**INTRODUZIONE A JAVA** *24-09-24*

**Java** è un linguaggio di programmazione.

Per sviluppare ed eseguire applicazioni in Java useremo la piattaforma Java JDK, contenente,   
tra le varie cose:

* **javac**: un compilatore;
* **Java Virtual Machine (JVM**): una macchina virtuale.

Installata la piattaforma, il codice Java va scritto in un file .java. Questo deve essere poi compilato attraverso javac, che darà in uscita un file oggetto .class contenente il cosiddetto bytecode,  
e cioè del codice binario che può essere eseguito solo dalla JVM. Sarà questa, dunque, ad eseguire effettivamente l’applicazione, garantendo così l’indipendenza delle applicazioni Java dal tipo   
di architettura su cui dovranno essere eseguite. La JVM, in particolare, *interpreta* il bytecode, e cioè traduce in codice nativo ogni sua istruzione, e passa alla successiva solo dopo che la CPU ha eseguito l’istruzione appena tradotta. Nelle versioni più recenti di Java JDK, inoltre, la JVM monitora l’esecuzione del programma, individuando le porzioni di bytecode che vengono interpretate più frequentemente.   
Una volta individuate, la JVM le traduce in codice nativo, e così la prossima volta che queste porzioni di bytecode si ripresentano non verranno nuovamente interpretate, ma verrà direttamente mandata   
in esecuzione la loro traduzione in codice nativo, portando così ad un miglioramento dei tempi   
di esecuzione.

Un codice Java è una sequenza di statement. Ogni statement termina per ‘;’.

Vediamo un esempio di codice Java:

Immagine che contiene testo, Carattere, linea, schermata

Descrizione generata automaticamente

Il codice Java va sempre scritto in classi. In particolare, la regola che useremo noi è quella di definire un’unica classe per file sorgente, e il file sorgente deve avere lo stesso nome della classe.   
La sintassi per definire una classe, i suoi campi e i suoi metodi (funzioni), costruttori compresi, così come la sintassi per accedere ad un campo di un oggetto o invocare un metodo, è la stessa del C++. Nel nostro caso abbiamo la classe Main (il nome può essere scelto liberamente dal programmatore). Per convenzione, i nomi delle classi vanno scritti in upper camel case.

Il punto d’ingresso di esecuzione del programma è rappresentato dal metodo main. Questo metodo va chiamato obbligatoriamente in questo modo, e deve avere le seguenti caratteristiche:

* E’ public (vedremo che significa);
* E’ static (vedremo che significa);
* Ha come tipo di ritorno void, e cioè non deve restituire alcun valore;
* Deve accettare un unico argomento di tipo String[].

Nel nostro caso, all’interno del metodo main viene invocato il metodo   
System.out.println(“Hello world!”):

* System è un classe predefinita del linguaggio. Rappresenta l’ambiente in cui il programma   
  viene eseguito;
* out è un campo static della classe System;
* println(string *stringa*) è un metodo dell’oggetto out. Stampa a video la String *stringa*.

Questo metodo, dunque, stampa a video la String “Hello World”.

Java è un linguaggio fortemente tipizzato: bisogna dichiarare il tipo di una variabile prima di memorizzare un valore in essa. I tipi si dividono in due categorie:

* Tipi primitivi:
  + boolean: booleano (true o false).
  + byte: intero rappresentato su 8 bit in complemento a 2;
  + short: intero rappresentato su 16 bit in complemento a 2;
  + int: intero rappresentato su 32 bit in complemento a 2;
  + long: intero rappresentato su 64 bit in complemento a 2;
  + float: reale rappresentato in precisione singola;
  + double: reale rappresentato in precisione doppia;
  + char: carattere rappresentato su 16 bit in codifica Unicode.
* Tipi riferimento:
  + Array;
  + Classe;
  + Interfaccia.

Un letterale carattere va scritto tra singoli apici. Si può scrivere direttamente il carattere,  
o il suo codice esadecimale preceduto da “\u”:

*Es.*

‘a’

‘\u61’

Un letterale stringa va scritto tra doppi apici:

*Es.*

“ciao mondo”

I commenti si scrivono in questo modo:

// commento su singola linea

/\*

commento su

più righe

\*/

Una variabile si dichiara in questo modo:

*tipo nome\_variabile*

Per inizializzare una variabile con un certo valore o per assegnarle un valore successivamente   
si usa l’operatore =. E’ possibile dichiarare e inizializzare più variabili dello stesso tipo   
nello stesso statement separandole da virgola:

*Es.*

int x;

float y = 1.5;

int i1 = 2, i2 = 3;

Una variabile costante si dichiara in questo modo:

final *tipo nome\_costante*

E’ possibile dichiarare una costante senza inizializzarla, assegnandole un valore successivamente.   
Una volta dato un valore ad una costante, non è più possibile cambiarlo.

*Es.*

Dichiaro la variabile costante *z* di tipo int e la inizializzo al valore 2.   
Essendo una costante, una volta datole un valore non è più possibile cambiarlo.

Dichiaro la variabile costante *zz* di tipo int, senza inizializzarla,   
e le assegno successivamente il valore 3.

final int z = 2;

//z = 5;

final int zz;

zz = 3;

Dividendo un intero per 0 viene generata l’eccezione ArithmeticException.

Dividendo un reale per 0:

* Se il reale è positivo, viene generata l’eccezione Double.POSITIVE\_INFINITY;
* Se il reale è negativo, viene generata l’eccezione Double.NEGATIVE\_INFINITY;
* Se il reale è 0, viene generata l’eccezione Double.NaN.

Il seguente schema mostra le conversioni implicite possibili:

* Le frecce in linea continua indicano le conversioni implicite che non portano   
  a perdita di precisione;
* Le frecce in linea tratteggiata indicano le conversioni implicite che possono portare   
  a perdita di precisione.

Immagine che contiene diagramma, linea, Carattere

Descrizione generata automaticamente

**NB:** i boolean non sono convertibili implicitamente in altri tipi, e viceversa. In una condizione, dunque, bisogna mettere proprio un boolean, per esempio restituito da un operatore di confronto   
(a differenza di ciò che succede in altri linguaggi, in cui un qualsiasi valore diverso da 0   
viene implicitamente convertito in true).

Le conversioni non mostrate nello schema si possono fare esclusivamente in modo esplicito,   
in questo modo:

(*tipo\_a\_cui\_convertire*) *valore*

*Es.*

int a = 1;

*b* è una variabile di tipo long. Per assegnarle il valore di *a*, un int,   
bisogna prima convertire questo valorein un long. Questa conversione avviene   
in modo implicito.

*c* è una variabile di tipo int. Per assegnarle il valore di *b*, un long,   
bisogna prima convertire questo valore in un int. Questa conversione non avviene   
in modo implicito, per cui va fatta in modo esplicito.

long b;

b = a;

int c;

// c = b;

c = (int)b;

Java presenta if..else, switch, while, do..while e for. La sintassi è la stessa del C++.

E’ presente anche il cosiddetto *for each*, la cui sintassi è la seguente:

for (*tipo\_variabile* *variabile* : *set\_expression*) {

*corpo*

}

Ad ogni iterazione, in *variabile* finirà il valore dell’elemento successivo di *set\_expression*.   
*set\_expression* può essere:

* Un array;
* Un oggetto che implementa l’interfaccia java.lang.Iterable.

*tipo*\_*variabile* deve corrispondere al tipo degli elementi di *set\_expression*.

**INGRESSO E USCITA CON LA CLASSE SCANNER** *26-09-24*

Ad ogni programma Java in esecuzione sono associati 3 flussi:

* System.in: flusso per l’ingresso standard;
* System.out: flusso per l’uscita standard;
* System.err: flusso per l’uscita di errore.

Di default, System.in è collegato alla tastiera, mentre System.out e System.err sono collegati al video (dunque un programma Java legge l’input da tastiera, e stampa l’uscita standard e i messaggi di errore a video).

E’ possibile ridirezionare questi flussi, usando i comandi specifici del sistema operativo su cui   
il programma è in esecuzione.

Per l’uscita standard si possono usare i seguenti metodi:

* System.out.println(*valore*): stampa *valore* e va a capo;
* System.out.print(*valore*): stampa *valore*, senza andare a capo;

Per l’ingresso standard si sfrutta la classe Scanner. Per poter usare questa classe bisogna scrivere questo all’inizio del file (vedremo che significa):

import java.util.Scanner;

Fatto ciò, bisogna creare un oggetto di tipo Scanner associato ad una certa *fonte\_di\_caratteri*,   
in questo modo (vedremo che significa):

Scanner s = new Scanner(*fonte\_di\_caratteri*);

Nel caso di ingresso standard, *fonte\_di\_caratteri* è System.in.

A questo punto si possono invocare sull’oggetto creato i seguenti metodi offerti dalla classe Scanner:

* nextBoolean(): estrae il token successivo da *fonte\_di\_caratteri* e restituisce il valore   
  di tipo boolean corrispondente;
* nextFloat(): estrae il token successivo da *fonte\_di\_caratteri* e restituisce il valore   
  di tipo float corrispondente;
* nextDouble(): estrae il token successivo da *fonte\_di\_caratteri* e restituisce il valore   
  di tipo double corrispondente;
* nextInt(): estrae il token successivo da *fonte\_di\_caratteri* e restituisce il valore   
  di tipo int corrispondente;
* nextLine(): estrae la riga successiva da *fonte\_di\_caratteri* e la restituisce come valore   
  di tipo String;
* next(): estrae il token successivo da *fonte\_di\_caratteri* e lo restituisce come valore   
  di tipo String;
* hasNext(): restituisce true se *fonte\_di\_caratteri* presenta altri token da leggere, false altrimenti.

Tutti questi metodi eccetto l’ultimo sono bloccanti: una volta invocati, il programma si blocca fintantoché *fonte\_di\_caratteri* non presenta effettivamente dei caratteri.

*Es.*

Dichiaro una variabile float *m,* inizialmente a 0,   
che conterrà il massimo dei float inseriti dall’utente, e una variabile boolean *primo*, inizialmente a *true*, che mi dirà se l’utente deve ancora inserire   
un primo float.

Creo successivamente un oggetto Scanner *sc* associato a System.in.

Quando si inserisce l’input da tastiera, ctrl+d   
(o ctrl+z, a seconda del sistema operativo usato) serve ad indicare che si è finito di inserire caratteri (non esiste propriamente un carattere EOF). Fintantoché l’utente non preme questa combinazione di tasti e inserisce caratteri   
da tastiera, dunque, leggo il float corrispondente alla sequenza di caratteri inserita, e se è maggiore   
del float massimo inserito finora, aggiorno   
la variabile *m* con il nuovo valore massimo.

Una volta che l’utente ha finito di inserire caratteri, se l’utente ha inserito almeno un float, stampo il massimo che ha inserito.

*Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente*

**CLASSI IN DETTAGLIO**

Quando dichiariamo una variabile di un certo tipo classe, come in questo caso:

*NomeClasse* var;

Quello che facciamo è creare un *riferimento* ad un oggetto di tale tipo classe(una sorta di puntatore, che conterrà l’indirizzo, detto a sua volta *riferimento*, di un oggetto di tale tipo classe).  
L’operatore new permette effettivamente di creare nello heap un oggetto di tipo classe,   
e restituisce un riferimento all’oggetto creato. Se la classe di cui si vuole creare un oggetto   
non ha alcun costruttore (vedremo in futuro), la sintassi è la seguente:

new *NomeClasse*()

*Es.*

class Punto {

double x;

Nella definizione di un metodo non static, quando si scrive il nome di un membro non static della classe,   
è sottinteso che si accede al membro dell’oggetto di cui verrà invocato tale metodo (si vedrà meglio dopo).

void trasla(double t){

x += t;

}

}

Creo un riferimento *p1* ad un oggetto di tipo Punto.

Creo un riferimento *p2* ad un oggetto di tipo Punto, e lo inizializzo   
con il riferimento di un nuovo oggetto di tipo Punto.   
*p2*, dunque, punterà a questo nuovo oggetto.

Assegno a *p1* il valore di *p2*, e cioè il riferimento dell’oggetto appena creato. *p1* e *p2*, dunque, punteranno allo stesso oggetto.

Punto p1;

Punto p2 = new Punto();

p1 = p2;

Una classe può avere tra i suoi campi anche riferimenti ad oggetti di altre classi:

*Es.*

class A {

…

Definisco una classe *A* che ha, tra i suoi campi,   
un riferimento *z* ad un oggetto di tipo *AltraClasse*.

AltraClasse z;

…

}

**NB:** in futuro, quando non c’è rischio di confusione, per semplicità non faremo distinzione tra il riferimento e l’oggetto riferito, tenendo conto però che si avrà sempre a che fare con riferimenti,   
visto che si lavorerà direttamente sempre e solo con essi, e mai con gli oggetti effettivi.  
Questo, tra le varie cose, significa che:

* Useremo il nome del riferimento per indicare anche l’oggetto riferito (*es*. se abbiamo   
  “Punto p1 = new Punto()”, diremo per semplicità che “creiamo un oggetto *p1* di tipo Punto”).
* Se un metodo accetta come argomento un riferimento ad un oggetto   
  o restituisce un riferimento ad un oggetto, diremo semplicemente che accetta come argomento un oggetto e restituisce un oggetto;
* Se una classe ha come campo un riferimento a un oggetto, diremo semplicemente che ha come campo un oggetto
* …

Se non inizializzati esplicitamente, i campi assumono un valore di default, che è:

* false per i campi di tipo boolean;
* 0 per i campi di tipo byte, short, int, long, float e double;
* ‘\u0000’ per i campi di tipo char;
* null per i campi di tipo riferimento.

Quando si inizializza esplicitamente un campo, non si possono invocare metodi che lanciano   
eccezioni di tipo “checked”.

*Es.*

class A {

Dichiaro una classe *A* che ha, tra i suoi campi:

* *x,* di tipo int. Non avendolo inizializzato esplicitamente, assumerà come valore 0;
* *a*, di tipo double, inizializzato a 3.14;
* *b,* di tipo double, inizializzato a + 1.1 + Math.sqrt(3.5).   
  Il metodo Math.sqrt() non lancia eccezioni di tipo checked, per cui può essere invocato nell’inizializzazione del campo senza problemi.

int x;

double a = 3.14;

double b = a + 1.1 + Math.sqrt(3.5);

…

}

**COSTRUTTORI**

Un costruttore è un particolare metodo di una classe che viene invocato tutte le volte che viene creato un oggetto della classe. Serve ad assegnare ad ogni campo dell’oggetto un preciso valore iniziale.   
Non ha avere alcun tipo di ritorno (nemmeno void), ha lo stesso nome della classe   
e può avere 0 o più argomenti. Può essere invocato solo attraverso l’operatore new, così:

new *NomeClasse*(*argomenti\_costruttore*)

In questo modo verrà creato un oggetto di tipo *NomeClasse*, e verrà invocato su di esso il costruttore, passandogli come argomenti *argomenti\_costruttore*.

Una classe può avere più costruttori, che devono differire per il numero e/o il tipo degli argomenti.   
A seconda di ciò che si passa come *argomenti\_costruttore* verrà invocato il costruttore opportuno.   
Se una classe non ha alcun costruttore, viene usato un costruttore di default, privo di argomenti,   
che non fa nulla. Una volta definito almeno un costruttore, il costruttore di default non viene più usato (per cui, se si vorrà usare anche un costruttore senza argomenti, andrà definito esplicitamente).

