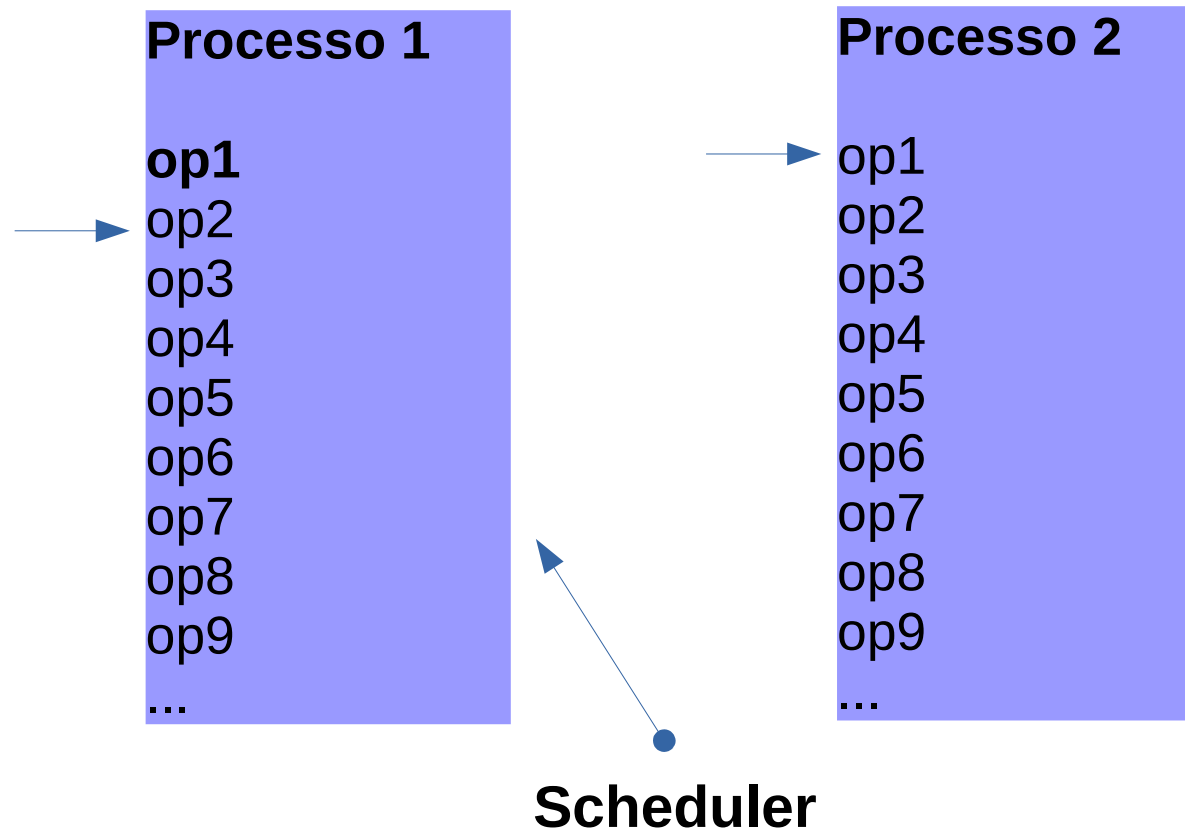
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

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.

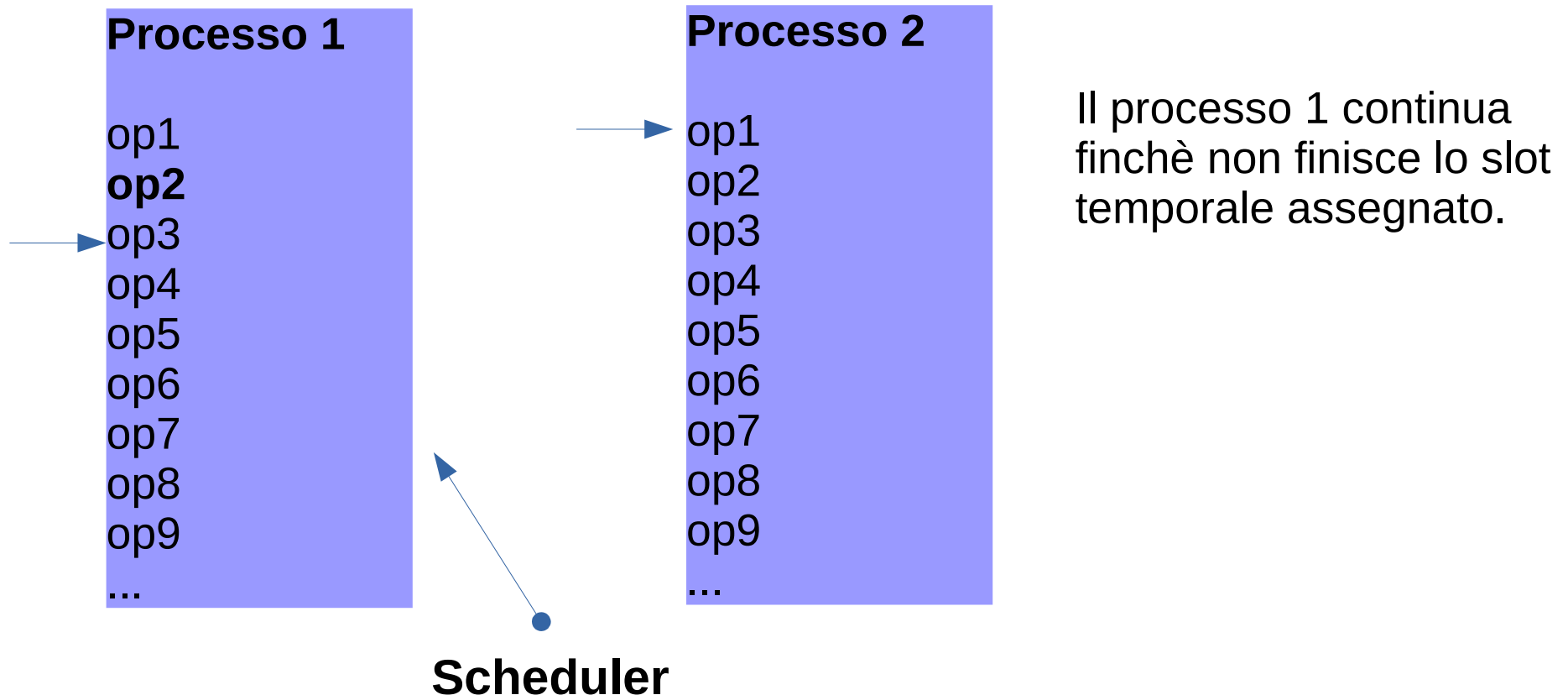
L'esecuzione è governata da uno **scheduler**.



Lo scheduler seleziona il processo 1, che esegue una operazione ed avanza la *testina*.

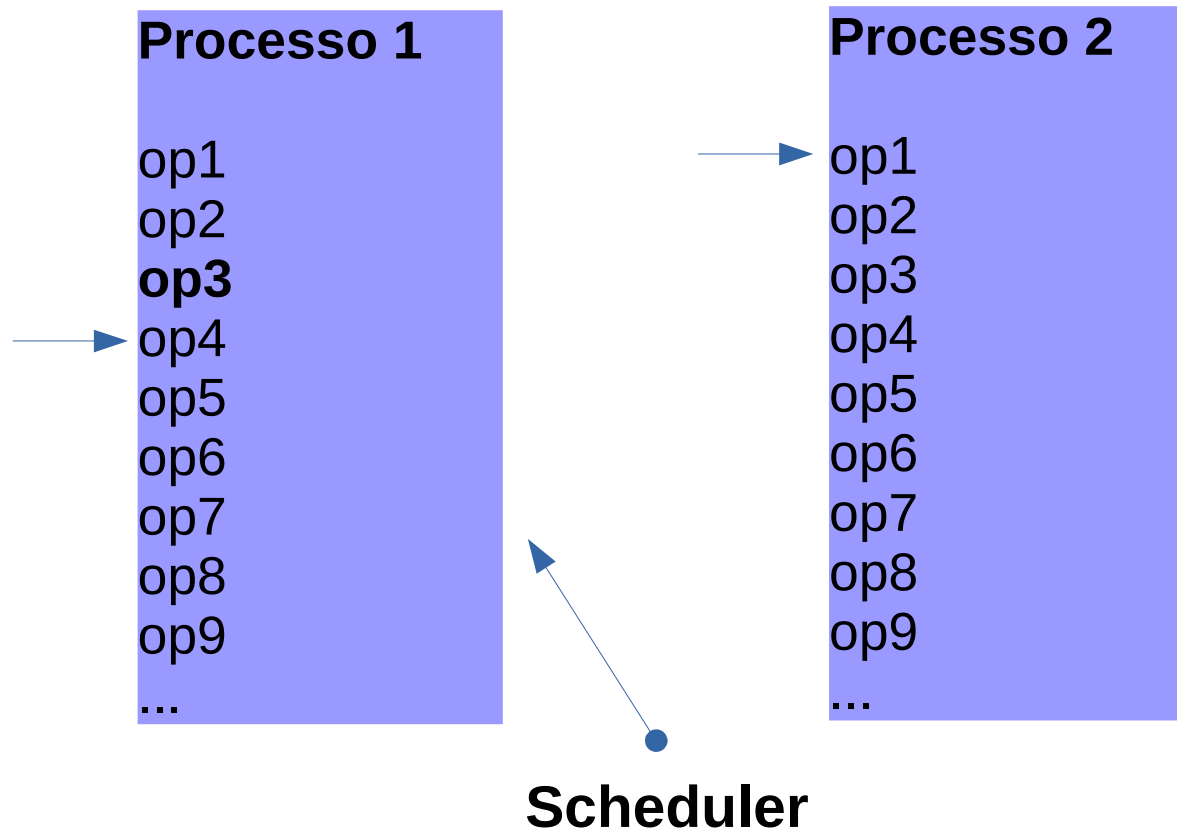
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

