



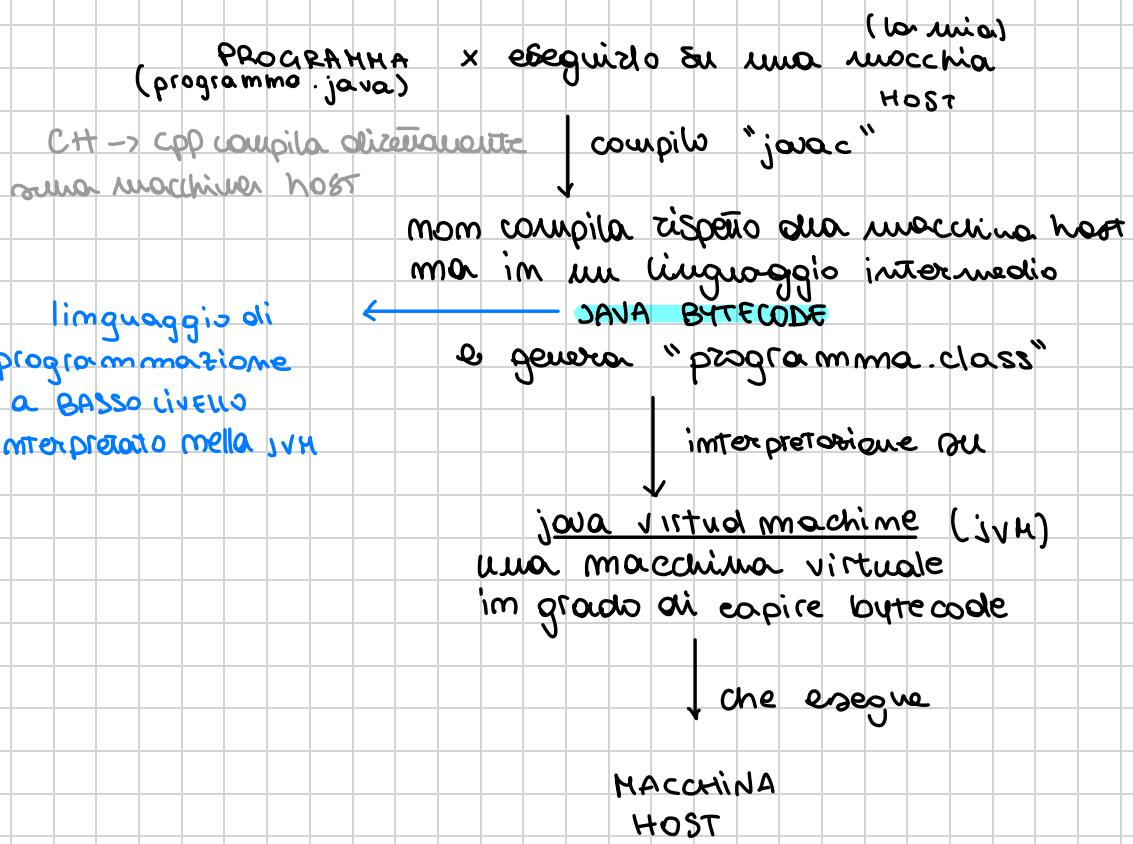
JAVA

- mosce come linguaggio completamente orientato agli oggetti
- sintassi simile a C++
 - facilitare il passaggio a java
 - semplificazioni:
 - no aritmetica dei puntatori
 - non abbiamo controllo sulla memoria (es. dinamica)
 - portabilità (indipendente dalla piattaforma)

(es. .exe in C++ funziona sul mio pc, ma non su quello di un altro)

"write once run everywhere"

- approccio misto → compilazione / interpretazione



Compilo da **java** a **java bytecode**
Interpreto da **java bytecode** su **JVM**
JVM esegue sulla **macchina host**

} portabilità

Java è implementato anche su android per lo sviluppo di app

il file program.class può essere eseguito su qualunque macchina fisica che ha installata una JVM.

• in java non esiste l'ambiente globale

in java tutto è una classe

• la prima funzione eseguita è la funzione main scritta.

"public static void main(String[] args)"

punto d'ingresso del programma

nome funzione
→ tipo di ritorno
(standard)

funzione può essere chiamata
anche dall'esterno
(public)

(standard) ←
array di stringhe chiamato
"args"

D.C. } dove vengono memorizzati eventuali
parametri passati in fase di
esecuzione del programma
(dall'esterno)

cout << → System.out.println (" ");
} a capo
} classe interna a java

System.out => ritorna lo Standard output

} e' un oggetto, con vari metodi

tra cui "println"
↓

stampo una stringa
e va a capo

• STRUTTURA DI PROGRAMMA

- non c'è previsto uno SPAZIO GLOBALE
- tutte le funzioni e i dati sono MESSOSSORTEMENTE interne a classi
- ogni classe è in un file separato

↳ il nome della classe deve essere uguale !
al nome del file

+ prima lettera deve essere maiuscola
(es. AgendaDiStudenti)

camelcase

ogni classe ha :

- metodi (methods)
- campi (fields)

- punto d'ingresso: public static void main (String[] args)
-----.