*Es1*

class Pippo {

int a;

Creo un oggetto *p1* di tipo *Pippo*. Poiché la classe *Pippo* non ha alcun costruttore, viene invocato il costruttore di default,   
privo di argomenti, che non fa nulla.

}

Pippo p1 = new Pippo();

*Es2*

class A {

double a, b;

A(double c, double d){

a = c;

b = d;

}

A(double e){

a = e;

}

}

Creo un oggetto *a1* di tipo *A*. In base agli argomenti passati,   
verrà invocato il primo costruttore.

Creo un oggetto a2 di tipo A. In base agli argomenti passati,   
verrà invocato il secondo costruttore.

A a1 = new A(1.1, 2.2);

A a2 = new A(3.3);

**METODI**

In Java il passaggio degli argomenti ad un metodo è solo per valore, e bisogna passare ogni argomento.

Una classe può avere più metodi con lo stesso nome, purché differiscano per il numero e/o il tipo   
degli argomenti.

Ogni volta che viene invocato un metodo, sullo stack viene creato un record di attivazione,   
e cioè una struttura dati contenente lo spazio per le variabili locali del metodo   
e gli argomenti del metodo, quest’ultimi inizializzati con i valori dei corrispondenti   
argomenti attuali. Al termine dell’esecuzione di un metodo, il suo record di attivazione viene deallocato.

*Es.*

Class Uno {

int valore;

Uno(int v){

valore = v;

}

void setValore(int v){

valore = v;

}

Class Due{

void m1(Uno u, int val){

u.setValore(val);

}

Creo un riferimento *u1* ad un oggetto di tipo *Uno*, e lo inizializzo   
con il riferimento di un nuovo oggetto di tipo *Uno* avente *valore* a 10.  
Creo un riferimento *u2* ad un oggetto *d1* di tipo *Due*, e lo inizializzo   
con il riferimento di un nuovo oggetto di tipo *Due*.

Invoco sull’oggetto riferito da *d1* il metodo *m1*, passando come argomenti *u1* e *10*. Sullo stack verrà dunque allocato il record di attivazione di *m1*,   
contenente gli argomenti formali *u* e *val*, inizializzati rispettivamente   
con il valore di *u1*, e quindi con il valore del riferimento dell’oggetto riferito   
da *u1,* e 10.

*m1* invoca sull’oggetto riferito da *u* il metodo setValore,   
passando come argomento *val*. Sullo stack verrà dunque allocato il record di attivazione di *setValore*, contenente l’argomento formale *v*, inizializzato con il valore di *val*.

Terminata l’esecuzione di *setValore*, viene deallocato il suo record di attivazione, e il controllo torna a *m1*. Terminata anche l’esecuzione di *m1*, viene deallocato anche il suo record di attivazione.

Uno u1 = new Uno(10);

Due d1 = new Due();

d1.m1(u1, 10);

**VARIABILI LOCALI**

Le variabili locali sono le variabili che vengono dichiarate all’interno di un metodo.   
Possono essere dichiarate in qualsiasi punto del metodo, non hanno un valore di default nel caso in cui non venissero inizializzate esplicitamente e vengono deallocate alla fine dell’esecuzione del metodo.

Il compilatore controlla che ogni variabile locale abbia un valore ben definito prima che si legga da esse. In alcuni casi, tuttavia, se non si inizializza una variabile locale e le si assegna un valore   
solo successivamente, anche se questa presenta un valore ben definito quando si va a leggere da essa,   
il compilatore comunque si lamenta. In questi casi, conviene inizializzare la variabile   
con un valore arbitrario.

**THIS**

All’interno di un metodo non static, la parola chiave this è un riferimento all’oggetto   
su cui viene invocato il metodo, detto *oggetto implicito*.

Ogni volta che viene creato un oggetto classe, insieme agli altri campi viene allocato sullo heap anche il campo this, contenente un riferimento all’oggetto stesso. All’interno dei metodi non static,   
ogni volta che si scrive il nome di un membro non static della classe, è sottinteso che si accede a this.*nome\_membro*, e quindi al membro dell’oggetto di cui verrà invocato tale metodo.   
Non è consigliato dunque usare la parola chiave this quando non necessario, come in questo caso:

class Esempio{

int a;

void incrementa(){

this.a++;

}

void m(){

this.incrementa();

}

}

in quanto rende il codice inutilmente prolisso, e quindi meno leggibile.

Sono 2 i casi in cui è necessario usare this:

1. Quando ci sono delle omonimie;
2. Quando bisogna restituire un riferimento all’oggetto implicito.

*Es.*

class Numero{

int n;

Il costruttore assegna al campo *n* dell’oggetto implicito   
il valore dell’argomento *n*. A causa di questa omonimia,   
in questo costruttore, l’identificatore *n* dell’argomento   
nasconde l’identificatore *n* dell’oggetto implicito.   
Per riferirmi allora al campo *n* dell’oggetto implicito posso usare this.*n*.

Numero(int n){

this.n = n;

}

Numero maggiore(Numero altro){

if(n > altro.n)

Se il campo *n* dell’oggetto implicito è maggiore del campo *n* dell’oggetto riferito dal riferimento passato come argomento, restituisco un riferimento all’oggetto implicito.

return this;

else

return altro;

}

}

La parola chiave this può essere utile anche per evitare codice duplicato nei costruttori.   
In un costruttore (e solo in esso), come primo statement, è possibile invocare un altro costruttore sull’oggetto implicito, in questo modo:

this(*argomenti\_costruttore*)

L’idea allora è quella di creare un primo costruttore più generale, e poi altri costruttori   
più specifici che, come prima cosa, invocano il primo costruttore.

*Es.*

class Studente{

String nome;

String cognome;

int matricola;

double media;

Studente(String nome, String cognome, int matricola, double media){

this.nome = nome;

Definisco un costruttore più generale, con un argomento per ogni campo,   
che assegna un valore ad ogni campo dell’oggetto implicito, e un secondo costruttore, senza il campo relativo   
alla media, che invoca il primo costruttore passando, come campo relativo alla media, un valore predefinito stabilito dal programmatore.

this.cognome = cognome;

this.matricola = matricola;

this.media = media;

}

Studente(String nome, String cognome, int matricola){

this(nome, cognome, matricola, 0.0);

}

**CAMPI E METODI STATIC**

E’ possibile definire un campo o un metodo static, usando il modificatore static.

Un campo static è un campo della classe, e non dei suoi singoli oggetti.

Esiste in copia singola, indipendentemente dal numero di oggetti della classe.   
Viene allocato in memoria statica la prima volta che, durante l’esecuzione del programma,   
viene incontrata la classe contenente tale campo static, e viene deallocato alla fine dell’esecuzione   
del programma.

Per accedere ad un campo static di una classe si può usare la seguente sintassi:

*NomeClasse*.*nomeCampoStatic*

All’interno di una classe che presenta il campo static si può usare semplicemente *nomeCampoStatic*.

Quando si inizializza un campo static si possono usare solo:

* Letterali;
* Altri campi static;
* Valori restituiti da metodi static.

Un metodo static è un metodo della classe, e non dei suoi singoli oggetti,

In un metodo static, infatti, non esiste il concetto di oggetto implicito, per cui non si può usare this.   
E’ possibile dunque usare campi e invocare altri metodi della classe, purché siano static.   
E’ possibile invece usare oggetti, anche della stessa classe, accedendo ai loro campi non static   
e invocando su di essi metodi non static, a patto di possedere un riferimento ad essi.  
Un metodo static può poi restituire oggetti di cui ha il riferimento o che esso stesso crea.

Per invocare un metodo static di una classe si può usare la seguente sintassi:

*NomeDellaClasse*.*nomeDelMetodo*(*args*)

All’interno di una classe che presenta il metodo static si può usare semplicemente *nomeDelMetodo*(argomenti).

*Es.*

class Studente{

String nome;

Definisco la classe *Studente,*   
che presenta il campo static *contatore*.

int matricola

static int contatore;

Studente(String n){

nome = n;

matricola = ++contatore;

}

Definisco il metodo static *quantiStudenti*(),  
che accede correttamente al campo static *contatore*. Trovandomi all’interno di una classe contenente questo campo static, posso accedere ad esso   
usando semplicemente *contatore*.

static int quantiStudenti(){

return contatore;

}

public static void main(String[] args){

Invoco il metodo static quantiStudenti().   
Trovandomi all’interno di una classe contenente questo metodo static, posso invocarlo   
usando semplicemente *quantiStudenti*().

Studente s1 = new Studente(“Mario”);

int quanti = quantiStudenti();

}

}

**PACKAGE**

Un package è un insieme di classi.

Può essere utile a:

* Raggruppare classi tra loro correlate;
* Evitare conflitti tra i nomi delle classi;
* Controllare l’accesso alle classi e ai loro membri.

Per mettere una classe dentro un package, come prima istruzione del file sorgente che definisce   
la classe da voler aggiungere al package bisogna scrivere:

package *nome\_package*;

Per convenzione, i nomi dei package sono scritti in minuscolo, separando ogni parola da un punto.   
E’ possibile mettere nello stesso package classi che si trovano in file sorgente diversi.

Chiamiamo *nome completamente qualificato* di una classe:

*nome\_package\_di\_appartenenza.NomeClasse*

Una classe può usare tutte le altre classi del suo stesso package e le altre classi di altri package   
che sono dichiarate public. Data una top-level class, e cioè, una classe che non è contenuta   
in altre classi, questa può essere:

* public, se si indica esplicitamente il modificatore public;
* non public, se si omette il modificatore public.

In una classe, per riferirsi a classi dello stesso package si può usare semplicemente il loro nome,   
mentre per riferirsi a classi public di package diversi va usato il loro nome completamente qualificato.

E’ possibile *importare* una classe public di un package scrivendo, prima della definizione della classe che userà la classe importata (e dopo l’eventuale istruzione package):

import *nome\_completamente\_qualificato\_classe\_da\_importare*

Per importare invece tutte le classi public di un package, si scrive:

import *nome\_package*.\*

Una volta importata una classe, è possibile riferirsi ad essa usando semplicemente il suo nome,   
anche se è di un package diverso da quello della classe che la usa. Nel caso in cui si importassero   
classi con lo stesso nome di package diversi, per riferirsi ad una precisa classe omonima va usato   
il suo nome completamente qualificato.

Alcuni package di sistema sono:

* java.lang: contiene le classi fondamentali del linguaggio, come String, Object…;
* java.io: contiene classi dedicate all’ingresso/uscita;
* java.net: contiene classi utili ad accedere alla rete.

Il package java.lang viene importato automaticamente (per questo motivo, per esempio, possiamo scrivere semplicemente String, invece di java.lang.String).

Le classi che non vengono messe esplicitamente dentro un package vanno a finire   
dentro un package di default, senza nome. Queste classi non possono essere usate da altre classi contenuti in package veri e propri, per cui la regola generale è di mettere sempre ogni classe   
dentro un package.

*Es.*

**A.java B.java**

package pack1 package pack1

**A.java**

Metto nel *pack1* la classe *A*, e importo   
la classe *pack2*.*C*. Definisco poi *A* che ha, come campi:

* Un oggetto *b* di tipo *B*.   
  Posso riferirmi alla classe *B* usando semplicemente il suo nome,   
  in quanto questa classe è   
  dello stesso package di *A*;
* Un oggetto *c* di tipo *C*.   
  Posso riferirmi alla classe *C* usando semplicemente il suo nome,   
  in quanto ho importato questa classe.

import pack2.C; import pack2.C;

public class A { class B {

B b = new B(); A a = new A();

C c = new C(); C c = new C();

} }

**C.java D.java**

package pack2 package pack2

import pack1.A; import pack2.C;

public class C { class D {

A a = new A(); A a = new A();

D d = new D(); C c = new C();

} }

**MODIFICATORI DI ACCESSO AI CAMPI E AI METODI DI UNA CLASSE**

E’ possibile applicare dei **modificatori di accesso** ai campi e ai metodi di una classe:

* private: il campo o il metodo può essere acceduto solo nella classe in cui è definito;
* protected: il campo o il metodo può essere acceduto nella classe in cui è definito, in tutte le classi dello stesso package e nelle sottoclassi dello stesso package o di package diversi;
* public: il campo o il metodo può essere sempre acceduto.

Se non si specifica alcun modificatore di accesso, il campo o il metodo è al cosiddetto livello package, e può essere acceduto nella classe in cui è definito e in tutte le classi dello stesso package.

*Es.*

**file Esempio.java**

package pack1;

public class Esempio {

private int v1;

int v2;

protected int v3;

public int v4;

private void m1(){

v1++;

v2++;

v3++;

v4++;

}

void m2(){}

protected void m3(){}

public void m4(){}

}

**file UsaStessoPackage.java**

package pack1;

Nella classe *UsaStessoPackage*, dato un oggetto   
di tipo *Esempio*:

* Non posso accedere al campo privato v1,   
  in quanto può essere acceduto solo nella classe in cui è definito;
* Posso accedere al campo a livello package v2, in quanto può essere acceduto in una classe   
  dello stesso package;
* Posso accedere al campo protected v3,   
  in quanto può essere acceduto in una classe dello stesso package;
* Posso accedere al campo public v4,   
  in quanto può essere sempre acceduto.

Stesso ragionamento per i metodi.

class UsaStessoPackage {

public static void main(String[] args) {

Esempio e = new Esempio();

int a;

// a = e.v1;

a = e.v2;

a = e.v3;

a = e.v4;

// e.m1();

e.m2();

e.m3();

e.m4();

}

}

I modificatori di accesso possono essere applicati anche ai costruttori. In particolare, attraverso un costruttore private è possibile creare una classe con al massimo un solo oggetto esistente.   
A tal proposito si può usare il cosiddetto **singleton pattern**, di cui vediamo un esempio:

Definisco una classe *MiaClasse* che ha:

* Come campi:
  + Un riferimento *m* ad un oggetto di tipo MiaClasse, che punterà all’unico oggetto esistente della classe. E’ private e static, dunque può essere acceduto solo in *MiaClasse*, ed è   
    un campo della classe, e non dei suoi singoli oggetti.
* Come metodi:
  + Un costruttore. E’ private, dunque può essere invocato solo in *MiaClasse*.
  + getIstanza(), che restituirà un riferimento all’unico oggetto della classe esistente, e se questo non esiste lo crea.   
    E’ public e static, dunque può essere sempre invocato, ed è un metodo della classe, e non dei suoi singoli oggetti.

public class MiaClasse {

private static MiaClasse m;

private MiaClasse() {

…

}

public static MiaClasse getIstanza() {

if (m == null){

m = new MiaClasse();

}

return m;

}

…

}

Il costruttore di default è public.

**CLASSI E FILE**

Come già detto, la regola che useremo noi è quella di definire un’unica classe per file sorgente,   
e il file sorgente deve avere lo stesso nome della classe. Supponiamo di avere una classe *Principale*, definita in un file *Principale.java*, che fa uso di un’altra classe *MiaClasse.*   
Andando a compilare *Principale.java*, succede questo:

1. Il compilatore si accorge che *Principale* usa *MiaClasse*;
2. Il compilatore cerca *MiaClasse.class*:
   1. se *MiaClasse.class* non c’è, allora compila *MiaClasse.java*;
   2. altrimenti, il compilatore confronta la data di ultima modifica di *MiaClasse.java*   
      e *MiaClasse.class*, e se *MiaClasse.java* è più recente, allora lo ricompila.