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



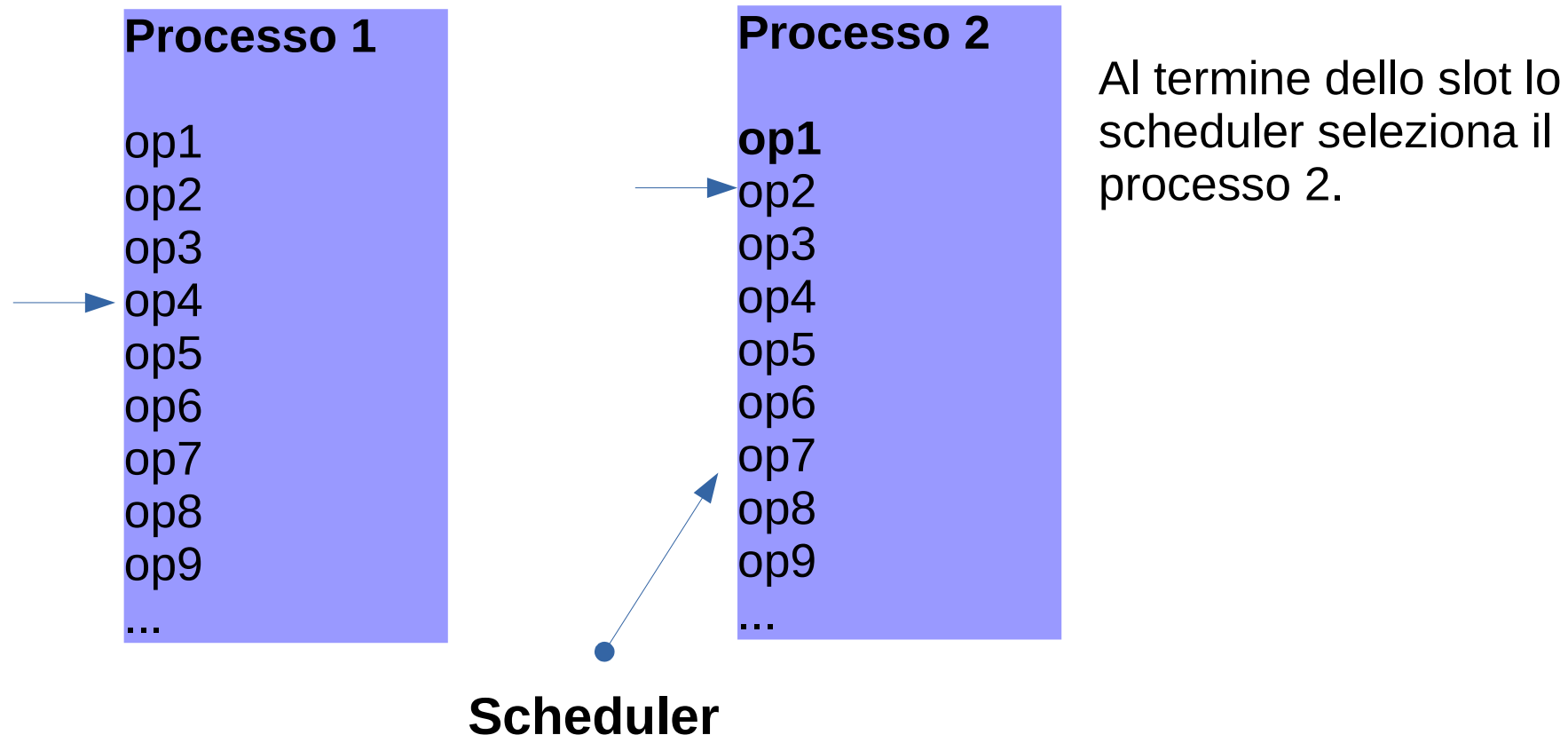
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

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



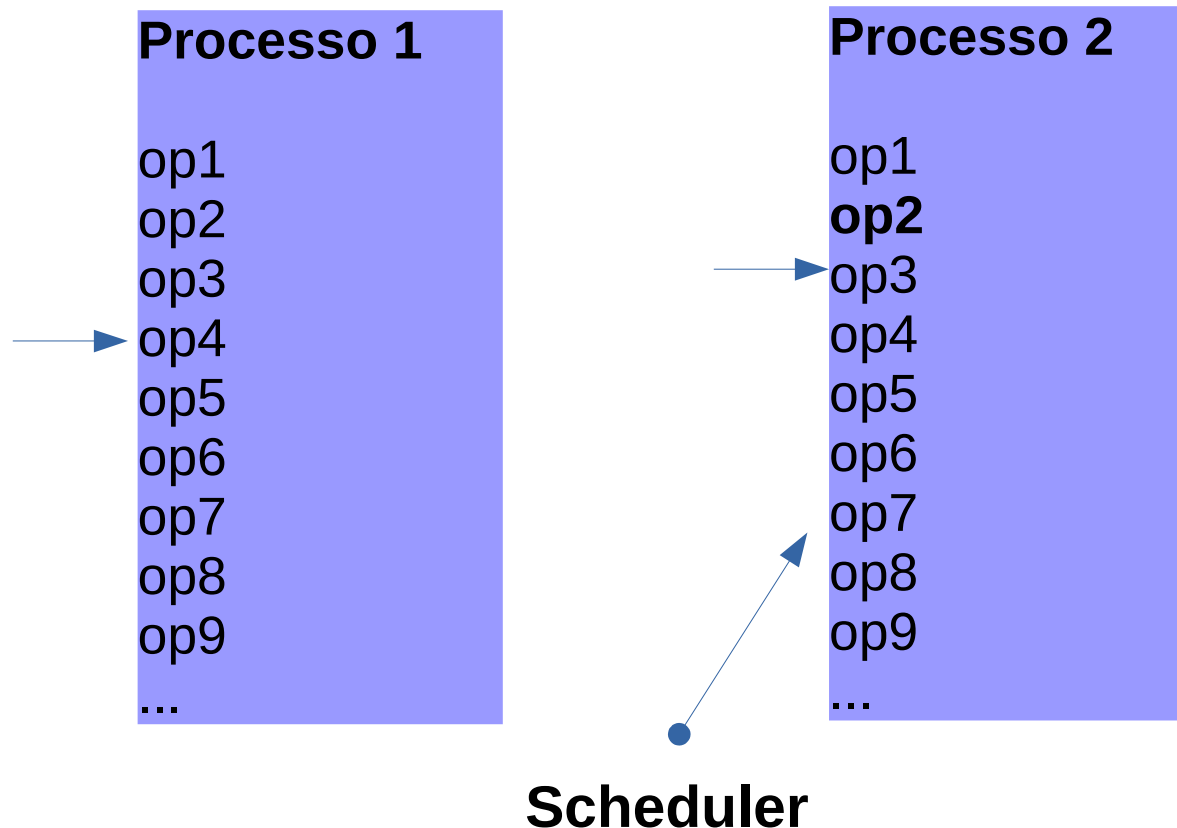
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

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



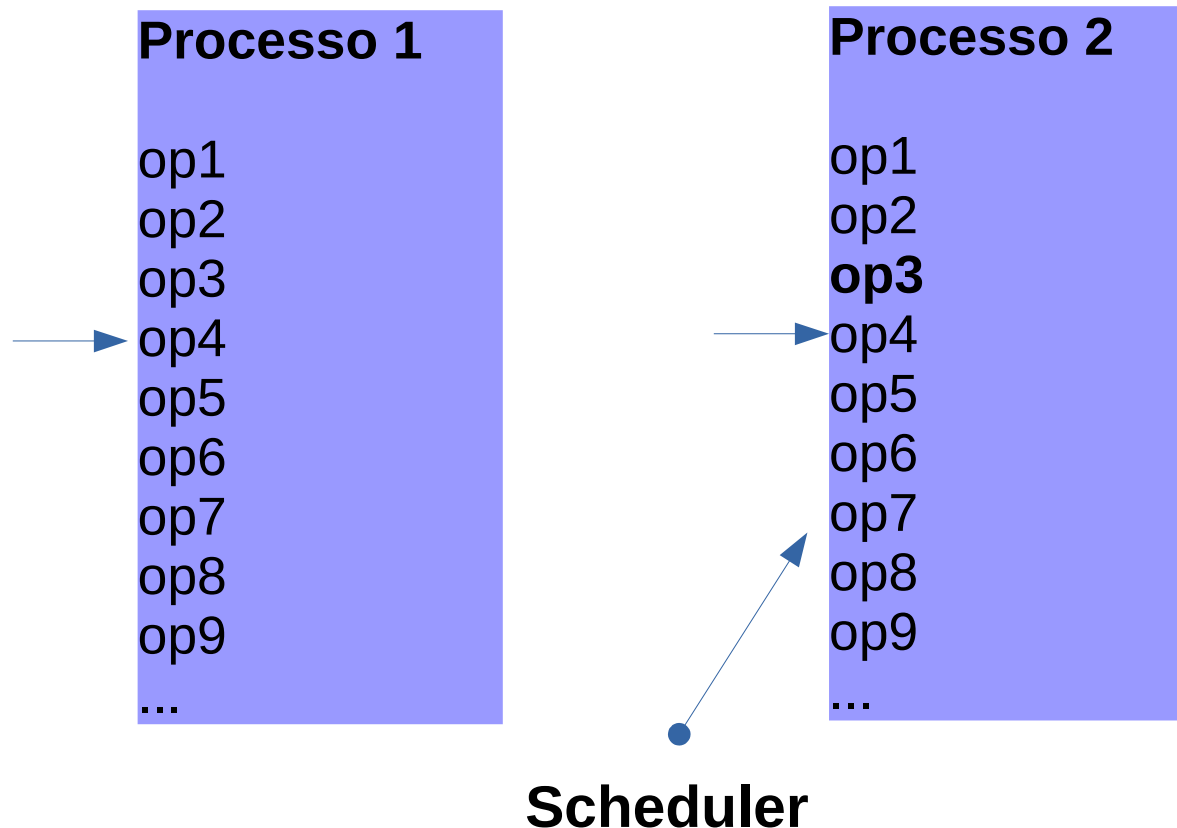
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

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



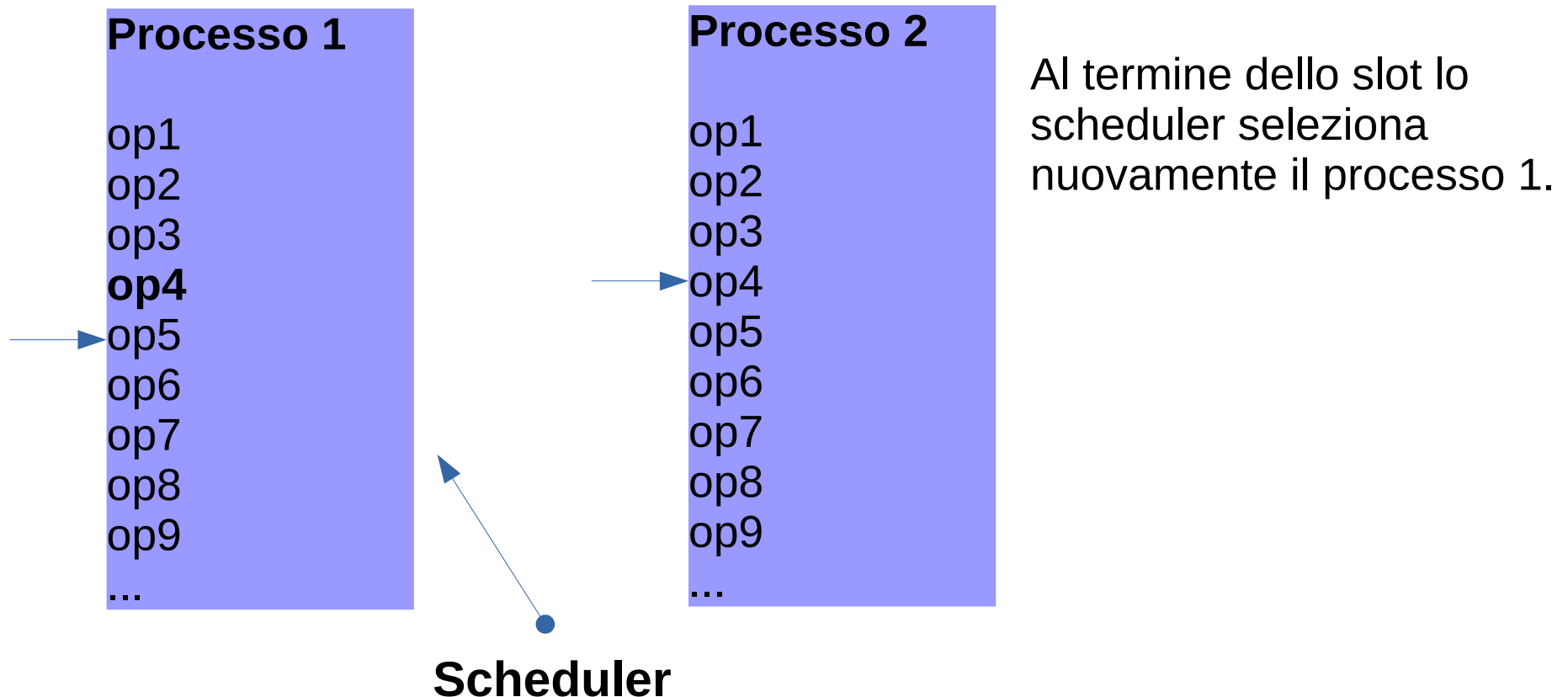
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

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



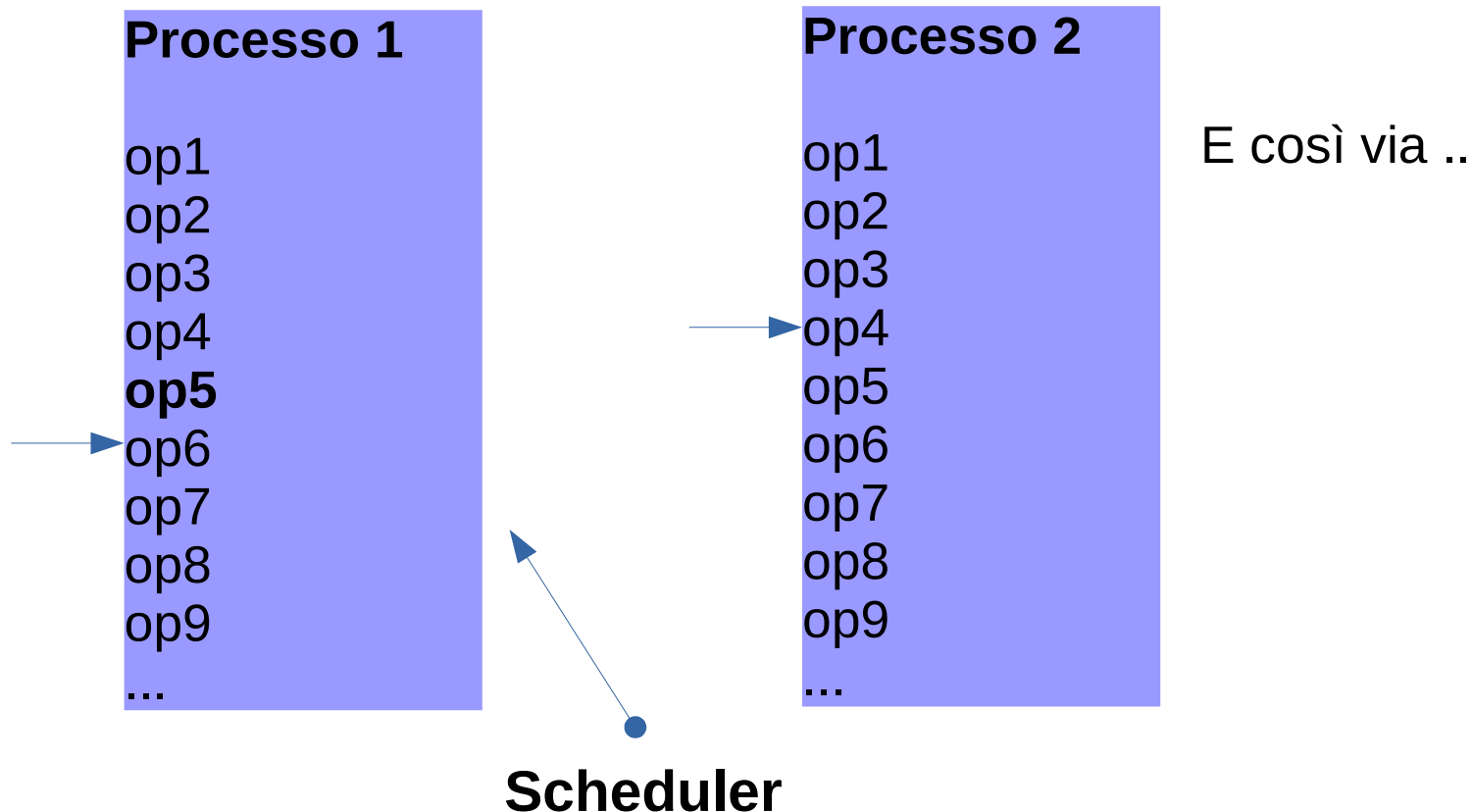
Utilizzando una unica unità di processamento (CPU) il multitasking deve essere *simulato*: il tempo viene suddiviso in *slot*; ogni slot è dedicato ad uno solo dei *processi* in esecuzione; al termine dello slot il task sospende la sua esecuzione e lo slot successivo viene assegnato ad un altro task.

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.

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

Disco	CPU	Stampante
p1		

p1

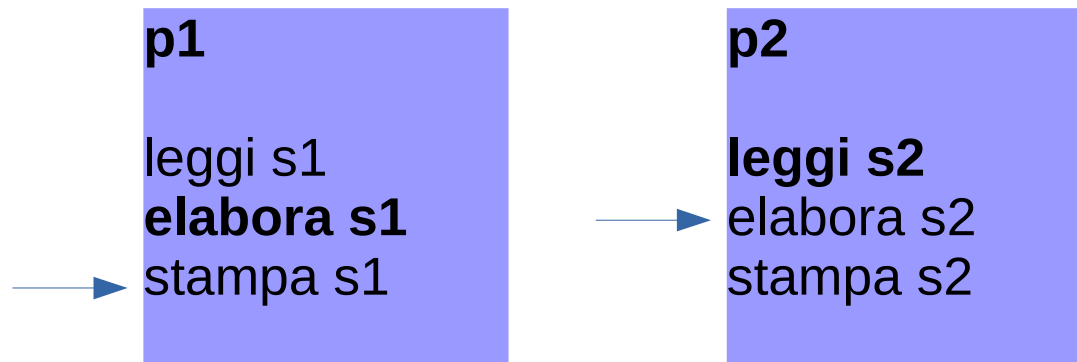
leggi s1

elabora s1

stampa s1

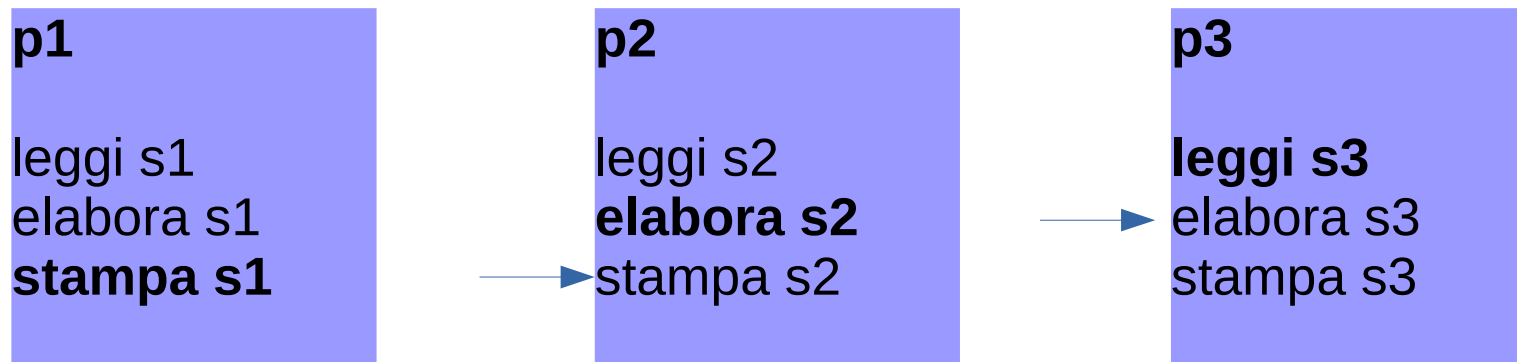
Terminata la lettura di s1 si procede alla sua elaborazione. Nel frattempo il disco è libero e può essere iniziata la lettura di s2.