In realtà, la vera regola è che all’interno di un file sorgente è possibile definire più classi,   
ma al più una classe può essere indicata come public, e se c’è una classe public, il file sorgente   
deve avere lo stesso nome di tale classe. Andando a compilare il file sorgente, verrà prodotto   
un file .class per ogni classe in esso definita, avente il nome della classe stessa.   
Questo approccio, tuttavia, conviene usarlo solo se le classi non public vengono usate esclusivamente dalla classe public. Supponiamo infatti di avere un file *Tre*.*java* che definisce le classi private *Uno* e *Due* e la classe public *Tre*, un file *Quattro.java* che definisce la classe *Quattro*, e supponiamo   
che *Quattro* usi *Uno*. Andando a compilare *Quattro.java*, succede questo:

1. Il compilatore si accorge che *Quattro* usa *Uno*;
2. Il compilatore cerca *Uno.class*. Supponendo che questo non esista (e quindi, che *Tre.java*   
   non sia stato compilato in precedenza), il compilatore cerca allora *Uno.java* per compilarlo,   
   ma anche questo non esiste (visto che *Uno* è definita in *Tre.java*). La compilazione allora   
   termina con errore.

**CLASSI, FILE E FILESYSTEM**

Dato un file sorgente *NomeClasse.java* che definisce una classe *NomeClasse* appartenente   
ad un package *parte\_nome1*.*parte\_nome2*. … . *parte\_nomeN, NomeClasse.java* va messo   
in una sequenza di directory, una per ogni *parte\_nome,* fatta così:

…/*parte\_nome1*/*parte\_nome2*/ … /*parte\_nomeN*/*NomeClasse.java*

Per compilare *NomeClasse.java* bisogna posizionarsi nella directory che contiene la radice del package cui *NomeClasse* appartiene (e cioè, nella directory che contiene *partenome1*) e usare da terminale   
il comando javac, passando come argomento il path relativo di *NomeClasse.java*:

javac *parte\_nome1*/*parte\_nome2*/ … /*parte\_nomeN*/*NomeClasse.java*

Il file *NomeClasse.class* risultante verrà creato nella stessa directory del file sorgente.   
Per eseguirlo, bisogna posizionarsi nella cartella che contiene la radice del package   
cui *NomeClasse* appartiene e usare da terminale il comando java, passando come argomento   
il nome completamente qualificato di *NomeClasse*:

java *parte\_nome1*.*parte\_nome2*. … . *parte\_nomeN.NomeClasse*

Attraverso l’opzione -classpath è possibile indicare esplicitamente il classpath, e cioè   
l’insieme di cartelle e file .jar a partire dai quali il compilatore (eseguito attraverso il comando javac)   
e la JVM (eseguito attraverso il comando java) cercano le classi che servono. Di default, il classpath   
è uguale alla directory corrente.

**GESTIONE DELLA MEMORIA**

Una volta allocato un oggetto nello heap, è la JVM stessa ad occuparsi di deallocarlo una volta   
che questo non è più raggiungibile (e cioè, una volta che nel programma non si ha più un riferimento tramite cui è possibile accedere a tale oggetto), attraverso un componente detto **garbage collector**.   
Il meccanismo con cui avviene il recupero della memoria è detto *mark and sweep*: periodicamente,   
partendo dai riferimenti presenti nella memoria statica e nello stack di ogni thread dell’applicazione,   
il garbage collector accede agli oggetti riferiti, agli oggetti riferiti dagli oggetti riferiti, agli oggetti riferiti dagli oggetti riferiti dagli oggetti riferiti…, segnandosi così quali sono gli oggetti ancora raggiungibili. Terminato questo procedimento, gli oggetti che non sono stati segnati come raggiungibili   
sono dunque irraggiungibili, e quindi il garbage collector li dealloca.

Quando va in esecuzione il garbage collector, l’applicazione si sospende per un po’,   
portando ad un tempo di attesa che può essere fastidioso. Per ridurre questo problema,   
il garbage collector usa un cosiddetto *approccio generazionale*. Innanzitutto, lo heap   
viene diviso in tre aree:

* Young Generation, in cui vengono allocati i nuovi oggetti;
* Old Generation, in cui vengono allocati gli oggetti creati da molto;
* Permanent Generation, in cui la JVM alloca delle strutture dati necessarie al suo funzionamento.

La Young generation è a sua volta divisa in altre tree aree:

* Eden;
* Survivor 0 (S0);
* Survivor 1 (S1).

Ogni nuovo oggetto viene allocato nell’Eden. Quando l’Eden è pieno, avviene una cosiddetta   
minor garbage collection, in cui gli oggetti della Young Generation non più raggiungibili   
vengono deallocati, mentre quelli raggiungibili vengono spostati tutti in una tra S0 e S1.   
Una volta che un oggetto è sopravvissuto ad un certo numero di minor garbage collection,   
questo viene spostato nella Old Generation. Quando l’Old Generation è piena, avviene una cosiddetta major garbage collection, in cui gli oggetti della Old Generation non più raggiungibili vengono deallocati.

Sia nella minor che nella major garbage collection viene usato il meccanismo mark and sweep,   
ma in virtù della divisione dello heap nelle tre aree viste prima vengono sfruttate delle ottimizzazioni (che non vediamo) che permettono in particolare di rendere la minor garbage collection molto veloce.

Ogni classe eredita il metodo finalize(), avente la seguente signature:

protected void finalize()

Questo metodo viene invocato dal garbage collector su ogni oggetto un istante prima di deallocarlo.   
In alcuni casi allora può essere utile ridefinirlo, per effettuare qualcosa sull’oggetto prima   
che venga deallocato.

**ARRAY**

Un array è un particolare oggetto, ed è un contenitore di variabili, detti *elementi*, dello stesso tipo.   
Ogni elemento è identificato da un indice intero progressivo, e il primo elemento ha indice 0.

Per creare un riferimento ad un array di un certo *tipo*, la sintassi è la seguente:

*tipo*[] *riferimento*

L’operatore new permette effettivamente di creare nello heap un array di un certo *tipo*   
avente una certa *dimensione*, e restituisce un riferimento all’array creato. La sintassi è la seguente:

new *tipo*[*dimensione*]

E’ possibile inizializzare esplicitamente un array in questo modo:

new *tipo*[] {*elemento1*, *elemento2*, …, *elementoN*}

Se non si inizializza esplicitamente un array, ogni suo elemento assume un valore di default, che è:

* false per gli elementi di tipo boolean;
* 0 per gli elementi di tipo numerico;
* null per gli elementi di tipo riferimento.

E’ possibile accedere ad un elemento di un array attraverso l’operatore di selezione [].   
Quest’operatore controlla che non vengano sforati i limiti dell’array, sollevando nel caso l’eccezione ArrayIndexOutOfBoundsException.

Ogni array eredita il campo *length*, contenente la dimensione dell’array.

*Es.*

public class Studente{

private String nome;

private String cognome;

public Studente(String n, String c){

nome = n;

cognome = c;

}

Dichiaro un riferimento *s1* ad un array di tipo *Studente*,   
e lo inizializzo con il riferimento di un nuovo array di 5 elementi di tipo Studente. Ogni elemento, dunque, è un riferimento   
ad un oggetto di tipo *Studente*, inizializzato a null.

Assegno poi all’elemento di indice 0 dell’array appena creato il riferimento a un nuovo oggetto di tipo Studente.

}

Studente[] s1 = new Studente[5];

s1[0] = new Studente(“Benito”, “Musso”);

Dichiaro un riferimento *s2* ad un array di tipo *Studente* e lo inizializzo con il riferimento di un nuovo array di 2 elementi di tipo *Studente*, ognuno inizializzato con un riferimento ad un nuovo oggetto di tipo Studente.

Assegno poi a *s1* il valore di *s2*, e cioè   
il riferimento dell’array puntato da *s2*.   
*s1* e *s2*, dunque, punteranno allo stesso array.

Studente[] s2 = new Studente[2] {new Studente(“Ad”, “Hit),

new Studente(“Jos”, “Sta”)};

s2 = s1;

E’ possibile creare degli array multidimensionali, e cioè array che contengono riferimenti ad altri array. In questo caso, la prima dimensione specificata, obbligatoria, è quella dell’array più esterno,   
mentre le dimensioni successive, facoltative, sono quelle degli array via via più interni, e specificandole vengono creati anche tali array, altrimenti i rispettivi riferimenti valgono null.

*Es.*

Creo un array *m1* di 5 elementi di tipo int[], e cioè riferimenti ad un array di tipo int, ognuno dei quali punta ad un array di 10 elementi di tipo int.

Assegno il valore 33 all’elemento di indice 5 dell’elemento di indice 3 dell’array *m1*.

int[][] m1 = new int[5][10];

m1[3][5] = 33;

Creo un array *m2* di 5 elementi di tipo double[], e cioè riferimenti ad un array di tipo double, ognuno dei quali vale null.

Non posso assegnare dunque un valore ad un elemento dell’array riferito   
da uno degli elementi di *m2*, perché tale array non esiste.

Assegno poi ad ogni elemento di *m2* il riferimento ad un nuovo array di int,   
di dimensione variabile.

Assegno il valore 33 all’elemento di indice 5 dell’elemento di indice 3 dell’array *m1*.

double[][] m2 = new double[3][];

//m2[0][0] = 33

for(int i = 0; i < m2.length; i++){

m2[i] = new int[i+1];

}

**NB:** in futuro, quando non c’è rischio di confusione, oltre a non fare distinzione tra il riferimento   
e l’array riferito, nel caso di un array di riferimenti a oggetti diremo per semplicità che è un array   
di oggetti, tenendo a mente però che l’array contiene effettivamente riferimenti   
(*es*. se abbiamo “Studente s1 = new Studente[3] {…}”, diremo per semplicità che “creiamo un array *s1*   
di oggetti *Studente*”);

Alcuni metodi utili per lavorare sugli array sono i seguenti:

* **System.arraycopy(*src*, *srcPos*, *dest*, *destPos*, *length*):** copia *length* elementi dell’array *src*   
  a partire dall’indice *srcPos* nell’array *dest* a partire dall’indice *destPos.*
* **Arrays.sort(*a*):** ordina l’array *a* in senso crescente di valore;
* **Arrays.binarySearch(*a*, *key*):** effettua una ricerca binaria nell’array *a*, restituendo l’indice dell’elemento di valore *key*, un valore negativo se questo non è presente.   
  *a* deve essere ordinato in senso crescente/decrescente di valore.

**STRING**

Una String è un oggetto della classe String, e rappresenta una sequenza di caratteri.   
Ogni carattere in una String è identificato da un indice intero progressivo, e il primo carattere   
ha indice 0.

Una volta creato un riferimento ad una String:

String *var*

per creare una String nello heap si può inizializzare/assegnare a tale riferimento un letterale stringa:

*var* = “*caratteri*”

In questo modo, verrà creata una String di valore “*caratteri*”, e *var* punterà a tale String.

Una String è immutabile (e assegnando a un riferimento a una String un altro letterale stringa,   
verrà semplicemente creata un’altra String che ha come valore tale letterale stringa,   
e il riferimento punterà a questa nuova String, smettendo così di puntare alla vecchia).

E’ possibile concatenare due String attraverso l’operatore +, che crea una nuova String   
avente come valore il risultato della concatenazione delle due String date e restituisce la String creata. Nel caso in cui un operando fosse una String e l’altro non lo fosse, quest’ultimo viene prima convertito in una String.

*Es.*

String s1 = “Ciao”;

Creo una String *s3* che ha come valore la concatenazione   
tra la String *s1*, il letterale stringa “ “ e la String *s2*.

Creo una String *t* che ha come valore la concatenazione   
tra la String *r* e l’int *v*, convertito prima in una String.

L’operatore + è associativo a sinistra. Nel creare la String *y*, dunque, il suo valore si ottiene eseguendo prima *v* + *w*, che è una somma di int,   
e poi concatenando l’int risultante, convertito prima in una String,   
alla String *r*.

String s2 = “Mondo”

String s3 = s1 + “ “ + s2;

int v = 4;

String r = “Rocky”;

String t = r + v;

int w = 5;

String z = v + w + r;

E’ possibile creare una String partendo da un array di char, passando tale array a un costruttore   
della classe String. La String creata avrà, come valore, la sequenza di caratteri contenuta nell’array.

*Es.*

Creo una String passando al costruttore l’array *ar*.   
La String creata avrà così come valore “Ciao”.

char[] ar = new char[]{‘C’, ‘i’, ‘a’, ‘o’};

String st = new String(ar);

Alcuni metodi utili della classe String (che dunque si possono invocare su ogni String) sono i seguenti:

* **substring(*startIndex, endIndex*):** restituisce una nuova String che ha come valore   
  la porzione della String implicita compresa tra l’indice *startIndex* incluso   
  e l’indice *endIndex* escluso;
* **length():** restituisce la lunghezza della String implicita;
* **chartAt(*i*):** restituisce il char che si trova all’indice *i* nella String implicita.
* **equals(*str*):** confronta la String implicita con la String *str*, restituendo true se sono uguali   
  (e cioè, se sono uguali carattere per carattere), false altrimenti.

**METODO MAIN IN DETTAGLIO**

Il metodo main deve accettare un unico argomento di tipo String[], e cioè un array di String.   
Queste String sono fornite da terminale.

Più classi della stessa applicazione possono avere il metodo main, specificando a seguito   
della compilazione quale file .class eseguire, e dunque quale main eseguire. Questo può servire   
a due scopi:

* Per avere un main di prova;
* Per fare in modo che l’applicazione possa funzionare in modalità diverse, ognuna identificata   
  da un main.

**EREDITARIETA’**

Il **meccanismo dell’ereditarietà** permette di definire una classe a partire da un’altra classe   
già esistente. La nuova classe, detta sottoclasse o classe derivata, eredita i campi e i metodi   
della classe già esistente, detta superclasse o classe base, e per definirla basta esprimere   
le sole differenze rispetto alla superclasse, aggiungendo nuovi campi e metodi e/o ridefinendo   
i metodi ereditati. I campi e i metodi ereditati saranno accessibili o meno dalla sottoclasse   
secondo quanto specificato dai rispettivi modificatori di accesso.

La sintassi per dichiarare una classe *B* sottoclasse di un’altra classe *A* è la seguente:

class B extends A {

…

}

Una classe può avere può avere una sola superclasse e più sottoclassi. La relazione semantica   
che si stabilisce tra una sottoclasse *B* e la sua superclasse *A* è del tipo “*B* *è un* *A*”.   
Graficamente, le classi si indicano attraverso dei rettangoli con bordo continuo, e per indicare   
che una classe è sottoclasse di un’altra si usa una freccia che va dalla sottoclasse alla superclasse.

*Es.*

*Immagine che contiene calligrafia, linea, Carattere, schizzo

Descrizione generata automaticamente*

Se una superclasse è dotata di soli costruttori con argomenti, allora ogni sua sottoclasse   
deve essere dotata di almeno un costruttore, e ogni costruttore della sottoclasse,   
come prima istruzione deve invocare un costruttore della superclasse, in questo modo:

super(*argomenti\_costruttore*)

Se una superclasse è dotata di un costruttore senza argomenti, allora una sua sottoclasse può anche   
non definire alcun costruttore o non invocare esplicitamente un costruttore della superclasse,   
e in automatico verrà invocato il costruttore della superclasse senza argomenti.