Disco	CPU	Stampante
p2	p1	



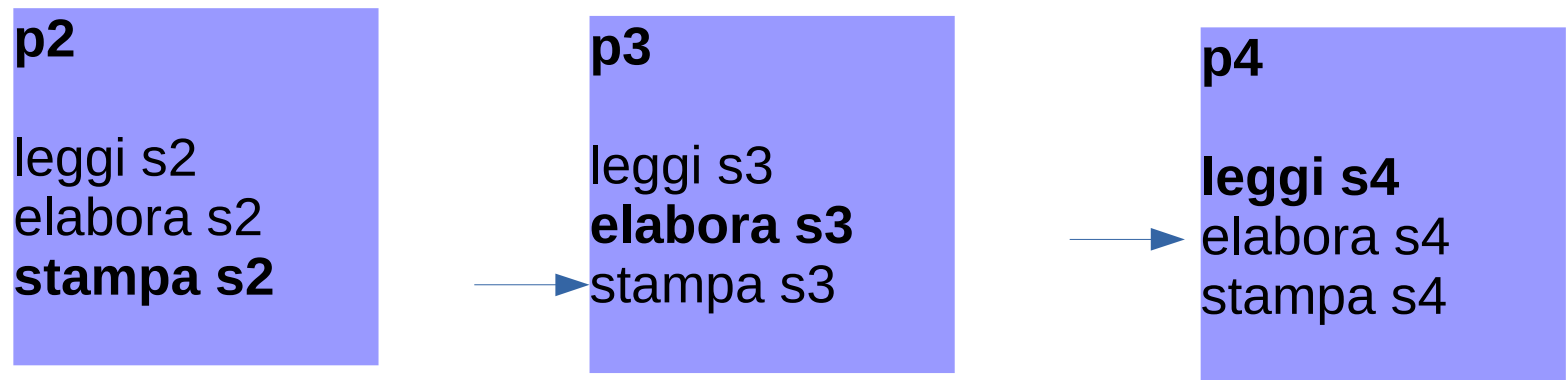
Terminata la elaborazione di s1 si può mandarlo in stampa. Nel frattempo può essere avviata l'elaborazione di s2 e la lettura di s3.

Disco	CPU	Stampante
p3	p2	p1

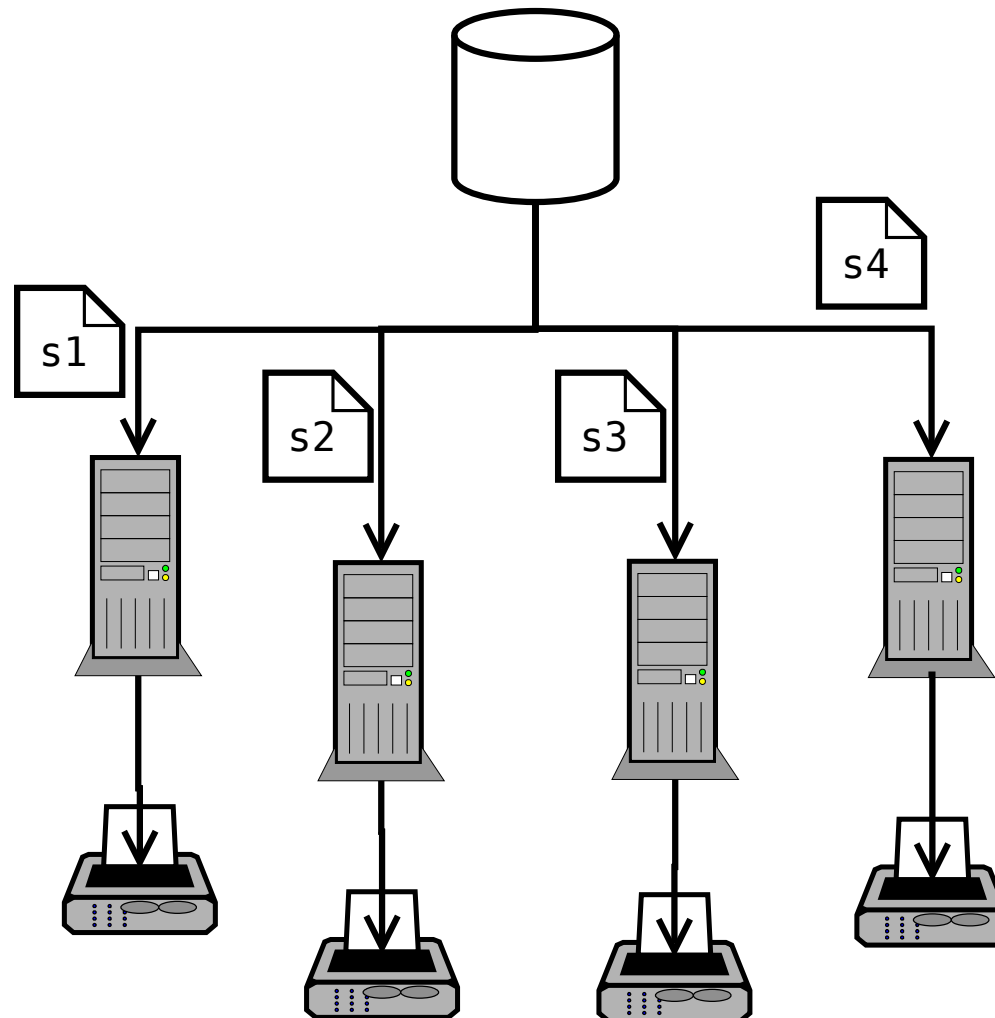


Terminata la stampa di s1 il processo p1 conclude la sua esecuzione, gli altri proseguono e nuovi processi entreranno in gioco per s4, s5, ..., sn.

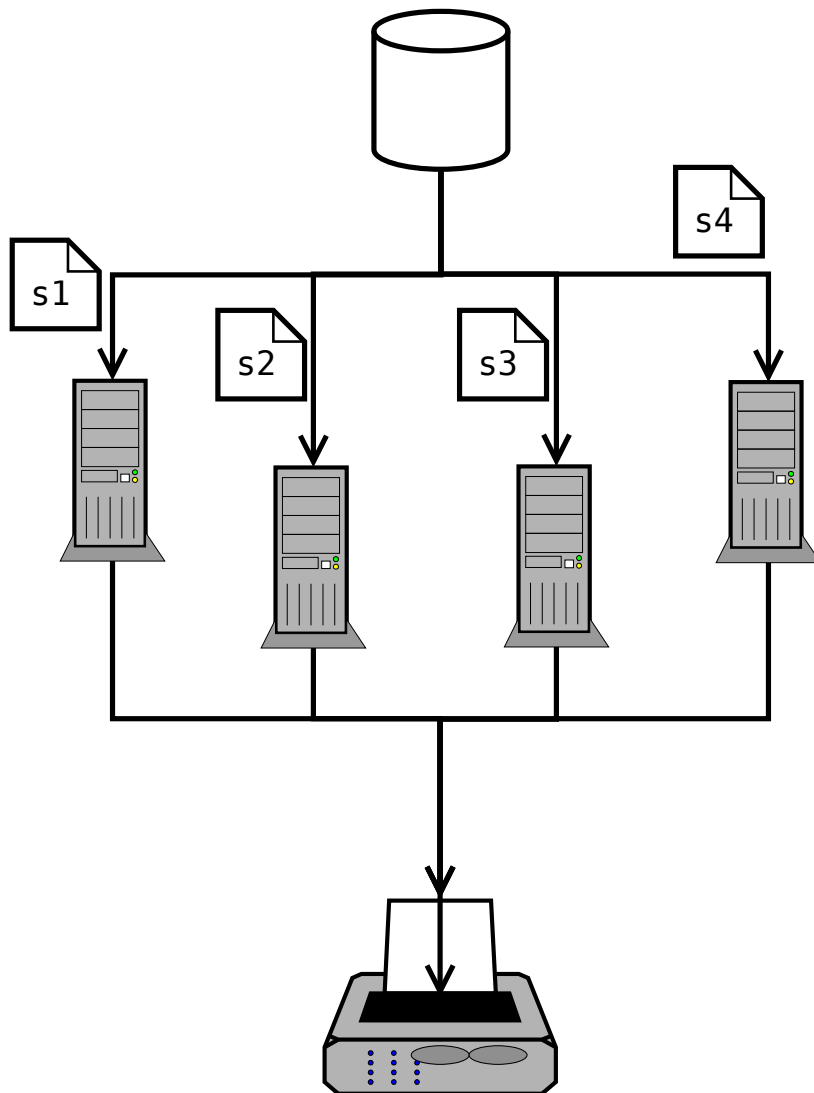
Disco	CPU	Stampante
p4	p3	p2



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

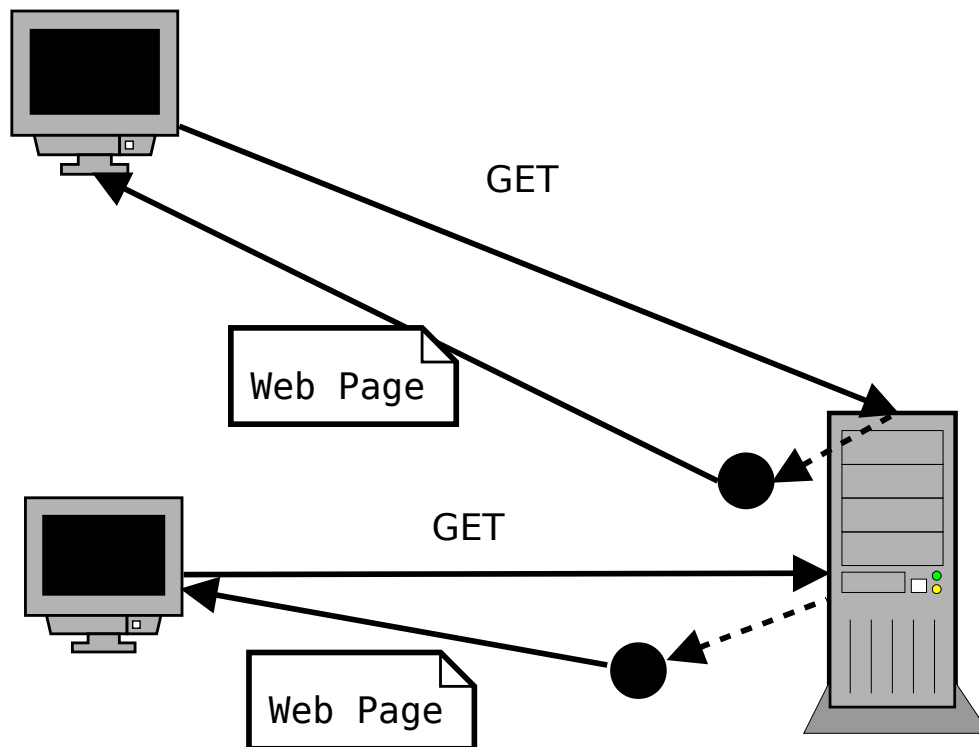


Sia in caso di multitasking effettivo che simulato, l'accesso a risorse condivise può essere problematico. Ad esempio, se si usa un unico terminale di stampa l'output dei vari processi potrebbe accavallarsi.

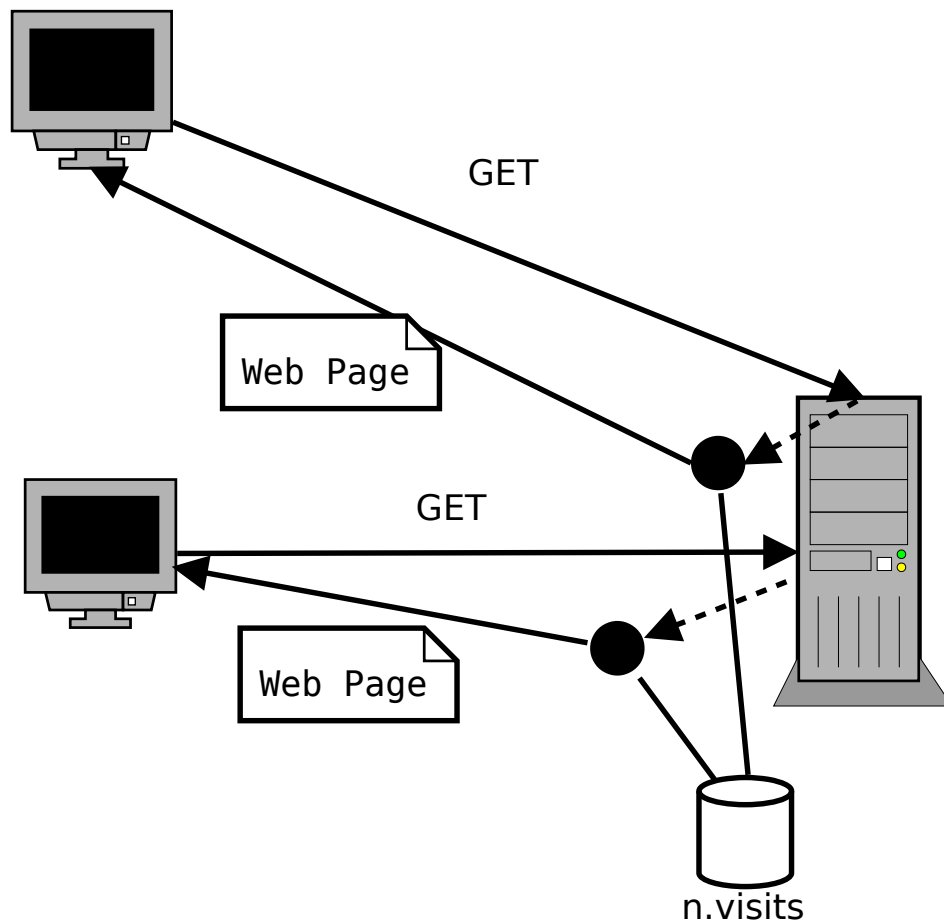


```
s1 year of birth 1978  
s1 age 38  
s2 year of birth 1979  
s2 age 37  
s3 year of birth 1980  
s4 year of birth 1981  
s3 age 36  
s4 age 35
```

Un server web serve ogni richiesta su un thread dedicato.



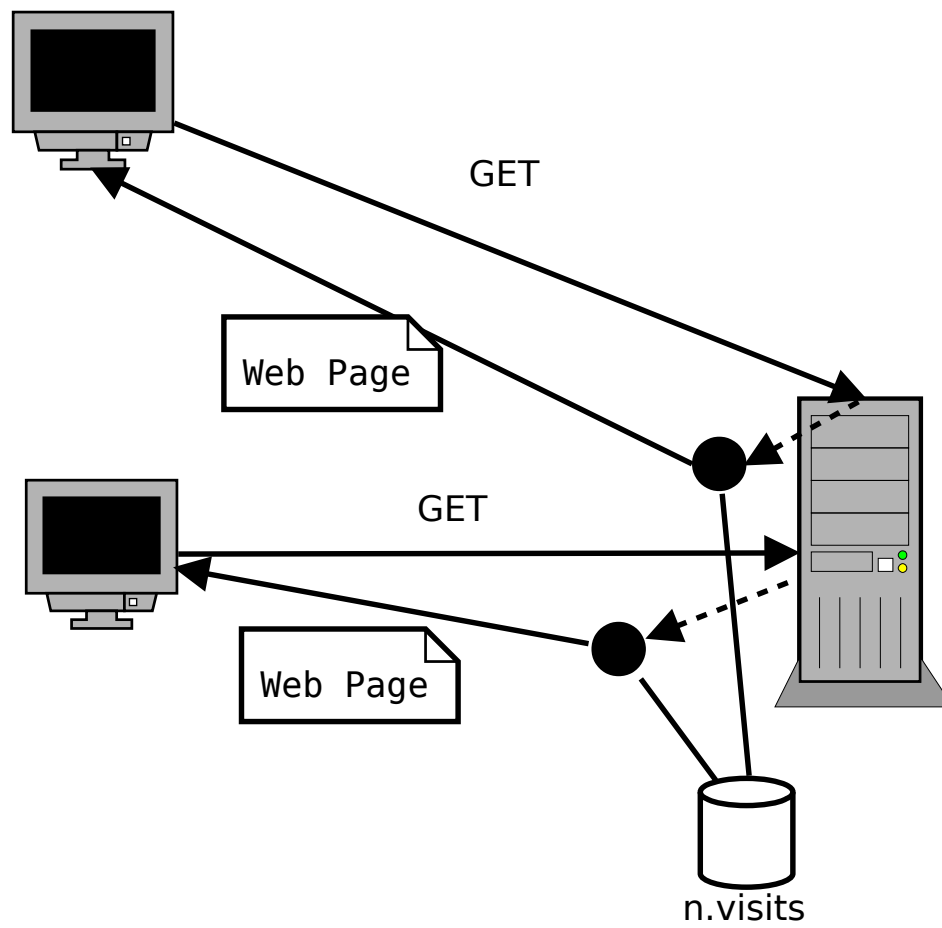
Un server web serve ogni richiesta su un thread dedicato. Supponiamo che per ogni richiesta si voglia aumentare il valore di un contatore salvato su uno storage.