La parola chiave super permette inoltre di invocare in una sottoclasse un metodo della superclasse dopo che questo è stato ridefinito nella sottoclasse stessa, in questo modo:

super.*metodo*(*argomenti\_metodo*)

*Es.* realizziamo la seguente gerarchia di classi:

*Immagine che contiene calligrafia, Carattere, testo, linea

Descrizione generata automaticamente*

Un ContoBancario è dotato di:

* Un intestatario;
* Un bilancio;
* Un identificatore univoco;

Le operazioni possibili su un ContoBancario sono:

* Depositare;
* Prelevare;
* Trasferire;
* Conoscere le caratteristiche;
* Avere una rappresentazione in formato stringa.

La classe ContoBancario può essere allora definita in questo modo:

public class ContoBancario {

private double bilancio;

private final String intestatario;

private final int numero;

Il campo static *prossimo* conterrà il *numero*   
del prossimo oggetto *ContoBancario*, e verrà incrementato ogni volta che viene creato un nuovo oggetto *ContoBancario*.

private static int prossimo = 1;

public ContoBancario(String i){

intestatario = i;

numero = prossimo++;

}

public double getBilancio(){

return bilancio;

}

public String getIntestatario(){

return intestatario;

}

public int getNumero(){

return numero;

}

public void deposita(double d){

bilancio += d;

}

public void preleva(double d){

bilancio -= d;

}

public void trasferisci(ContoBancario c, double d){

preleva(d);

c.deposita(d);

}

public String toString(){

return “intestatario: “ + intestatario +

“, numero: “ + numero +

“, bilancio: “ + bilancio;

}

}

Un ContoConInteressi è un ContoBancario dotato in più di un tasso di interesse.  
L’operazione aggiuntiva possibile su un ContoConInteressi è aggiungere al bilancio   
gli interessi maturati.

La classe ContoConInteressi sottoclasse di ContoBancario può essere allora definita in questo modo:

public class ContoConInteressi extends ContoBancario {

Definisco la classe ContoConInteressi sottoclasse di ContoBancario. Questa, dunque, eredita tutti i campi e i metodi della superclasse, e per definirla basta esprimere le sole differenze rispetto ad essa.

private final double tasso;

La superclasse *ContoBancario* è dotata   
solo di un costruttore con argomenti,   
per cui la sua sottoclasse *ContoConInteressi* deve essere dotata di almeno un costruttore,   
e ogni costruttore della sottoclasse,   
come prima istruzione, deve invocare il costruttore della superclasse.

public ContoConInteressi(String i, double t){

super(i);

tasso = t;

}

public double getTasso(){

return tasso;

}

Definisco aggiungiInteressi() che, tra le varie cose, invoca il metodo getBilancio() ereditato dalla superclasse.  
**NB**: uso questo metodo per recuperare *bilancio*,   
in quanto *bilancio* è un campo private ereditato dalla superclasse, e quindi non è accessibile nella sottoclasse (mentre getBilancio() è stato definito nella superclasse, in cui si può accedere a tutti i campi private di tale classe).

public void aggiungiInteressi(){

double q = getBilancio() \* tasso;

deposita(q);

}

public void toString(){

String s = super.toString();

Ridefinisco il metodo toString() che, tra le varie cose,   
invoca il metodo toString() della superclasse.   
Avendo ridefinito il metodo, in questa classe, l’identificatore toString() del metodo ridefinito nella sottoclasse nasconde l’identificatore toString() del metodo della superclasse. Per invocare allora il metodo toString() della superclasse si può usare super.toString().

return s + “, tasso: “ + tasso;

}

}

Un ContoGratuito è un ContoBancario in cui un certo numero di operazioni sono gratuite, e poi si comincia a pagare.

La classe ContoGratuito sottoclasse di ContoBancario può essere allora definita in questo modo:

public class ContoGratuito extends ContoBancario {

private static final double COSTO\_OPERAZIONE = 1.0;

private final int operazioniGratuite;

private int operazioniEseguite;

public ContoGratuito(String i, int n) {

super(i);

operazioniGratuite = n;

}

public int operazioniGratuiteRimanenti() {

return operazioniGratuite – operazioniEseguite;

}

public void deposita(double d) {

operazioniEseguite++;

super.deposita(d);

}

public void preleva(double d) {

operazioniEseguite++;

super.preleva(d);

}

public void assegnaCosti() {

if(operazioniEseguite > operazioniGratuite){

double importo = (operazioniEseguite

– operazioniGratuite) \* COSTO\_OPERAZIONE;

super.preleva(importo);

}

operazioniEseguite = 0;

}

public String toString() {

String s = super.toString();

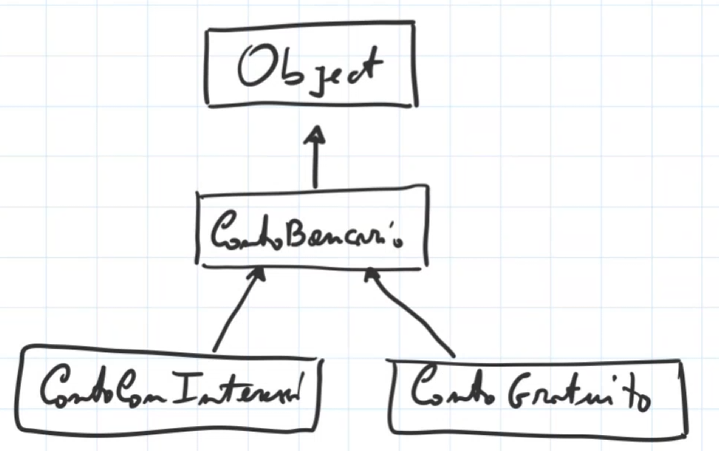
return s + “, op. gratuite: “ + operazioniGratuiteRimanenti();

}

}

Esiste una classe di sistema Object. Una classe che non dichiara di essere sottoclasse di un’altra   
è implicitamente sottoclasse della classe Object.

*Es.*



La classe Object è dotata di alcuni metodi, che vengono dunque ereditati da tutte le classi.  
Questi metodi si dividono in due categorie:

* Metodi per la sincronizzazione tra thread (vedremo più avanti);
* Metodi di utilità generale.

Alcuni metodi di utilità generale sono i seguenti:

* **getClass():** restituisce un particolare oggetto che indica la classe dell’oggetto implicito.   
  Gli oggetti restituiti da questo metodo si possono confrontare con == e !=,   
  per vedere se i relativi oggetti impliciti rispettivamente sono o meno della stessa classe.
* **public boolean equals(Object o):** confronta l’oggetto implicito con l’Object *o*, restituendo true se i due oggetti sono *uguali*, false altrimenti.

Nella definizione della classe Object, il metodo restituisce true se il riferimento all’oggetto implicito usato per invocare il metodo e il riferimento *o* sono uguali, e quindi se puntano allo stesso oggetto. Il concetto di *uguale*, tuttavia, può essere diverso da classe a classe,   
per cui ogni classe può ridefinire questo metodo secondo la sua interpretazione.

* **public int hashcode():** restituisce l’hashcode dell’oggetto implicito.

Nella definizione della classe Object, il metodo restituisce valori diversi per oggetti diversi. Quando una classe ridefinisce questo metodo, è bene che, se il metodo equals   
(eventualmente ridefinito) restituisce true per due oggetti, allora il metodo hashcode   
deve restituire lo stesso valore per tali oggetti.

* **public String toString():** restituisce la rappresentazione in formato String dell’oggetto implicito.

Nella definizione della classe Object, il metodo restituisce una String fatta così:

NomeClasseOggetto@hashCodeOggetto

Questo metodo viene tipicamente ridefinito da una classe, a seconda di quella che è rappresentazione in formato String desiderata. Il metodo toString() viene invocato automaticamente quando si concatena un oggetto a una String.

Un riferimento di tipo sottoclasse può essere convertito implicitamente in un riferimento   
di tipo superclasse diretta o indiretta.

*Es.*

ContoBancario cb;

ContoConInteressi cci = new ContoConInteressi(“Mario”, 0.01);

*cb* è un riferimento di tipo ContoBancario.   
Per assegnargli il valore di *cci*, un riferimento   
di tipo ContoConInteressi, bisogna prima convertire questo valorein un riferimento di tipo ContoBancario. Questa conversione avviene in modo implicito,   
in quanto si converte un riferimento di tipo sottoclasse in uno di tipo superclasse.

cb = cci;

Un riferimento di tipo superclasse può essere convertito solo esplicitamente in un riferimento   
di tipo sottoclasse diretta o indiretta, e solo se l’oggetto puntato è di tale sottoclasse o di una sottoclasse diretta o indiretta di tale sottoclasse. Se non è possibile convertire il riferimento,   
viene sollevata un’eccezione ClassCastException.

*Es.*

ContoConInteressi x = new ContoConInteressi(“Peach”, 0.04);

ContoBancario y = x;

*z* è un riferimento di tipo ContoConInteressi.   
Per assegnargli il valore di *y*, un riferimento   
di tipo ContoBancario, bisogna prima convertire questo valorein un riferimento di tipo ContoConInteressi. Questa conversione non avviene in modo implicito,   
in quanto si converte un riferimento di tipo superclasse   
in uno di tipo sottoclasse, per cui va fatta in modo esplicito, e si può fare, in quanto l’oggetto puntato da *y*   
è di tipo ContoConInteressi, e *z* è un riferimento di tipo ContoConInteressi.

ContoConInteressi z;

z = (ContoConInteressi) y;

L’operatore *oggetto* istanceof *NomeClasse* restituisce true se *oggetto* è della classe *NomeClasse*o di una sottoclasse diretta o indiretta di *NomeClasse*, false altrimenti. Può essere utile, per esempio, prima di eseguire un cast.

*Es.*

Se l’oggetto *r* è della classe ContoConInteressi o di una sottoclasse diretta o indiretta   
di ContoConInteressi, allora converto esplicitamente il riferimento *r* in un riferimento di tipo *ContoConInteressi*.

If (r istanceof ContoConInteressi){

ContoConInteressi s = (ContoConInteressi) r;

}

E’ possibile avere riferimenti di tipo diverso che puntano allo stesso oggetto:

* Il tipo del riferimento determina i campi e i metodi dell’oggetto a cui si può accedere   
  usando tale riferimento, e cioè solo i campi e i metodi della classe indicata   
  dal tipo del riferimento;
* Il tipo dell’oggetto determina i campi e i metodi dell’oggetto ai quali si accede effettivamente,   
  e cioè i metodi e i campi della classe dell’oggetto.

*Es.*

ContoConInteressi cci = new ContoConInteressi(“Luigi”, 0.01);

ContoBancario cb = cci;

Object o = cci;

Usando il riferimento *cb* di tipo ContoBancario  
che punta ad un oggetto di tipo ContoConInteressi:

* Non posso invocare aggiungiInteressi(), in quanto   
  la classe ContoBancario non presenta   
  questo metodo;
* Posso invocare deposita(), in quanto la classe ContoBancario presenta questo metodo,   
  e verrà invocato il metodo della classe ContoConInteressi (ereditato da ContoBancario);
* Posso invocare toString(), in quanto la classe ContoBancario presenta questo metodo (ereditato dalla classe Object), e verrà invocato il metodo   
  della classe ContoConInteressi.

Ragionamento simile usando il riferimento *o* di tipo Object che punta ad un oggetto di tipo ContoConInteressi.

cb aggiungiInteressi();

cb.deposita(30.0);

cb.toString();

o.aggiungiInteressi();

o.deposita(30.0);

o.toString();

In base a quanto visto finora, data una classe che definisce un certo metodo, e date delle sottoclassi dirette o indirette che ridefiniscono questo metodo, un riferimento di tipo superclasse   
può puntare a oggetti di tipo sottoclasse diretta o indiretta, e invocando tale metodo usando   
questo riferimento, verrà usata una ridefinizione diversa a seconda del tipo dell’oggetto puntato.   
Questo permette di usare un oggetto di tipo sottoclasse diretta o indiretta in tutto il codice   
scritto per una superclasse, e se su tale oggetto viene invocato un certo metodo, verrà usata   
la specifica ridefinizione fornita da tale sottoclasse diretta o indiretta.

*Es.*

ContoBancario x = new ContoBancario(“Mario”);

ContoGratuito y = new ContoGratuito(“Vaglini”, 100);

*y* è un oggetto di tipo ContoGratuito, sottoclasse   
di ContoBancario, per cui posso usarlo in tutto il codice scritto per ContoBancario.

Il metodo trasferisci() invocherà poi su *y* il metodo deposita(), e verrà usata la specifica ridefinizione di questo metodo fornita dalla sottoclasse ContoGratuito.

x.trasferisci(y, 200.0);

**CLASSI ASTRATTE** *11-10-24*

Una **classe astratta** è una classe che non può essere istanziata. Viene dunque usata solo   
come superclasse nella definizione di nuove classi. E’ possibile comunque creare riferimenti   
di tipo classe astratta.

Per dichiarare una classe come astratta si usa il modificatore abstract.

In una classe astratta è possibile dichiarare alcuni metodi come astratti, usando sempre il modificatore abstract. Questi metodi vengono solo dichiarati, e le sottoclassi concrete (e cioè, non abstract)   
della classe astratta devono fornire una definizione di questi metodi.

Un metodo concreto di una classe astratta può invocare metodi astratti della stessa classe.   
Questo non è un problema, in quanto, non essendo possibile creare un oggetto di tipo classe astratta,   
questo metodo concreto potrà essere invocato solo su un oggetto di una sottoclasse concreta   
diretta o indiretta della classe astratta, e quindi i metodi astratti invocati nel metodo concreto   
saranno stati prima o poi definiti.

*Es.*

Definisco la classe astratta OggettoGrafico.   
Non è possibile dunque creare un oggetto di tipo OggettoGrafico.

Questa classe presenta, tra i vari metodi,   
i metodi astratti disegna() e m1(),   
e il metodo concreto m2(), che invoca il metodo astratto m1().

**NB:** in una classe è possibile accedere a campi   
e a metodi della stessa classe dichiarati in qualsiasi punto della classe stessa.

abstract class OggettoGrafico {

int x, y;

void sposta(int nuovax, int nuova y) {

…

}

void m2(){

m1();

}

abstract void disegna();

abstract void m1();

}

class Bottone extends OggettoGrafico {

void disegna() {

Definisco la classe Bottone sottoclasse della classe astratta OggettoGrafico. Questa classe definisce tutti i metodi astratti della superclasse.

…

}

void m1() {

…

}

}

Creo un riferimento *og1* di tipo *OggettoGrafico*, che punta ad un nuovo oggetto di tipo Bottone.   
Usando questo riferimento invoco,   
tra i vari metodi, il metodo m2() ereditato   
da OggettoGrafico, che fa uso del metodo m1() dichiarato astratto in OggettoGrafico   
e definito poi in Bottone.