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
```

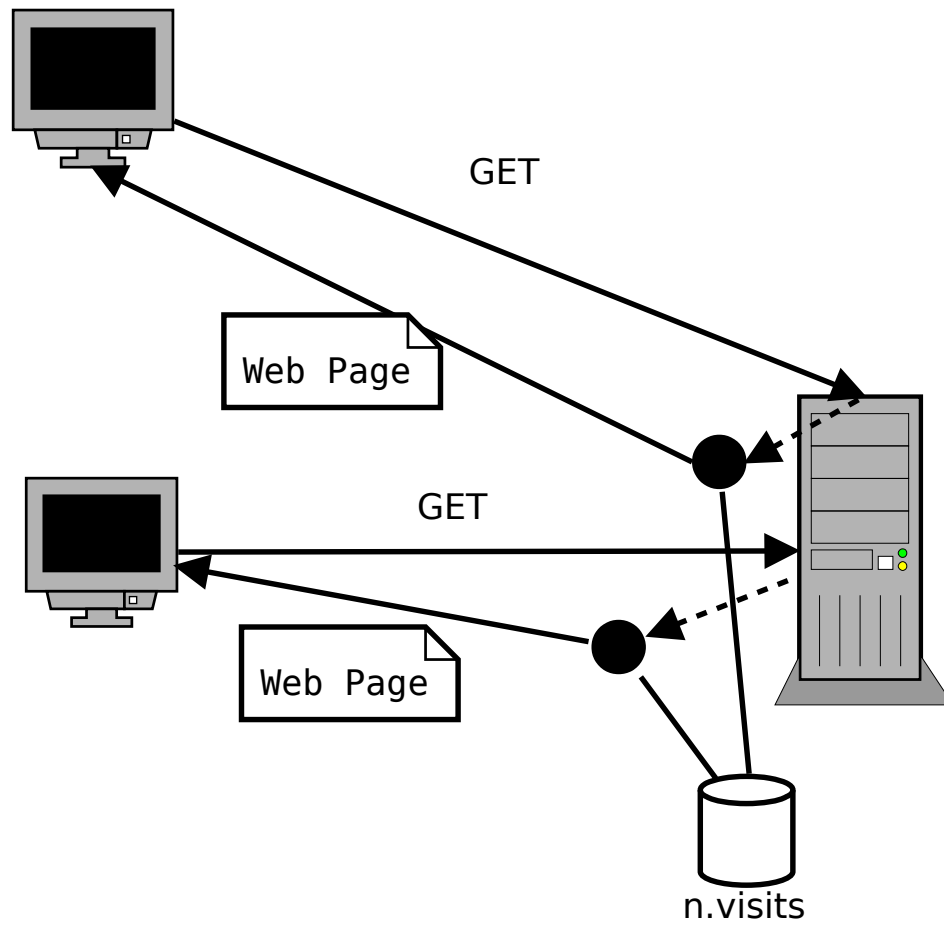
```
leggi c
incrementa c
scrivi c
```

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`

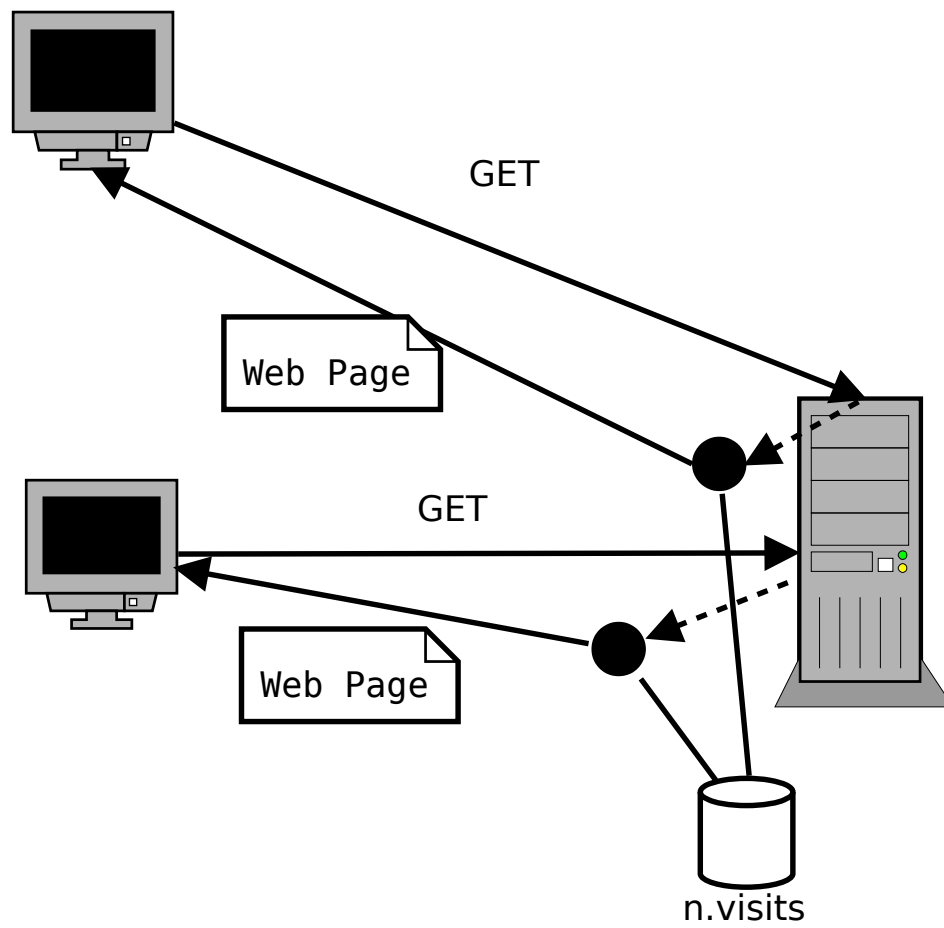
leggi c
incrementa c
scrivi c

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.



leggi c
incrementa c
scrivi c

c=n

leggi c
incrementa c
scrivi c

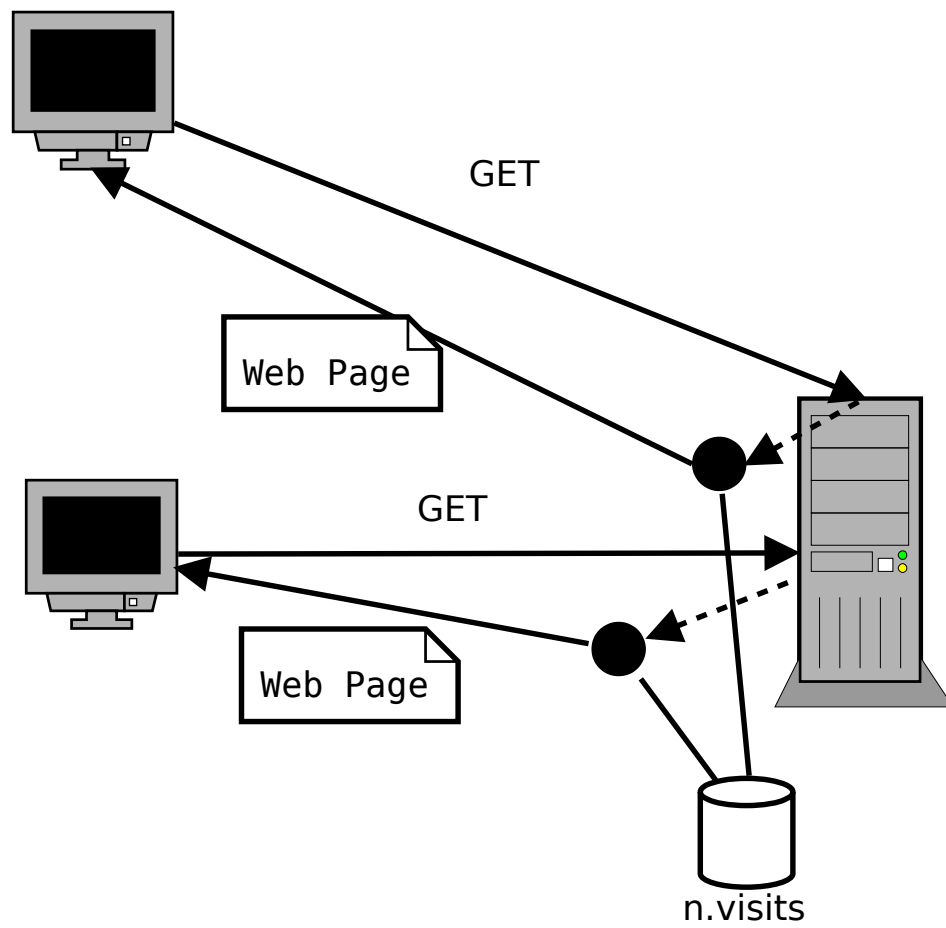
c=n

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

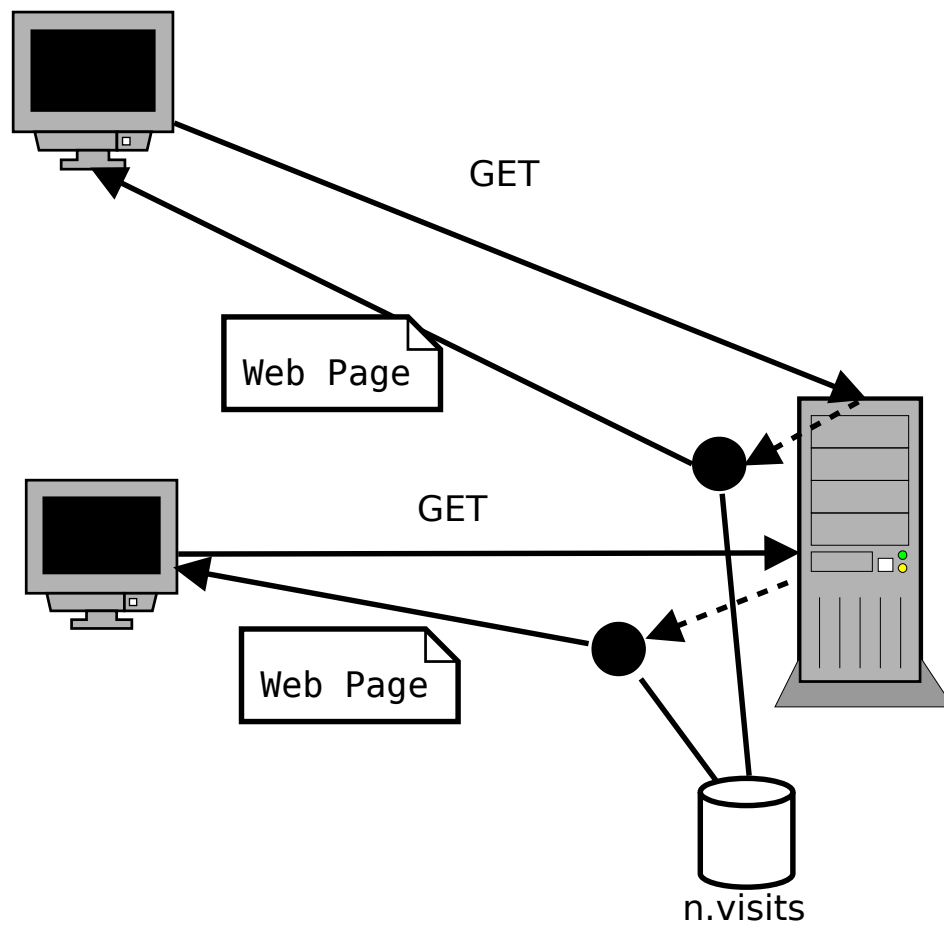
 $c=n$

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

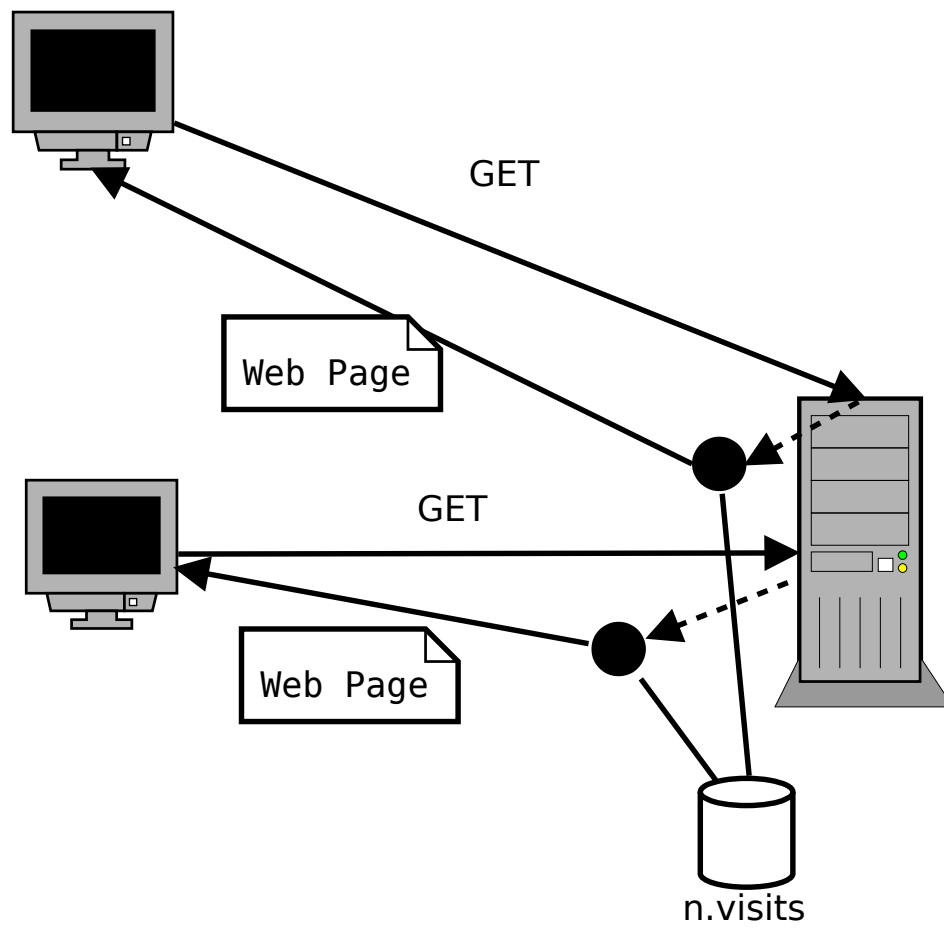
 $c=n+1$

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.



leggi c
incrementa c
scrivi c

 $c = n + 1$

leggi c
incrementa c
scrivi c

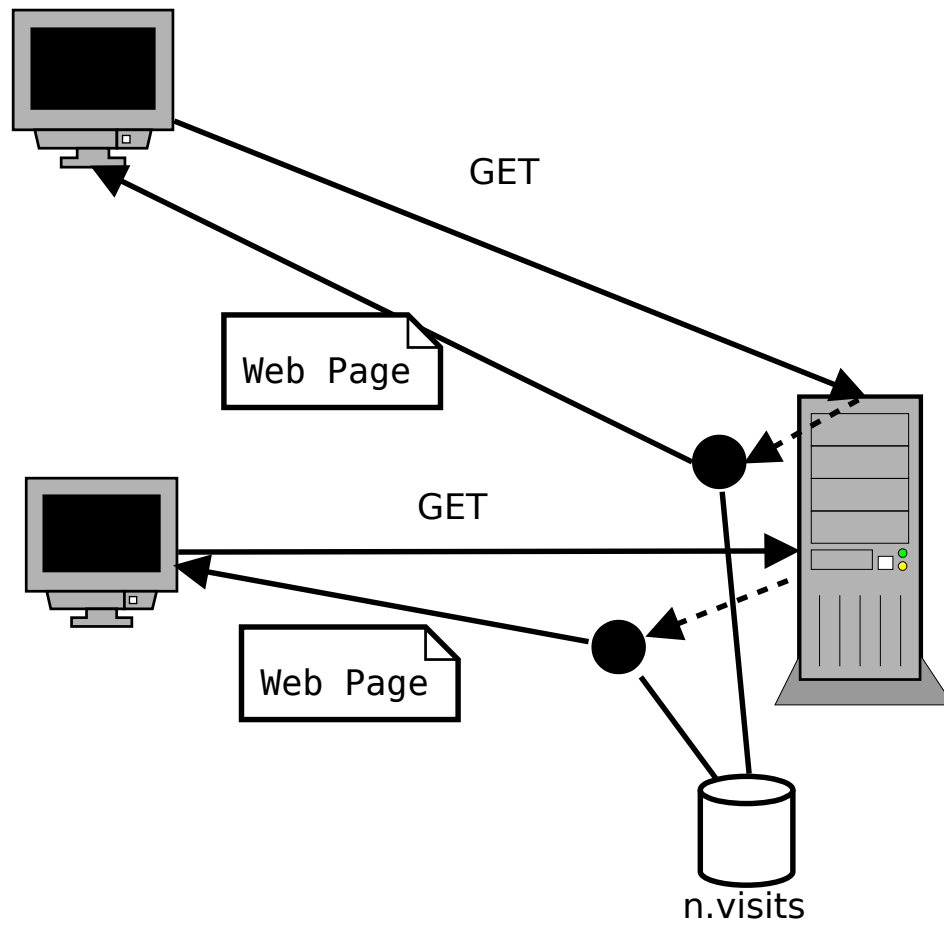
 $c = n + 1$

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.



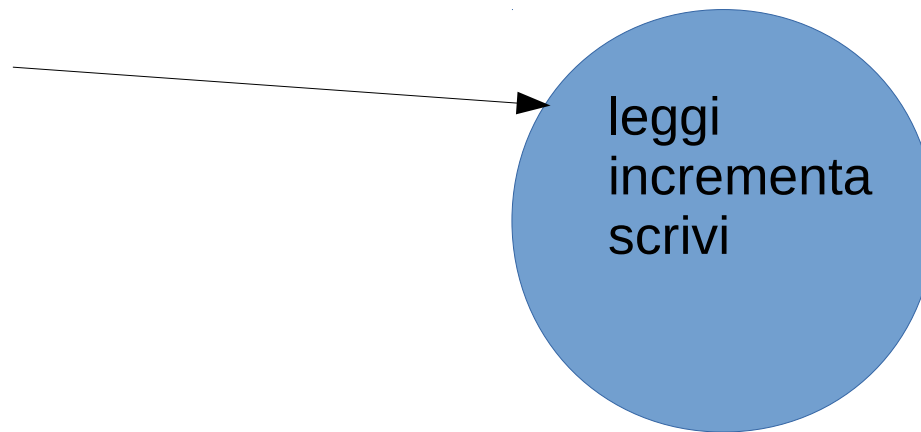
Contatore su file = n

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.



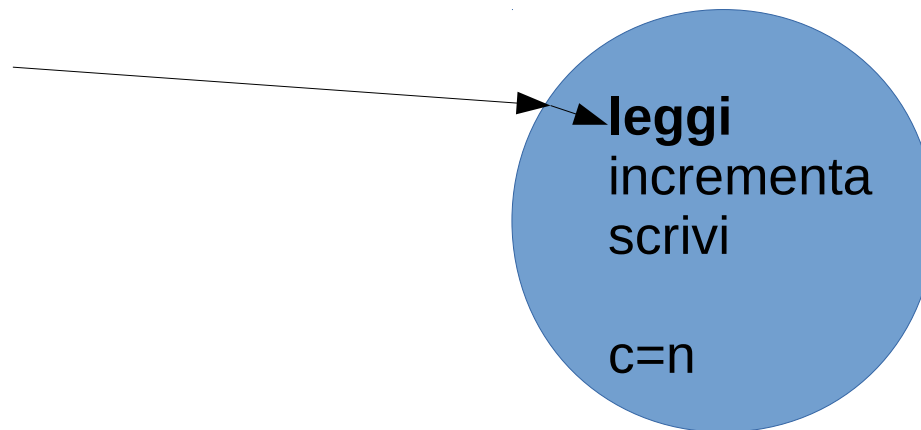
Contatore su file = n

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. Inizia quindi ad eseguire.



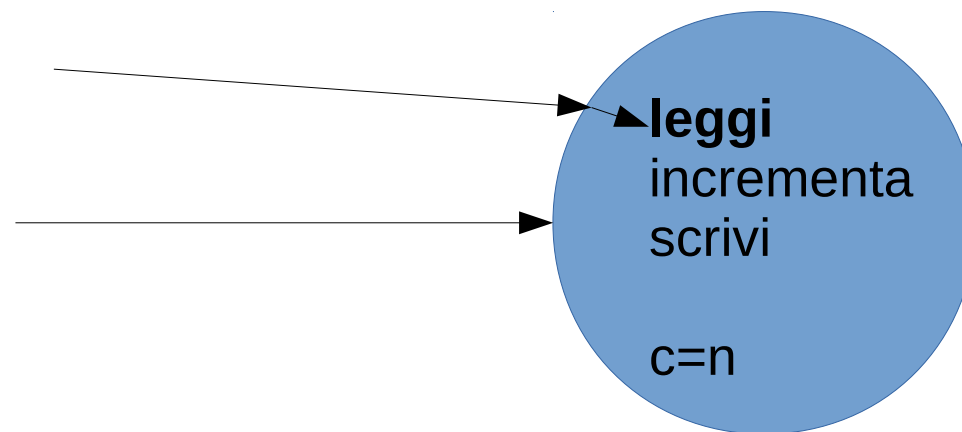
Contatore su file = n