OggettoGrafico og1 = new Bottone();

og1.disegna();

og1.m2();

**CLASSI E METODI FINAL**

E’ possibile definire un campo o un metodo final, usando il modificatore final:

* Una **classe final** è una classe che non può avere sottoclassi.
* Un **metodo final** è un metodo di una classe che non può essere ridefinito   
  dalle sottoclassi dirette o indirette.

**INTERFACCE**

Un’**interfaccia** è una raccolta di metodi abstract. Questa definisce un protocollo di comportamento,   
al quale le classi possono decidere di aderire, implementando tale interfaccia.

Una classe può implementare una o più interfacce e contemporaneamente essere sottoclasse   
di un’altra classe. Graficamente, le interfacce si indicano attraverso rettangoli con bordo tratteggiato,   
e per indicare che una classe implementa un’interfaccia si usa una freccia che va dalla classe all’interfaccia.

*Es.*

Immagine che contiene testo, calligrafia, linea, Carattere

Descrizione generata automaticamente

La sintassi per definire un’interfaccia è la seguente:

interface *NomeInterfaccia* {

…

}

Similmente alle classi, un’interfaccia può essere public o non public. Inoltre:

* Tutti i campi dichiarati in un’interfaccia sono implicitamente public, final e static (e cioè, dell’interfaccia);
* Tutti i metodi dichiarati in un’interfaccia sono implicitamente public e abstract.

La compilazione di un file sorgente contenente la definizione di un’interfaccia è simile   
a quanto visto per le classi.

Per definire una classe che implementa delle interfacce, la sintassi è la seguente:

class *NomeClasse* implements *NomeInterfaccia1*, *NomeInterfaccia2*, …, *NomeInterfacciaN* {

…

}

Una classe che implementa un’interfaccia deve definire tutti i metodi dell’interfaccia. Se non lo fa,   
deve essere dichiarata abstract.

Per accedere ad un campo dell’interfaccia si può usare la seguente sintassi:

*NomeInterfaccia*.*nomeCampo*

Dentro una classe che implementa tale interfaccia o dentro una sottoclasse diretta o indiretta   
di tale classe si può accedere ad un campo dell’interfaccia scrivendo semplicemente il suo nome.

Non essendo una classe, non è possibile istanziare un’interfaccia, mentre è possibile creare   
un riferimento di tipo interfaccia, che può puntare ad oggetti di classi che implementano tale interfaccia o di sottoclassi dirette o indirette di tali classi. Similmente a quanto visto nell’ereditarietà tra classi, usando un riferimento di tipo interfaccia è possibile invocare solo i metodi dell’interfaccia,   
mentre il metodo effettivamente invocato sarà quello della classe dell’oggetto puntato.

*Es.*

Definisco l’interfaccia Comparabile, che definisce il protocollo di comportamento di “essere comparabile con un altro oggetto”.

Questa interfaccia presenta, tra i membri:

* Il campo *giornoSimone*, che è implicitamente public, static e final;
* il metodo *compara()*, che è implicitamente public e abstract.

public interface Comparabile {

int giornoSimone = 19;

int compara(Object o);

}

public class Data implements Comparabile {

Definisco la classe Data, che implementa l’interfaccia Comparabile.

int giorno;

int mese;

int anno;

public Data(int g, int m, int a) {

giorno = g;

mese = m;

anno = a;

}

Definisco il metodo *isGiornoSimone*. Trovandomi all’interno di una classe che implementa l’interfaccia Comparabile, posso accedere ad un campo dell’interfaccia usando semplicemente il suo nome.

public boolean isGiornoSimone {

return giorno == giornoSimone

}

Definisco il metodo compara(Object *o*) dell’interfaccia Comparabile.

In questo metodo, dopo aver convertito *o* in un riferimento di tipo Data così da poter accedere a campi dichiarati in Data dell’oggetto passato come argomento,   
restituisco il risultato della comparazione   
tra l’oggetto implicito e l’oggetto passato   
come argomento.

public int compara(Object o) {

Data d = (Data) o;

if(anno != d.anno)

return anno;

if(mese != d.mese)

return mese;

return giorno;

}

}

Data d1 = new Data(10, 5, 1950);

Creo un riferimento *c1* di tipo interfaccia Compara,   
che punta ad un oggetto della classe Data,   
classe che implementa tale interfaccia.   
Usando questo riferimento posso invocare solo   
metodi dell’interfaccia, e il metodo effettivamente invocato sarà quello della classe dell’oggetto puntato, e quindi in questo caso verrà invocato compara()   
della classe Data.

Data d2 = new Data(11, 6, 2000);

Comparabile c1 = d1;

int r = c1.compara(d2);

Java fornisce diverse interfacce di sistema. Una di queste è **java.lang.Comparable**, che definisce   
proprio il protocollo di comportamento di “essere comparabile con un altro oggetto”.   
Questa interfaccia dichiara il metodo **int compareTo(*o*)**, che dovrà essere definito dalle classi   
che implementano tale interfaccia e che sarà usato per confrontare l’oggetto implicito con l’oggetto *o* passato come argomento, restituendo un valore < 0, = 0 o > 0 a seconda che l’oggetto implicito   
sia rispettivamente minore, uguale o maggiore dell’oggetto *o* passato come argomento.   
In particolare, un array di oggetti di un tipo classe che implementa questa interfaccia   
o che è sottoclasse diretta o indiretta di una classe che la implementa   
può essere ordinato invocando il metodo Arrays.sort(*a*), che ordina l’array *a* passato come argomento in senso crescente di valore, facendo uso proprio del metodo compareTo() della classe   
degli oggetti dell’array.

**TIPO ENUMERAZIONE**

In Java, un **tipo enumerazione** è una sottoclasse della classe Enum.

E’ possibile definire un tipo enumerazione in questo modo:

enum *tipo\_enumerazione* {

*ENUMERATORE0*(*args*0), *ENUMERATORE1*(*args1*), …, *ENUMERATOREN-1*(*argsN-1*);

*altri campi e metodi*

};

Ogni enumeratore di un tipo enumerazione sarà implicitamente un campo public, static e const,   
e sarà un riferimento di tale tipo enumerazione, che punterà in automatico ad un certo oggetto   
di tale tipo enumerazione, creato a sua volta automaticamente, passando come argomenti   
al costruttore quelli indicati tra parentesi.

Una volta creato un riferimento di un certo tipo enumerazione:

*tipo\_enumerazione* var;

E’ possibile inizializzarlo/assegnargli un enumeratore di tale tipo enumerazione, in questo modo:

var = *tipo\_enumerazione*.*ENUMERATORE*

Vediamo ora alcuni metodi utili che ogni tipo enumerazione eredita dalla classe Enum:

* **public String toString()**: restituisce una String contenente il nome dell’enumeratore   
  che il riferimento usato per invocare il metodo ha come valore;
* **public static *tipo\_enumerazione*[] values()**: dato il tipo enumerazione di cui è stato invocato   
  il metodo, restituisce un array contenente tutti gli enumeratori;
* **public static *tipo\_enumerazione* valueOf(String *s*):** dato il tipo enumerazione di cui   
  è stato invocato il metodo, restituisce l’enumeratore che ha, come nome, la String *s*.
* **public int ordinal():** restituisce il numero d’ordine dell’enumeratore che il riferimento usato   
  per invocare il metodo ha come valore (e cioè, la posizione di tale enumeratore   
  tra i vari enumeratori presenti all’interno della definizione del suo tipo enumerazione,   
  dove il primo enumeratore è in posizione 0).

**ECCEZIONI**

Il **meccanismo delle** **eccezioni** permette di rappresentare e gestire situazioni di errore   
in maniera separata dal codice “normale” dell’applicazione.

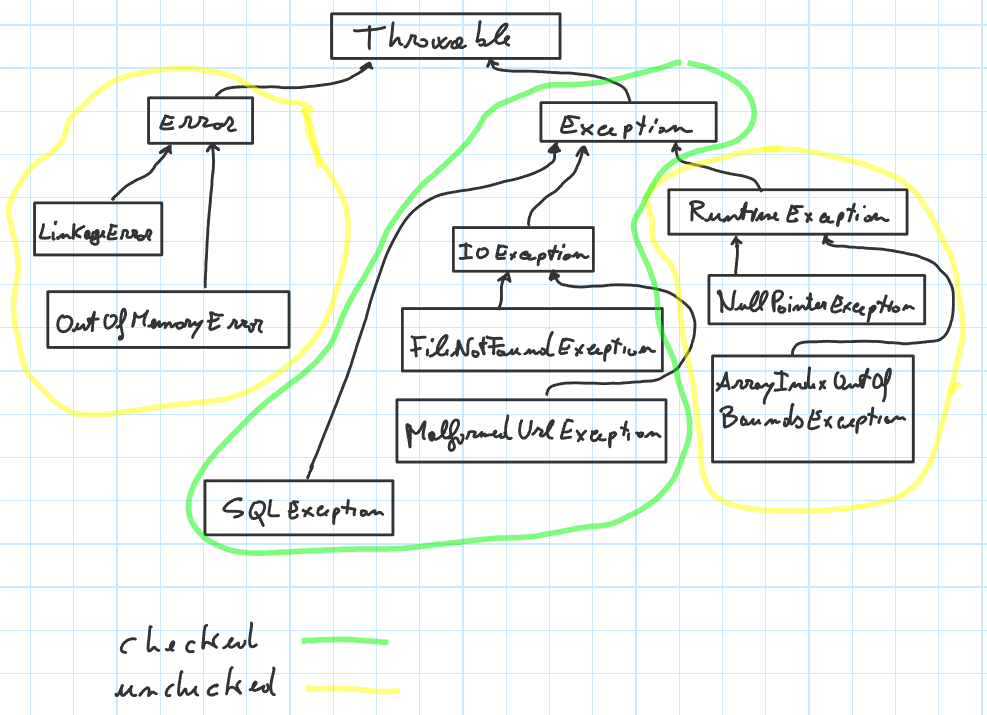
Alcune possibili situazioni di errore sono:

1. L’applicazione non riesce a scrivere in una porzione del filesystem;
2. Si verifica un problema mentre l’applicazione comunica con il mondo esterno attraverso la rete;
3. Si accede a un array usando un indice non valido;
4. Si invoca un metodo usando un riferimento null;
5. Si verifica un problema nella JVM.

A seconda della situazione di errore, l’approccio da adottare è diverso:

* Nelle situazioni 1) e 2) ha senso prevedere azioni di recupero:
  + Se l’applicazione non riesce a scrivere in una certa porzione del filesystem,   
    si può chiedere all’utente se vuole scrivere in un’altra porzione del filesystem;
  + Se si verifica un problema mentre l’applicazione comunica con il mondo esterno attraverso la rete, si può riprovare a comunicare dopo un po’ di tempo;
* Nelle situazioni 3), 4) e 5) non ha senso prevedere azioni di recupero:
  + Se si accede a un array usando un indice non valido o se si invoca un metodo usando   
    un riferimento null, significa che il programma è scritto in maniera sbagliata,   
    dunque bisogna interrompere l’esecuzione dell’applicazione e correggere il programma;
  + Se si verifica un problema nella JVM, tipicamente non c’è nulla che si può fare,   
    dunque bisogna interrompere l’esecuzione dell’applicazione.

Le situazioni di errore sono rappresentate dalle **eccezioni**. Queste sono oggetti di opportune classi organizzate in una gerarchia, di cui vediamo una parte qui sotto:



Le eccezioni si dividono in:

* **Checked** (oggetti delle classi racchiuse in verde): devono essere obbligatoriamente gestite dal programmatore;
* **Unchecked** (oggetti delle classi racchiuse in giallo): possono non essere gestite   
  dal programmatore.

Per ogni eccezione possiamo distinguere due fasi fondamentali:

* Il **lancio,** che avviene quando si verifica una situazione di errore;
* La **cattura**, in cui viene gestita la situazione di errore.

Per lanciare un’eccezione si usa l’istruzione throw, in questo modo:

throw *eccezione*

Per catturare e gestire un’eccezione si usa il costrutto try-catch, in questo modo:

try {

*istruzioni che possono lanciare eccezioni*

}

catch(*TipoEccezione1* *e1*){

*codice per la gestione della situazione di errore rappresentata da TipoEccezione1*

}

catch(*TipoEccezione2* *e2*){

*codice per la gestione della situazione di errore rappresentata da TipoEccezione2*

}

…

catch(*TipoEccezioneN* *eN*){

*codice per la gestione della situazione di errore rappresentata da TipoEccezioneN*

}

Ogni blocco catch ha come argomento un riferimento di un certo tipo.

Una volta lanciata un’eccezione nel blocco try, questa viene catturata dal blocco catch *corrispondente* all’eccezione lanciata, e cioè l’esecuzione salta al blocco tale per cui la classe dell’eccezione lanciata è la stessa classe o una sottoclasse diretta o indiretta del tipo dell’argomento del blocco stesso.   
In questo blocco è possibile poi accedere all’eccezione lanciata usando proprio l’argomento del blocco.   
In particolare, la classe Exception fornisce, tra la varie cose:

* Un costruttore senza argomenti;
* Un costruttore che accetta come argomento una String. Questa String può essere recuperata invocando sull’eccezione il metodo getMessage(), fornito sempre dalla classe Exception.

Lanciando dunque un’eccezione creata passando al costruttore una String,   
nel blocco catch corrispondente posso invocare il metodo getMessage() su tale eccezione   
usando l’argomento del blocco e recuperare così questa String.

I blocchi catch vengono analizzati in sequenza per trovare qual è quello corrispondente   
all’eccezione lanciata, e una volta trovato un blocco corrispondente, gli altri non vengono analizzati.   
Questo comporta che i blocchi catch vanno scritti in modo che i tipi degli argomenti rispettino  
l’ordine sottoclasse-superclasse, così che le eccezioni più specifiche siano catturate   
dal corretto blocco catch.

Terminata l’esecuzione del blocco catch corrispondente, oppure terminata l’esecuzione del blocco try senza che sia stata lanciata alcuna eccezione, l’esecuzione salta dopo l’ultimo blocco catch.

*Es.*

class Main {

public static void main(){

Leggo un int dallo Scanner *s* e lo assegno a *numero*:

* Se *numero* < 0, lancio un’eccezione di tipo Exception, creata passando al costruttore la String “Puttana”.

Nel blocco catch corrispondente invoco il metodo getMessage()   
sull’eccezione usando l’argomento del blocco, recuperando così   
la String, che stampo.

* Altrimenti, stampo “Brava”.

In entrambi i casi, alla fine stampo “Madonna”.

Scanner s = new Scanner(System.in);

int numero = s.nextInt();

try {

if(numero < 0){

throw new Exception(“Puttana”);

}

System.out.print(“Brava”);

}

catch(Exception e){

String ss = e.getMessage();

System.out.print(s);

}

System.out.println(“Madonna”);

}

}

E’ possibile definire un nuovo tipo di eccezione definendo una nuova classe come sottoclasse   
di un tipo di eccezione già esistente (tipicamente, Exception o sue sottoclassi dirette o indirette).  
Tipicamente, ogni nuovo tipo di eccezione deve essere dotato di due costruttori, uno senza argomenti   
e l’altro che accetta come argomento una String, che si limitano ad invocare i rispettivi costruttori   
della superclasse.

Le eccezioni del nuovo tipo saranno checked o unchecked, a seconda se le eccezioni   
del tipo superclasse sono rispettivamente checked o unchecked.

*Es.*

class PortaRottaException extends Exception{

Definisco un nuovo tipo di eccezione *PortaRottaException* sottoclasse di *Exception* dotato di due costruttori, uno senza argomenti   
e l’altro che accetta come argomento una String, che si limitano a invocare i rispettivi costruttori della superclasse.

Le eccezioni di tipo *PortaRottaException* saranno checked, in quanto le eccezioni del tipo superclasse *Exception* sono checked.

public PortaRottaException(){

super();

}

public PortaRottaException(String s){

super(s);

}

}

Se un’eccezione lanciata non viene catturata (e cioè, se viene lanciata fuori da un blocco try,   
oppure se viene lanciata in un blocco try ma non c’è nessun blocco catch corrispondente),   
l’esecuzione torna al chiamante, propagando ad esso l’eccezione. Se neanche questo la cattura, l’esecuzione torna al chiamante, propagando ad esso l’eccezione, e si continua così fintantoché   
un metodo non la catturerà, e l’esecuzione proseguirà da tale metodo.   
Se invece l’eccezione non viene catturata neanche dal main, il programma termina forzatamente.

Se in un metodo possono essere lanciate delle eccezioni che non vengono catturate   
(e quindi vengono propagate al chiamante), diremo che il metodo può *lanciare* tali tipi di eccezioni.   
Le eccezioni checked che possono essere lanciate in un metodo vanno indicate nella dichiarazione   
del metodo, in questo modo:

*nomeMetodo*(*argomenti*) throws *TipoEccezione1*, *TipoEccezione2*, …, *TipoEccezioneN* {

…

}

*Es.*

void salva() throws IOException{

Il metodo write() può lanciare eccezioni   
di tipo IOException, checked. Poiché   
nel metodo salva() non le catturo,   
devo indicare nella dichiarazione che questo può lanciare eccezioni di tipo IOException.

…

write();

…

}

Il costrutto try-catch può prevedere alla fine un blocco finally, che viene sempre eseguito.   
In particolare:

* Terminata l’esecuzione del blocco catch corrispondente all’eccezione lanciata nel blocco try, oppure terminata l’esecuzione del blocco try senza che sia stata lanciata alcuna eccezione, l’esecuzione salta dopo l’ultimo blocco catch, e quindi al blocco finally,   
  terminato il quale l’esecuzione continua dopo questo blocco;
* Se nel blocco try viene lanciata un’eccezione che non viene catturata da alcun blocco catch, prima viene eseguito il blocco finally, e poi l’esecuzione torna al chiamante,   
  propagando ad esso l’eccezione;

Tipicamente il blocco finally contiene del “codice di pulizia”, che deve essere eseguito a prescindere   
se si è verificata o meno una situazione di errore.

**THREAD**

Un **thread** è un flusso di esecuzione indipendente all’interno di un processo.

Un processo può avere più thread, ognuno dei quali esegue il proprio codice, ha il proprio stack  
e condivide le risorse del processo (tra cui la memoria, come lo heap), in modo da poter comunicare   
con gli altri thread del processo.

Quando viene mandato in esecuzione un programma, viene creato un processo con un unico thread, detto main thread, che esegue il codice “principale” del processo (nel caso di un programma Java,   
il main). Questo thread può poi creare altri thread dello stesso processo, che possono creare a loro volta altri thread dello stesso processo, e così via. Dato un thread che ne crea un altro,   
chiameremo *thread padre* il thread creante e *thread figlio* il thread creato.

In Java ogni thread è rappresentato da un oggetto della classe java.lang.Thread o di sue sottoclassi,   
detto **oggetto thread**. Il codice che il thread rappresentato da questo oggetto dovrà eseguire   
va specificato nella definizione del metodo public void run(). In particolare, esistono due approcci:

* Si definisce una sottoclasse di Thread, in cui si ridefinisce il metodo public void run()  
  ereditato dalla classe Thread. Fatto ciò, si può creare un oggetto thread come oggetto   
  della sottoclasse definita;
* Si definisce una classe che implementa l’interfaccia Runnable, in cui si definisce   
  il metodo public void run() dell’interfaccia Runnable. Fatto ciò, si può creare   
  un oggetto thread come oggetto della classe Thread, passando al costruttore un oggetto   
  della classe definita.

Per *attivare* un thread, e cioè fare in modo che il thread possa andare in esecuzione prima o poi,   
bisogna invocare sull’oggetto thread che lo rappresenta il metodo **start()** della classe Thread.

*Es.*

Definisco la classe UomoLupo sottoclasse di Thread.

public class UomoLupo extends Thread {

private String urlo;

public UomoLupo(String s) {

urlo = s;

}

public void run(){

Ridefinisco il metodo run() ereditato da Thread,   
in cui specifico il codice che i thread rappresentati dagli oggetti di tipo UomoLupo dovranno eseguire.

for(int i=0; i<10000; i++) {

System.out.println(urlo);

}

}

}

class Main {

public static void main(String[] args) {

Creo due oggetti thread *ul1* e *ul2* di tipo UomoLupo.

Invoco successivamente su   
di essi il metodo start(),   
attivando così i thread rappresentati da questi oggetti.

UomoLupo ul1 = new UomoLupo("Auu!");

UomoLupo ul2 = new UomoLupo("Bau!");

ul1.start();

ul2.start();

}

}

*Es.*

public class ComportamentoMostro implements Runnable {

Definisco la classe ComportamentoMostro che implementa l’interfaccia Runnable.

private String nome;

public ComportamentoMostro(String n) {

nome = n;

}

Definisco il metodo run() dell’interfaccia Runnable,   
in cui specifico il codice che dovranno eseguire   
i thread rappresentati dagli oggetti di tipo Thread   
creati passando al costruttore oggetti di tipo ComportamentoMostro.

public void run(){

for(int i=0; i<10000; i++) {

System.out.print(nome + ": Ah!");

}

}

}

class Main {

public static void main(String[] args) {

ComportamentoMostro cz1 = new ComportamentoMostro("Uomo lupo");

Nel main, creo un oggetto thread *t1* di tipo Thread, passando al costruttore un oggetto di tipo ComportamentoMostro.

Invoco successivamente su di esso il metodo start(), attivando così il thread rappresentato da questo oggetto.

Thread t1 = new Thread(cz1);

t1.start();

}

}

*Es.*

class A {

void m() {

for(int i = 0; i < 100; i++){

System.out.println(“Sesso”);

}

}

}

class Prova extends Thread{

int x;

void incrementa() {

x++;

}

public void run() {

A azzo = new A();

azzo.m();

…

}

}

class Main{

public static void main(String[] args){

Prova p1 = new Prova();

Nel main, creo un oggetto thread *p1* di tipo ProvaThread.

Invoco successivamente su di esso il metodo start(), attivando così il thread rappresentato da questo oggetto.

Attenzione a non confondere un thread dall’oggetto thread:   
nel main successivamente invoco sull’oggetto thread *p1*   
il metodo *incrementa*(). Questo metodo, dunque, verrà eseguito dal main thread, e non dal thread rappresentato da *p1*,   
che invece fa solo ciò che è specificato nel metodo run(),   
e quindi creare un nuovo oggetto di tipo A e farci varie cose.

p1.start();

p1.incrementa();

}

}

Durante la sua “vita”, un thread passa per vari stati:

Immagine che contiene testo, diagramma, schermata, linea

Descrizione generata automaticamente

Non appena viene creato un oggetto thread, il thread che rappresenta si trova nello stato *creato*.  
Una volta invocato sull’oggetto thread il metodo start(), il thread viene attivato e finisce   
nella coda *pronti*, contenente tutti thread che, se avessero a disposizione un core,   
potrebbero andare in esecuzione. Non vanno in esecuzione perché il numero di core è limitato,   
e ognuno di questi può eseguire un solo thread per volta.

Quando arriverà il momento, un componente della JVM, chiamato *scheduler*, sceglierà un thread   
tra quelli che si trovano nella coda pronti e lo manderà in esecuzione, togliendolo da tale coda   
e assegnandogli un core.

Ad un thread che si trova in esecuzione possono succedere tre cose:

* Può succedere che termini di eseguire il metodo run(). Lo scheduler allora   
  dealloca la memoria del thread, che passa così allo stato *terminato*, e al suo posto   
  manda in esecuzione un altro thread tra quelli presenti in coda pronti.   
  Una volta che un thread è terminato, non è possibile attivarlo nuovamente invocando   
  il metodo start() sull’oggetto thread che lo rappresenta;
* Può succedere che si sospenda in attesa che si verifichi un qualche evento (*es.* che un dispositivo sia pronto per un’operazione di I/O). Lo scheduler allora inserisce il thread sospeso in una coda *bloccati*, contenente tutti i thread che sono in attesa del verificarsi   
  di un certo evento, e al suo posto manda in esecuzione un altro thread tra quelli presenti   
  in coda pronti. Quando poi si sarà verificato l’evento atteso, lo scheduler risveglierà il thread sospeso, spostandolo dalla coda bloccati alla coda pronti, così che questo possa tornare   
  in esecuzione prima o poi;
* Può succedere che lo scheduler lo interrompa forzatamente. Lo scheduler allora inserisce   
  il thread interrotto nella coda pronti e al suo posto manda in esecuzione un altro thread tra quelli presenti in coda pronti.

Gli oggetti thread non sono soggetti alla garbage collection fintantoché i thread che rappresentano   
non sono terminati.

La JVM termina una volta che l’applicazione in esecuzione non ha più thread attivi.  
E’ possibile terminare forzatamente la JVM indipendentemente dal numero di thread attivi dell’applicazione in esecuzione invocando il metodo **System.exit()** in un qualunque thread dell’applicazione.

Ogni thread ha un nome che, ai fini di debug, può essere impostato quando si crea l’oggetto thread, passando al costruttore una String contenente il nome del thread. Questo nome può poi   
essere recuperato invocando sull’oggetto thread il metodo **getName()** della classe Thread.   
Di default, i thread avranno un nome del tipo *thread1*, *thread2*, …, *threadN*.

L’algoritmo di scheduling usato dallo scheduler è a priorità statica: viene mandato in esecuzione   
il processo con la priorità maggiore tra quelli presenti in coda pronti. Ogni thread ha infatti una priorità, che varia da Thread.MIN\_PRIORITY (1) a Thread.MAX\_PRIORITY (10). Di default, il main thread   
ha priorità Thread.NORM\_PRIORITY (5), e ogni thread, non appena viene creato, eredita la priorità   
del thread padre. E’ possibile successivamente cambiare il livello di priorità di un thread, invocando il metodo setPriority(*livello\_di\_priorità\_scelto*) sull’oggetto thread che lo rappresenta.

Questo algoritmo di scheduling è inoltre preemptive: se durante l’esecuzione di uno o più thread   
entra in coda prontiun altro thread con una priorità maggiore rispetto a quella di un thread   
in esecuzione, allora il thread in esecuzione a priorità minima viene interrotto forzatamente   
e inserito nella coda pronti, e al suo posto va in esecuzione il thread a priorità maggiore.

Vediamo ora alcuni metodi utili della classe Thread:

* **public static void sleep(long *millis*) throws InterruptedException:** sospende il thread   
  che esegue il metodo per *millis* millisecondi.
* **public static Thread currentThread():** restituisce un riferimento all’oggetto thread   
  relativo al thread che esegue il metodo;

**CORSE CRITICHE: MUTUA ESCLUSIONE** *24-10-24*

I thread di uno stesso processo condividono tra loro lo heap, e quindi gli oggetti. Tuttavia, quando   
più thread accedono allo stesso oggetto possono verificarsi delle **corse critiche**, e cioè situazioni in cui il risultato dell’insieme di operazioni svolte dai thread dipende da come si incastrano le loro esecuzioni.

Il problema delle corse critiche viene risolto attraverso il meccanismo dei **monitor.** Ogni oggetto   
ha associato un lock (lucchetto), e ogni volta che un thread deve accedere a dei campi di un oggetto che nel frattempo potrebbero essere modificati dal thread stesso o da altri thread:

1. Attende finché non acquisisce il lock dell’oggetto;
2. Accede ai campi dell’oggetto;
3. Rilascia il lock.

In questo modo, il thread accederà ai campi dell’oggetto in mutua esclusione.

Un primo modo per sfruttare il meccanismo dei monitor è dichiarando un **metodo synchronized**, attraverso il modificatore synchronized. Quando un thread invoca un metodo synchronized   
su un oggetto:

1. Si sospende finché non acquisisce il lock dell’oggetto;
2. Esegue il metodo synchronized;
3. Rilascia il lock dell’oggetto;

*Es.*

class ContoBancario {

Definisco la classe ContoBancario, che presenta  
il metodo deposita(). Questo è un metodo synchronized,   
in quanto il thread che lo invoca accede al campo *bilancio*, che nel frattempo potrebbe essere modificato   
da altri thread proprio attraverso questo metodo.

int bilancio;

public synchronized void deposita(){

bilancio += 10;

}

}

Definisco la classe Depositatore sottoclasse di Thread. Questa classe presenta, tra i vari campi, un riferimento *c* ad un oggetto di tipo ContoBancario.

public class Depositatore extends Thread {

ContoBancario c;

Depositatore(ContoBancario c){

this.c = c;

}

Ogni thread di tipo Depositatore, continuamente, invoca sull’oggetto riferito da *c* il metodo synchronized deposita(), per cui:

1. Si sospende finché non acquisisce il lock di tale oggetto;
2. Esegue il metodo synchronized;
3. Rilascia il lock di tale oggetto.

public void run(){

while(true){

c.deposita();

}

}

Nel main, creo un oggetto *conto*di tipo ContoBancario, e creo   
due oggetti thread *d1* e *d2*   
di tipo Depositatore, passando   
al costruttore lo stesso oggetto *conto*.  
Attivo successivamente i due thread, che dunque, quando andranno   
in esecuzione, accederanno allo stesso oggetto invocando su di esso   
il metodo deposita(), ma essendo questo un metodo synchronized,   
verrà eseguito accedendo all’oggetto   
in mutua esclusione, per cui non si avrà alcun problema di corse critiche.

public static void main(String[] args){

ContoBancario conto = new ContoBancario();

Depositatore d1 = new Depositatore(conto);

Depositatore d2 = new Depositatore(conto);

d1.start();

d2.start();

}

}

Un secondo modo per sfruttare il meccanismo dei monitor è attraverso un **blocco synchronized:**

synchronized(*oggetto*){

…

}

Quando un thread deve eseguire un blocco synchronized:

1. Si sospende finché non acquisisce il lock dell’oggetto riferito dal riferimento indicato   
   come argomento;
2. Esegue il blocco synchronized;
3. Rilascia il lock dell’oggetto;

Ci sono alcune situazioni in cui è necessario usare un blocco synchronized invece di un metodo synchronized:

1. Se non tutte le istruzioni di un metodo vanno eseguite accedendo all’oggetto implicito   
   in mutua esclusione;
2. Se bisogna acquisire il lock di un oggetto diverso dall’oggetto implicito (un oggetto della   
   stessa classe o di altre classi, oppure un array).

*Es.*

Definisco un metodo m()che presenta un blocco synchronized avente come argomento un riferimento all’oggetto implicito.   
In questo modo, quando un thread deve eseguire questo blocco:

1. Si sospende finché non acquisisce il lock di tale oggetto;
2. Esegue il blocco synchronized;
3. Rilascia il lock di tale oggetto.