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



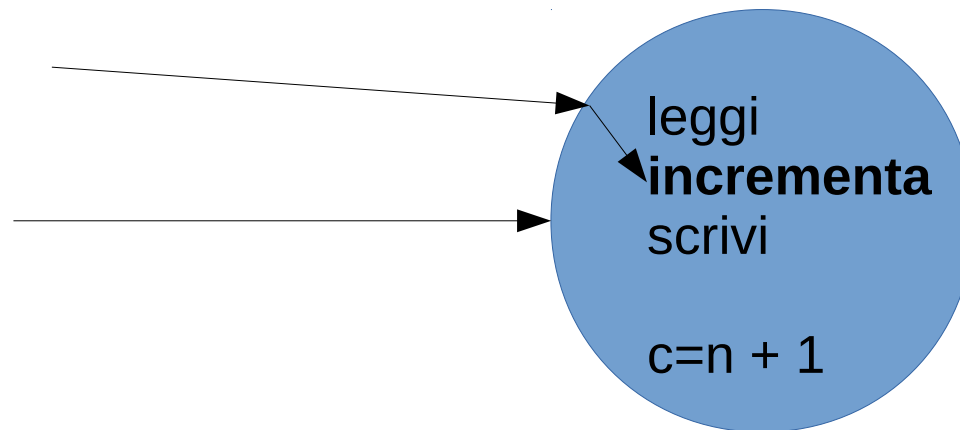
Contatore su file = n

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.

L'esecuzione della richiesta del primo thread continua.

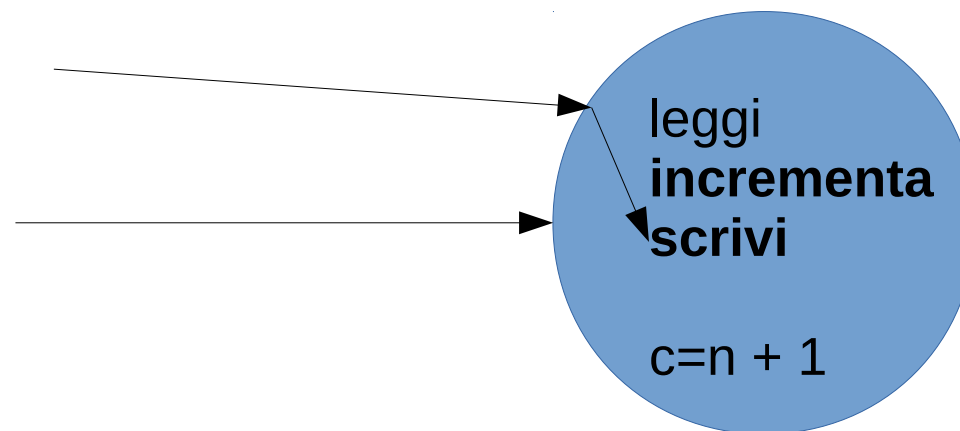


Contatore su file = n

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.

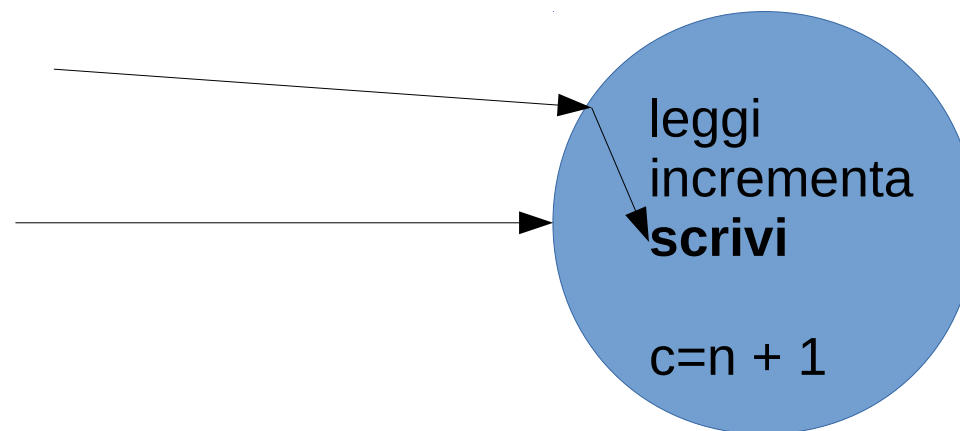


Contatore su file = $n + 1$

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.



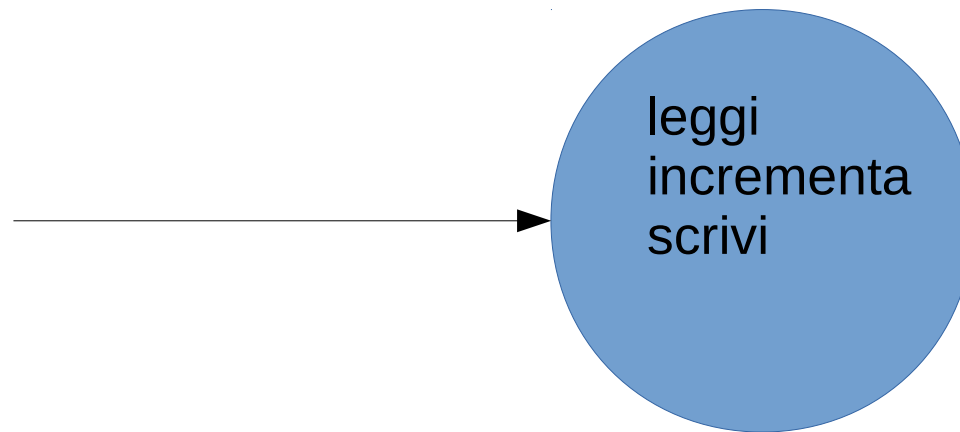
Contatore su file = $n + 1$

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.



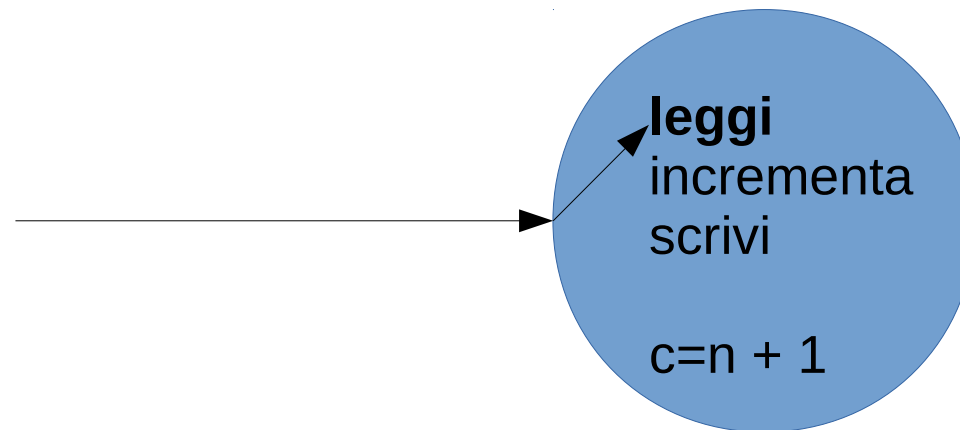
Contatore su file = $n + 1$

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.

La richiesta del secondo thread verrà portata a compimento.



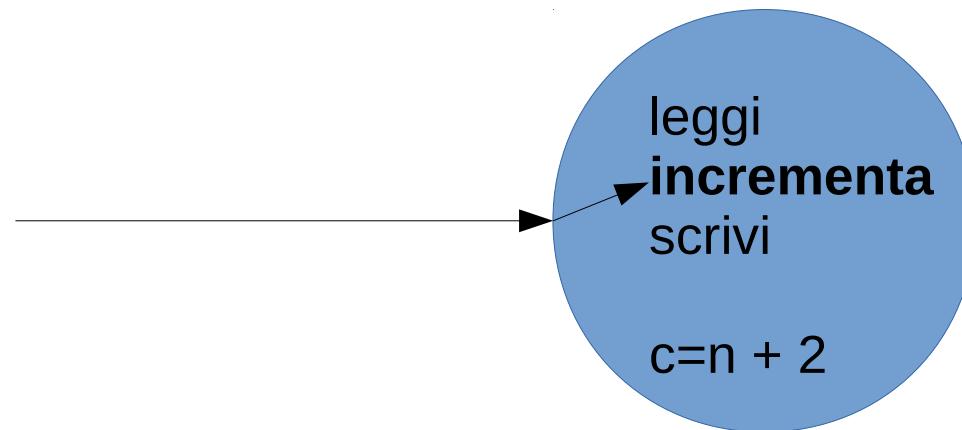
Contatore su file = $n + 1$

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.

La richiesta del secondo thread verrà portata a compimento.



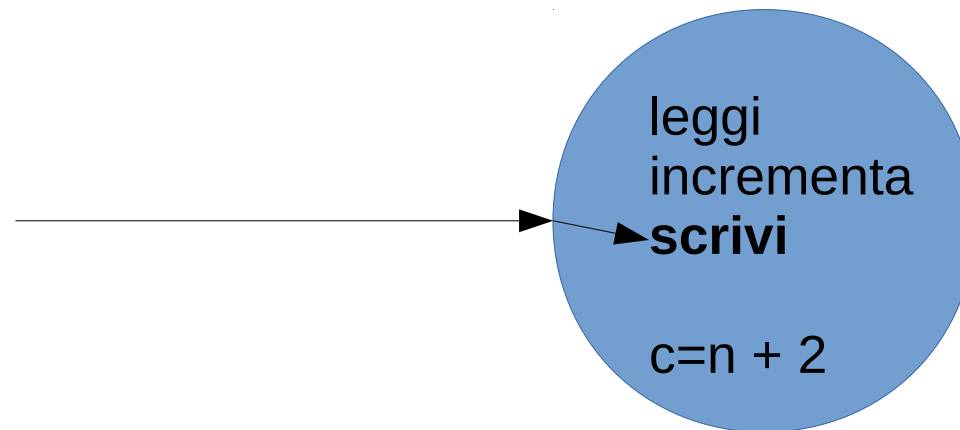
Contatore su file = $n + 1$

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.

La richiesta del secondo thread verrà portata a compimento.



Contatore su file = $n + 2$

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  
    }  
}
```