Solo le istruzioni del metodo all’interno del blocco synchronized   
verranno dunque eseguite accedendo all’oggetto implicito   
in mutua esclusione, mentre le altre istruzioni del metodo possono essere eseguite in contemporanea con altri thread.

dsadsa

public void m(){

…

synchronized(this){

…

}

…

}

*Es.*

Definisco un metodo m2()che presenta un blocco synchronized avente come argomento un riferimento *p* ad un oggetto di tipo Parametro, diverso dall’oggetto implicito. In questo modo, quando un thread   
deve eseguire questo blocco:

1. Si sospende finché non acquisisce il lock di tale oggetto;
2. Esegue il blocco synchronized;
3. Rilascia il lock di tale oggetto.

dsadsa

void m2(Parametro p){

…

synchronized(p){

…

}

…

}

*Es.*

int[] v = new int[]{…};

Dichiaro un riferimento *v* ad un array di tipo *int*, e lo inizializzo con   
il riferimento di un nuovo array di tipo *int*.

Definisco inoltre un metodo m3()che presenta un blocco synchronized avente come argomento un riferimento *v* all’array di tipo int.   
In questo modo, quando un thread deve eseguire questo blocco:

1. Si sospende finché non acquisisce il lock dell’array;
2. Esegue il blocco synchronized;
3. Rilascia il lock dell’array.

dsadsa

void m3(){

…

synchronized(v){

…

}

…

}

Se un thread acquisisce il lock di un oggetto e successivamente invoca su di esso un metodo synchronized o deve eseguire un blocco synchronized che ha come argomento un riferimento a tale oggetto, il thread eseguirà direttamente il metodo/blocco synchronized, avendo già il lock dell’oggetto. Questo lock verrà rilasciato poi:

* Al termine dell’esecuzione del metodo/blocco synchronized che ha causato l’acquisizione   
  del lock;
* Se viene lanciata un’eccezione nel metodo/blocco synchronized che ha causato l’acquisizione del lock e non viene catturata.

E’ possibile dichiarare metodi static synchronized. Anche ciascuna classe ha associato un lock,  
e quando un thread invoca un metodo static synchronized di una classe, acquisisce il lock della classe, per cui più thread non possono eseguire in contemporanea metodi static synchronized   
della stessa classe. Un metodo static synchronized, dunque, serve affinché un thread acceda   
in mutua esclusione ad un campo static della classe.

*Es.*

Definisco la classe ContoBancario che presenta, tra i vari campi, il campo private *numero*   
e il campo static *nextNumero*, che conterrà   
il valore di *numero* del prossimo oggetto di tipo ContoBancario. Tuttavia, nel costruttore,   
non posso fare semplicemente:

numero = nextNumero;

nextNumero++;

Infatti, più thread potrebbero invocare   
il costruttore e modificare contemporaneamente   
*nextNumero*, portando così ad inconsistenze   
(e acquisire il lock dell’oggetto implicito non servirebbe a nulla, in quanto *nextNumero* è un campo della classe, e non di un singolo oggetto)   
Per garantire allora che l’accesso a *nextNumero* venga fatto in mutua esclusione, definisco   
il metodo static synchronized getNextNumero(),   
in cui incremento *nextNumero* e restituisco   
il valore di *numero* prima dell’incremento.   
In questo modo, quando un thread invoca   
questo metodo nel costruttore:

1. Si sospende finché non acquisisce il lock della classe ContoBancario;
2. Esegue il metodo synchronized;
3. Rilascia il lock della classe.

dsadsa

public class ContoBancario {

private int numero;

private static int nextNumero;

public ContoBancario(){

numero = getNextNumero();

}

private static synchronized int getNextNumero(){

return nextNumero++;

}

}

Un problema che può verificarsi quando si hanno almeno 2 thread e 2 lock è il **deadlock**.  
Un insieme di thread è in deadlock se ogni thread dell’insieme è sospeso finché non acquisisce   
il lock di un oggetto posseduto da un altro thread dello stesso insieme.

Un modo per evitare i deadlock è fare in modo che i lock siano acquisiti sempre nello stesso ordine   
(*es.* dati 2 oggetti *o1* e *o2*, si può fare in modo che ogni thread che deve acquisire il lock di questi acquisisca sempre prima il lock di *o1* e poi il lock di *o2*).

**COMUNICAZIONE TRA THREAD: SINCRONIZZAZIONE**

I thread di uno stesso processo condividono tra loro lo heap, e quindi gli oggetti, che dunque   
possono usare per comunicare fra loro. Questo, tuttavia, richiede che sia garantita la **sincronizzazione** tra i thread, e cioè che le operazioni dei thread sugli oggetti avvengano in un preciso ordine.   
Per garantire la sincronizzazione, ad ogni oggetto è associato un wait-set, e un thread che possiede   
il lock di tale oggetto può agire sul wait-set associato attraverso alcuni metodi offerti   
dalla classe Object:

* **public final void wait() throws InterruptedException:** il thread che lo invoca si sospende, rilascia il lock dell’oggetto implicito e viene aggiunto al wait-set di tale oggetto.   
  Quando viene risvegliato, prima di proseguire con la sua esecuzione, si sospende finché   
  non riacquisisce il lock dell’oggetto implicito.
* **public final void wait(long *timeout*) throws InterruptedException:** come wait()   
  senza argomenti, con la differenza che se dopo *timeout* millisecondi il thread è ancora   
  nel wait-set dell’oggetto implicito, viene risvegliato.
* **public final void notify():** risveglia un thread a caso tra quelli presenti nel wait-set   
  dell’oggetto implicito.
* **public final void notifyAll():** risveglia tutti i thread presenti nel wait-set dell’oggetto implicito.

Se un thread invoca wait(), notify() o notifyAll() su un oggetto di cui non possiede il lock   
viene sollevata un’eccezione di tipo IllegalMonitorStateException.

I metodi che usano wait(), notify() o notifyAll() tipicamente si dividono in due categorie   
(spesso coincidenti):

* Quelli che eseguono delle operazioni solo dopo che una certa condizione dell’oggetto implicito   
  è soddisfatta;
* Quelli che cambiano una delle condizioni dell’oggetto implicito controllate dai metodi sopra.

Questi metodi, dunque, hanno tipicamente una struttura del genere:

synchronized void faiQuandoCondizione() throws InterruptedException{

while(*condizione\_non\_soddisfatta*){

wait();

}

*operazioni da eseguire*

}

synchronized void cambiaCondizione(){

*cambio condizione*

notifyAll();

}

In un metodo che esegue delle operazioni solo dopo che una certa condizione dell’oggetto implicito  
è soddisfatta, fintantoché la condizione non è soddisfatta, viene invocato wait() sull’oggetto implicito.   
Il thread che la invoca, dunque, si sospende, rilasciando il lock dell’oggetto (permettendo così ad altri thread   
di invocare sull’oggetto implicito altri metodi di questa categoria o di quelli che cambiano una condizione)   
e venendo aggiunto al wait-set di tale oggetto. Questa condizione verrà ricontrollata ogni volta che il thread   
si risveglierà, riacquisendo il lock dell’oggetto, e quando finalmente sarà soddisfatta, il thread passerà ad eseguire   
le operazioni del metodo.

In un metodo che cambia una delle condizioni dell’oggetto implicito controllate dai metodi della categoria sopra,   
una volta cambiata la condizione, viene invocato notifyAll() sull’oggetto implicito. Tutti i thread presenti nel wait-set dell’oggetto implicito verranno così risvegliati, riacquisendo il lock dell’oggetto, e proseguiranno con la loro esecuzione.   
Tra i thread risvegliati ci saranno anche quelli che, durante l’esecuzione di un metodo della categoria sopra, si sono sospesi proprio perché la condizione, che ora è stata modificata, prima non era soddisfatta: questi thread, dunque, potranno così ricontrollare la condizione, comportandosi di conseguenza.

Una volta cambiata una condizione di un oggetto, per risvegliare un thread che si è sospeso   
ed è finito nel wait-set dell’oggetto perché tale condizione in precedenza non era soddisfatta,   
si può sempre invocare notifyAll() sull’oggetto, risvegliando così tutti i thread presenti nel wait-set.  
Invocare invece notify(), risvegliando così un solo thread tra quelli presenti nel wait-set,  
è un’ottimizzazione possibile se e solo se si verificano tutte le seguenti circostanze:

* La condizione dell’oggetto per cui i thread possono sospendersi è una sola;
* Solo uno dei thread presenti nel wait-set dell’oggetto può trarre beneficio dal cambiamento della condizione dell’oggetto.

**MODELLO DI COMUNICAZIONE TRA THREAD: PRODUTTORI-CONSUMATORI**

Un primo modello di comunicazione tra thread è il **modello produttori-consumatori**.

In questo modello si hanno *n* thread *produttori*, che producono informazioni, e *m* thread *consumatori*,  
che consumano queste informazioni. Il passaggio delle informazioni avviene attraverso un buffer.

Ogni produttore, ciclicamente:

* Produce un nuovo valore;
* Attende fintantoché il buffer non è pieno per inserirci il nuovo valore.

Ogni consumatore, ciclicamente:

* Attende fintantoché il buffer non è vuoto per estrarci un valore;
* Consuma il valore.

Il buffer:

* E’ in grado di contenere un numero limitato di valori;
* E’ condiviso tra produttori e consumatori.

Per prima cosa, allora, possiamo definire la classe Buffer, in cui gestiamo anche la mutua esclusione   
e la sincronizzazione tra i thread, in un modo del genere:

public class Buffer {

private static final int DEFAULT\_SIZE = 4;

private String[] vett;

private int testa, coda;

private int quanti;

public Buffer(){

this(DEFAULT\_SIZE);

}

public Buffer(int n) {

vett = new String[n];

}

public synchronized void inserisci(String s) throws InterruptedException {

while(pieno()) {

wait();

}

vett[coda] = s;

coda = (coda + 1) % vett.length;

quanti++;

notifyAll();

}

public synchronized String estrai() throws InterruptedException {

while(vuoto()) {

wait();

}

String t = vett[testa];

testa = (testa + 1) % vett.length;

    quanti--;

    notifyAll();

    return t;

}

public synchronized boolean pieno() {

    return quanti == vett.length;

  }

public synchronized boolean vuoto() {

return quanti == 0;

  }

}

La classe Buffer presenta, tra i vari campi, un riferimento ad un array, che punterà all’array, creato nel costruttore,   
che rappresenterà il buffer vero e proprio.

Definisco poi i metodi inserisci() ed estrai(). Questi sono metodi synchronized, in quanto un thread che li invoca   
accede a campi dell’oggetto implicito che nel frattempo potrebbero essere modificati da altri thread proprio attraverso   
questi metodi. Questi metodi, inoltre, inoltre, invocano wait(), che può lanciare eccezioni di tipo InterruptedException, checked. Tipicamente, a questo livello di codice, non si sa come gestire queste eccezioni, per cui conviene non catturarle (propagandole così al chiamante).

Nel metodo inserisci(), fintantoché il buffer è pieno, invoco wait() sull’oggetto implicito. Il thread produttore che la invoca, dunque, si sospende, rilasciando il lock dell’oggetto e venendo aggiunto al wait-set dell’oggetto. Questa condizione   
verrà ricontrollata ogni volta che il thread si risveglierà, riacquisendo il lock dell’oggetto, e quando finalmente sarà soddisfatta, effettuo l’inserimento nel buffer.

Avendo cambiato una condizione del buffer, e in particolare avendo reso il buffer sicuramente non vuoto, invoco notifyAll()   
sull’oggetto implicito, così da risvegliare eventuali thread consumatori presenti nel wait-set dell’oggetto perché, provando   
ad estrarre un valore dal buffer, questo era vuoto.

Nel metodo estrai(), fintantoché il buffer è vuoto, invoco wait() sull’oggetto implicito. Il thread consumatore che la invoca, dunque, si sospende, rilasciando il lock dell’oggetto e venendo aggiunto al wait-set dell’oggetto. Questa condizione   
verrà ricontrollata ogni volta che il thread si risveglierà, riacquisendo il lock dell’oggetto, e quando finalmente sarà soddisfatta, effettuo l’estrazione dal buffer.

Avendo cambiato una condizione del buffer, e in particolare avendo reso il buffer sicuramente non pieno, invoco notifyAll() sull’oggetto implicito, così da risvegliare eventuali thread produttori presenti nel wait-set dell’oggetto perché, provando   
ad inserire un valore nel buffer, questo era pieno.

Definisco infine i metodi pieno() e vuoto(), in cui controllo rispettivamente se il buffer è pieno o vuoto. Anche questi   
sono metodi synchronized, in quanto un thread che li invoca accede a campi dell’oggetto che nel frattempo potrebbero essere modificati da altri thread attraverso i metodi inserisci() ed estrai().

**NB:** se i metodi pieno() e vuoto() fossero stati *private*, e se venissero chiamati esclusivamente dentro i metodi inserisci()   
ed estrai(), non sarebbe stato necessario dichiararli synchronized, in quanto un thread che li invoca accede a campi dell’oggetto   
che nel frattempo non possono essere modificati da altri thread, visto che questo thread, eseguendo inserisci() o estrai(), possiede il lock dell’oggetto del quale accede ai campi.

dsadsa

Le classi Produttore e Consumatore possono essere fatte in un modo del genere:

public class Produttore extends Thread {

private Buffer b;

public Produttore(Buffer b) {

this.b = b;

}

public void run(){

int c = 0;

try {

while(true) {

sleep((long)(Math.random()\*1000));

b.inserisci(getName() + ": " + c++);

}

}

catch(InterruptedException ie) {

System.out.println("Sono stato interrotto");

}

}

}

A questo punto, una classe di prova può essere fatta in un modo del genere:

Le classi Produttore e Consumatore, sottoclassi di Thread, presentano entrambe come campo un riferimento *b* ad un oggetto   
di tipo Buffer, che punterà all’oggetto Buffer condiviso, passato al costruttore, su cui i thread produttori rappresentati   
dagli oggetti thread di tipo Produttore lavoreranno insieme ai thread consumatori rappresentati dagli oggetti thread di tipo Consumatore.

Ridefinisco poi per entrambe le classi il metodo run(), specificando il codice che i thread di tipo Produttore e Consumatore dovranno eseguire. In questo caso, ciclicamente:

* Nel metodo run() di Produttore, inserisco nel buffer un valore, invocando il metodo inserisci() sull’oggetto *b*   
  di tipo Buffer.
* Nel metodo run() di Consumatore, estraggo dal buffer un valore, invocando il metodo estrai() sull’oggetto *b* di tipo Buffer.

Poiché inserisci() ed estrai() possono lanciare eccezioni di tipo InterruptedException, checked, le catturo e gestisco   
attraverso un blocco try-catch.

dsadsa

public class Consumatore extends Thread {

private Buffer b;

public Consumatore(Buffer b) {

this.b = b;

}

public void run(){

try {

while(true) {

sleep((long)(Math.random()\*1000));

String s = b.estrai();

System.out.println(getName() + ": ho estratto \"" + s + "\"");

}

}

catch(InterruptedException ie) {

System.out.println("Sono stato interrotto");

}

}

}

import java.util.Scanner;

public class Prova {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Quanti produttori? ");

int p = sc.nextInt();

System.out.println("Quanti consumatori? ");

int c = sc.nextInt();

System.out.println("Dimensione del buffer? ");

int b = sc.nextInt();

Buffer buf = new Buffer(b);

for(int i=0; i<p; i++) {

new Produttore("Produttore " + i, buf).start();

}

for(int i=0; i<c; i++) {

new Consumatore("Consumatore " + i, buf).start();

}

}

}

Nel main, creo un oggetto *buf* di tipo Buffer, e degli oggetti thread di tipo Produttore e Consumatore, passando al costruttore   
lo stesso oggetto *buf*. Attivo successivamente i thread rappresentati da questi oggetti thread, che dunque, quando andranno   
in esecuzione, accederanno allo stesso oggetto di tipo Buffer, condividendo così lo stesso buffer.

dsadsa

**MODELLO DI COMUNICAZIONE TRA THREAD: GESTORE DI RISORSE EQUIVALENTI** *25-10-24*

Un secondo modello di comunicazione tra thread è il **modello gestore di risorse equivalenti**.

In questo modello si hanno *n* risorse equivalenti, e cioè uguali tra loro, *m* thread *utilizzatori*,   
che utilizzano queste risorse. Queste risorse sono gestite attraverso un *gestore*.

Ogni risorsa può trovarsi in uno di questi due stati:

* *Libera*, e cioè non acquisita da nessuno;
* *Occupata*, e cioè acquisita da un utilizzatore.

Ogni utilizzatore, attraverso il gestore, può:

* Richiedere di acquisire una risorsa libera (senza specificare quale, essendo le risorse equivalenti), attendendo fintantoché non c’è almeno una risorsa libera;
* Rilasciare una risorsa da lui occupata.

Per prima cosa, allora, possiamo definire la classe Gestore, in cui gestiamo anche la mutua esclusione   
e la sincronizzazione tra i thread, in un modo del genere:

public class Gestore {

private boolean[] risOccupate;

private int numOccupate;

public Gestore(int n) {

risOccupate = new boolean[n];

}

public synchronized int richiesta() throws InterruptedException {

while(numOccupate == risOccupate.length){

wait();

}

for(int i = 0; i< risOccupate.length; i++){

if(risOccupate[i] == false){

risOccupate[i] = true;

numOccupate++;

return i;

}

}

}

public synchronized void rilascio(int k) {

risOccupate[k] = false;

numOccupate--;

notify();

}

}

La classe Gestore presenta, tra i vari campi, un riferimento ad un array di boolean, che punterà all’array di boolean,   
creato nel costruttore, che verrà usato per rappresentare lo stato di ogni risorsa.

Definisco poi i metodi richiesta() e rilascio(). Questi sono metodi synchronized, in quanto un thread che li invoca   
accede a campi dell’oggetto implicito che nel frattempo potrebbero essere modificati da altri thread proprio attraverso   
questi metodi. Il metodo richiesta(), inoltre, invoca wait(), che può lanciare eccezioni di tipo InterruptedException, checked. Tipicamente, a questo livello di codice, non si sa come gestire queste eccezioni, per cui conviene non catturarle (propagandole così al chiamante).

Nel metodo richiesta(), fintantoché non c’è almeno una risorsa libera, invoco wait() sull’oggetto implicito. Il thread utilizzatore   
che la invoca, dunque, si sospende, rilasciando il lock dell’oggetto e venendo aggiunto al wait-set dell’oggetto.   
Questa condizione verrà ricontrollata ogni volta che il thread utilizzatore si risveglierà, riacquisendo il lock dell’oggetto,   
e quando finalmente sarà soddisfatta, assegno al thread utilizzatore una risorsa libera.

Nel metodo rilascio(), rilascio la risorsa occupata avente l’indice passato come argomento. Avendo cambiato una condizione del gestore, e in particolare essendoci adesso sicuramente almeno una risorsa libera, invoco notify() sull’oggetto implicito,   
così da risvegliare un eventuale thread utilizzatore sospeso nel wait-set dell’oggetto perché, provando a richiedere una risorsa, non ce n’era alcuna libera.

dsadsa

La classe Utilizzatore può essere fatta in un modo del genere:

public class Utilizzatore extends Thread {

private Gestore g;

private int numRichieste;

private static final int DEFAULT\_NUM\_RICHIESTE = 100;

public Utilizzatore(Gestore g, int r) {

this.g = g;

this.numRichieste = r;

}

public Utilizzatore(Gestore g) {

this(g, DEFAULT\_NUM\_RICHIESTE);

}

public void run(){

try {

for(int i=0; i<numRichieste; i++) {

int k = g.richiesta();

System.out.println(getName() + ": ho ottenuto la risorsa " + k);

sleep((long)(Math.random()\*1000));

g.rilascio(k);

System.out.println(getName() + ": ho rilasciato la risorsa " + k);

}

}

catch(InterruptedException ie) {

System.out.println("Sono stato interrotto");

}

}

}

La classe Utilizzatore, sottoclasse di Thread, presenta tra i vari campi un riferimento *g* ad un oggetto di tipo Gestore, che punterà all’oggetto Gestore attraverso il quale i thread utilizzatori rappresentati dagli oggetti thread di tipo Utilizzatore   
richiederanno di acquisire una risorsa libera o rilasceranno una risorsa da loro occupata.

Ridefinisco poi il metodo run(), specificando il codice che i thread di tipo Utilizzatore dovranno eseguire. In questo caso,   
per un certo numero di volte, richiedo di acquisire una risorsa libera invocando il metodo richiesta() sull’oggetto *g* di tipo Gestore, dopodiché la rilascio, invocando il metodo rilascia() sullo stesso oggetto. Poiché richiesta() e rilascia() possono lanciare eccezioni di tipo InterruptedException, checked, le catturo e gestisco attraverso un blocco try-catch.

dsadsa

E POI PARLI DI QUANDO E’ POSSIBILE USARE NOTIFY AL POSTO DI NOTIFYALL PER OTTIMIZZAZIONE

A questo punto, una classe di prova può essere fatta in un modo del genere:

import java.util.Scanner;

public class Prova {

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.println("Quanti utilizzatori?");

int u = sc.nextInt();

System.out.println("Quante risorse?");

int r = sc.nextInt();

Gestore g = new Gestore(r);

for(int i=0; i<u; i++){

new Utilizzatore("U"+i, g).start();

}

}

}

Nel main, creo un oggetto *g* di tipo Gestore, e degli oggetti thread di tipo Utilizzatore, passando al costruttore   
lo stesso oggetto *g*. Attivo successivamente i thread rappresentati da questi oggetti thread, che dunque, quando andranno   
in esecuzione, accederanno allo stesso oggetto di tipo Gestore, condividendo così lo stesso gestore.

dsadsa

**ATTENDERE LA TERMINAZIONE DI UN THREAD: JOIN()** *31-10-24*

In alcuni casi, un thread può voler attendere la terminazione di un altro, ad esempio per aspettare   
che quest’ultimo finisca di svolgere delle operazioni utili al primo. A tale scopo si può usare   
il metodo **join()** della classe Thread, presente in due varianti:

* **public final void join() throws InterruptedException:** il thread che lo invoca si sospende fintantoché il thread rappresentato dall’oggetto thread sul quale il metodo è stato invocato   
  non termina;
* **public final void join(long timeout) throws InterruptedException:** come join()   
  senza argomenti, con la differenza che se dopo *timeout* millisecondi il thread rappresentato dall’oggetto thread sul quale il metodo è stato invocato non è ancora terminato,   
  il thread che ha invocato il metodo viene risvegliato.

*Es.*

Nel main, creo un oggetto thread *a* di tipo Analizzatore, e attivo successivamente il thread rappresentato da questo oggetto. Fatto ciò, invoco il metodo join() sull’oggetto thread *a*. Il main thread, dunque, si sospende fintantoché il thread rappresentato dall’oggetto thread *a* non termina, così che, in questo caso, questo thread finisca di svolgere delle operazioni utili al main thread. Una volta che il thread rappresentato dall’oggetto thread *a* è terminato, il main thread   
si risveglia e recupera il risultato delle operazioni svolte dal thread appena terminato.

dsadsa

public class Lancia {

public static void main(String[] args) {

int n = Integer.parseInt(args[0]);

Analizzatore a = new Analizzatore(n);

a.start();

try {

a.join();

boolean r = a.getRisultato();

System.out.println(r ? "primo" : "non primo");

} catch (InterruptedException ie) {

System.out.println(“sono stato interrotto”);

}

}

}

public class Analizzatore extends Thread {

private int n;

private boolean risultato;

public Analizzatore(int n) {

this.n = n;

}

public void run(){

risultato = primo(n);

}

public boolean getRisultato(){

return risultato;

}

private boolean primo(int n){

for(int i=2; i<n; i++) {

if(n%i == 0)

return false;

}

return true;

  }

}

**MANDARE UN INTERRUPT AD UN THREAD: INTERRUPT()**

In alcuni casi, un thread può voler mandare un interrupt ad un altro thread per comunicargli il verificarsi di un certo evento (tipicamente, per chiedergli di terminare). A tale scopo si può usare   
il metodo **interrupt()** della classe Thread, che manda un interrupt al thread rappresentato   
dall’oggetto thread su cui si invoca il metodo. In particolare, ad ogni thread è associato un flag   
chiamato interrupt status, e quando si invoca il metodo interrupt() su un certo oggetto thread:

* Se il thread rappresentato dall’oggetto thread è sospeso a seguito dell’invocazione   
  di sleep(), wait() o join(), il thread viene risvegliato e sleep(), wait() o join() lanceranno un’eccezione di tipo InterruptedException. L’interrupt status del thread, invece,   
  rimane invariato;
* Altrimenti, l’interrupt status del thread viene settato.

Nel primo caso, dunque, un thread si accorge di aver ricevuto un interrupt perché viene lanciata   
un eccezione di tipo InterruptedException. Nel secondo caso, invece, un thread deve controllare   
il proprio interrupt status per vedere se ha ricevuto o meno un interrupt, e questo può farlo   
attraverso i seguenti metodi offerti dalla classe Thread:

* **public static boolean interrupted():** restituisce true se l’interrupt status del thread   
  che lo invoca è settato, false altrimenti. Il metodo inoltre resetta tale interrupt status;
* **public boolean isInterrupted():** restituisce true se l’interrupt status del thread rappresentato dall’oggetto thread sul quale il metodo è stato invocato è settato, false altrimenti.   
  Il metodo non resetta tale interrupt status.

*Es.*

public class Creatore {

public static void main(String[] args){

Esecutore e = new Esecutore();

Nel main, creo un oggetto thread *e* di tipo Esecutore, che attivo successivamente. Fatto ciò, dopo un po’, invoco il metodo interrupt() sull’oggetto thread *e*,   
inviando così un interrupt al thread rappresentato da tale oggetto thread, chiedendogli in questo caso di terminare.

dsadsa

e.start();

try{

Thread.sleep(2000);

}

catch(InterruptedException ie) {

…

}

e.interrupt();

}

}

public class Esecutore extends Thread {

Ogni thread di tipo Esecutore esegue delle operazioni fintantoché interrupted() non restituisce true, e cioè fintantoché l’interrupt status del thread non è settato,   
a segnalare che tale thread ha ricevuto un interrupt.  
Ricevuto l’interrupt, il thread decide di terminare.

dsadsa

public void run(){

while(!interrupted()) {

fai();

}

}

private void fai(){

    System.out.print(".");

}

}

**CLASSI WRAPPER**

Per ogni tipo primitivo esiste una classe di sistema corrispondente, detta **classe wrapper**:

* Boolean, corrispondente al tipo primitivo boolean;
* Byte, corrispondente al tipo primitivo byte;
* Short, corrispondente al tipo primitivo short;
* Integer, corrispondente al tipo primitivo int;
* Long, corrispondente al tipo primitivo long;
* Float, corrispondente al tipo primitivo float;
* Double, corrispondente al tipo primitivo double;
* Character, corrispondente al tipo primitivo char.

Una classe wrapper serve a racchiudere in un oggetto un valore del tipo primitivo corrispondente.   
Per esempio, esistono alcune strutture dati in cui è possibile memorizzare solo oggetti, e non valori   
di tipo primitivo, per cui se si vuole memorizzare in essa un valore di tipo primitivo bisogna prima racchiuderlo in un oggetto della corrispondente classe wrapper.

E’ possibile creare un oggetto di una classe wrapper contenente un valore del tipo primitivo corrispondente alla classe, ottenendo un riferimento all’oggetto creato, in due modi:

* Attraverso l’operatore new, passando al costruttore della classe wrapper il valore   
  di tipo primitivo o una String che rappresenta tale valore;
* Attraverso il metodo static valueOf() della classe wrapper, passando come argomento   
  il valore di tipo primitivo o una String che rappresenta tale valore.

Ogni classe wrapper *Type*, inoltre, offre alcuni metodi e campi utili per lavorare sui valori   
del tipo primitivo *type* corrispondente:

* public *type* *type*Value(): restituisce il valore di tipo primitivo contenuto nell’oggetto wrapper   
  sul quale è stato invocato;
* public static final *type* MIN\_VALUE: contiene il minimo valore rappresentabile come un *type*;
* public static final *type* MAX\_VALUE: contiene il massimo valore rappresentabile come un *type*.

Ogni classe wrapper implementa l’interfaccia Comparable.

Per semplificare la creazione di un oggetto di una classe wrapper e l’estrazione del valore   
di tipo primitivo contenuto in un oggetto di una classe wrapper, nelle ultime versioni di Java   
sono stati introdotti i meccanismi dell’auto-boxing e dell’auto-unboxing:

* Il meccanismo dell’**auto-boxing** converte in maniera automatica un valore di tipo primitivo   
  in un oggetto della classe wrapper corrispondente contenente tale valore. Questo avviene quando:
  + Si inizializza/assegna un valore di tipo primitivo ad un riferimento di tipo classe wrapper corrispondente;
  + Si passa come argomento un valore di tipo primitivo ad un metodo che accetta   
    tale argomento come un oggetto della classe wrapper corrispondente;
* Il meccanismo dell’**auto-unboxing** converte in maniera automatica un oggetto di una classe wrapper nel valore del tipo primitivo corrispondente contenuto in tale oggetto. Questo avviene quando:
  + Si inizializza/assegna un oggetto di una classe wrapper ad una variabile del tipo primitivo corrispondente;
  + Si passa come argomento un oggetto di tipo classe wrapper ad un metodo che accetta tale argomento come un valore del tipo primitivo corrispondente.

*Es.*

Creo due oggetti *i1* e *i2* di tipo Integer, contenenti rispettivamente il valore di tipo int 40 e 50.

Inizializzo la variabile *x* di tipo int con il valore di tipo int contenuto nell’oggetto *i2*.

Creo un oggetto *j* di tipo Integer, contenente il valore di tipo int 23. Grazie al meccanismo dell’auto-boxing, questo posso farlo assegnando direttamente il valore di tipo int 23 al riferimento *j*;

Inizializzo la variabile *w* di tipo int con il valore di tipo int contenuto nell’oggetto *j* di tipo Integer. Grazie al meccanismo dell’auto-unboxing, questo posso farlo assegnando direttamente l’oggetto *j* alla variabile *w*;

dsadsa

Integer i1 = Integer.valueOf(40);

Integer i2 = Integer.valueOf(“50”);

int x = i2.intValue();

Integer j = 23;

int w = j;