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

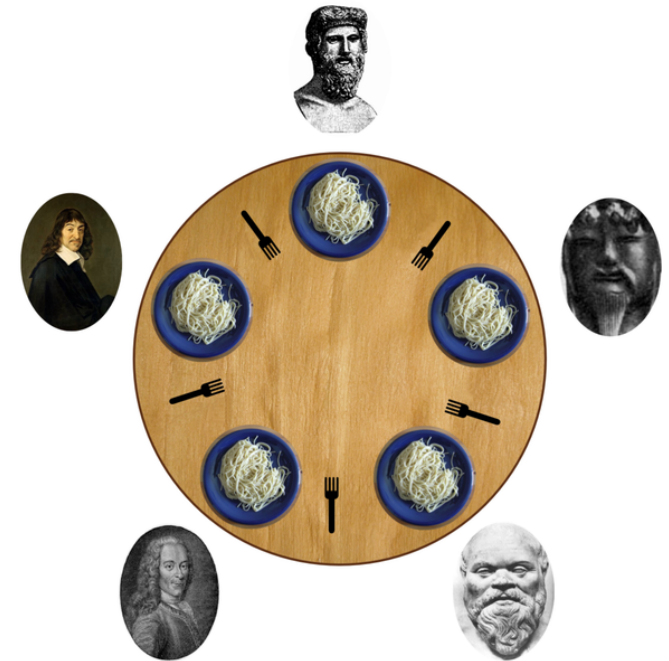
Un deadlock si verifica quando due o più processi sono bloccati per sempre in attesa l'uno dell'altro.

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