Scommer = classe data nel core di java

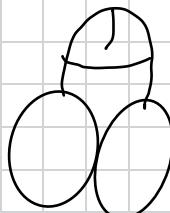
Scommer sc = new Scommer (System.in)

(dinamico),
↳ oggetto di
tipi scanner → collega a uno stream
di input
↓
cim >

X = sc.nextInt(),

↳ metodo della classe Scanner
per leggere un intero

C++: include <maine library>
java: import maine library;

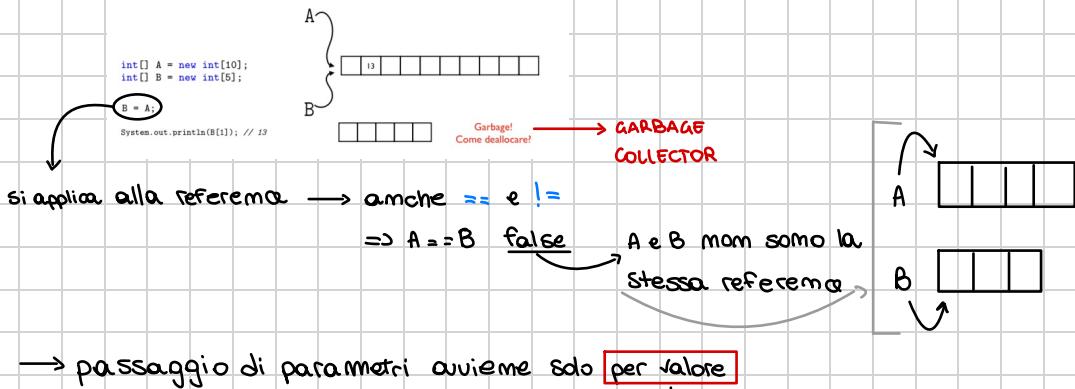


ARRAY → tipi di reference predefiniti
 → `int[] A = new int[10]` ⇒ oggetto di tipo
`array (10 int)`, forma una
 REFERENCE all'oggetto
`= new int[m]`

m → può essere specificata a run-time

`int[] A = {1, 5, 7, 1, 2, 6}`
`int[] A = new int[] {1, 5, 7, 1, 2, 6}` → zucchero sintattico

→ se l'indice dell'array esce dai limiti della dimensione, viene sollevata un'eccezione a RUN-TIME.
 → hanno un attributo `length` → `A.length` // torna la dimensione dell'array
 → sono aree contigue di memoria (come in C++)



```
public class Main {
  public static void f(int[] V) {
    V[0] = 7;
    return;
  }

  public static void main(String[] args) {
    int[] A = {1, 2, 3};
    f(A);
    System.out.println(A[0]);
    return;
  }
}
```

- A è di tipo reference
- A e V sono reference allo stesso oggetto
- A e V condividono lo stesso array

↓
 Se il parametro passato è una reference viene realizzato un passaggio di parametri per riferimento

Tipi PRIMITIVI: sono passati per valore

Tipi REFERENCE: sono passati per riferimento

GARBAGE COLLECTOR

- un oggetto viene automaticamente eliminato quando non è REFERENZIATO da nessuna REFERENCE
- programma interno alla JVM

pro

- AUTOMATICO: mom a carico del programmatore
- è impossibile la presenza di garbage

cons

- il programmatore mom ha controllo sulla memoria
- l'attività del GB può rallentare l'esecuzione del programma



TIPI PRIMITIVI

- int
- char
- float e double
- boolean



valori e possibili operazioni sono gli stessi previsti da C++

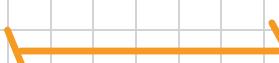
DIFFERENZA (java|C++): l'insieme dei valori di ciascun tipo è completamente specificato dal linguaggio (machine-independent) e mom dalla specifica implementazione (in C++: machine-dependent)

- BOOLEAN: mom dà come risultato 0/1 ma true/false
- java converte implicitamente valori di un tipo più grande (mom viceversa)

```
float x = 1.7;  
double y = 1.5;  
y = x; ✓  
x = y; ✗ casting esplicito (con perdita di informazione) → x = (float) y;
```

- mom esiste spazio globale
- dati e funzioni necessariamente interne a classi
- ogni classe in un file separato → nome classe = nome file
 - + prima lettera maiuscola
- tutte le funzioni sono metodi + overloading di metodi **SI**
di operatori **NO**
- la visibilità dei campi e dei metodi deve essere specificata per tutti i campi

- gli oggetti vengono sempre istanziati tramite l'operatore `new`, che va a creare una referenza all'oggetto appena creato inizializzato a `NULL`.

Java	C++
<pre>C1 x; x = new C1();</pre> <p>x è una reference ad oggetti di tipo C1</p> <p>E' possibile creare oggetti esclusivamente in modo dinamico</p>	<pre>C1* x; x = new C1;</pre> <p>x è un puntatore ad oggetti di tipo C1</p> <p>E' possibile creare oggetti in modo statico e dinamico</p>
	

STRINGHE

- Stringhe sono istanze della classe `String`

```
String s = new String()
char elem[] = {'h', 'e', 'l', 'l', 'o'};
String s = new String(elem);
```

→ oltre due costruttori:

- senza parametri, creazione stringa vuota
- con un parametro di tipo array di caratteri

- le costanti stringa sono oggetti della classe `String`

`String s = "hello";`
+ Non sono realizzate come array di caratteri terminati da un carattere speciale

- tra i costruttori della classe `String` troviamo anche il costruttore di copia

`String r = new String(s)` → crea una copia della stringa s
`String t = new String("abc")` → crea una copia della stringa "abc"

- op. primitivo messo a disposizione è `+`, per la concatenazione

`String s = "hello";
String r = s + "world" + "!";`

- le altre op. sono realizzate come metodi della classe `String`.

→ `s.length();` ritorna la lunghezza della stringa s

→ `s.charAt(i);` accesso all'i-esimo carattere di una stringa, ritorna il carattere in posizione i.

Se i è fuori dai limiti viene lanciata un'eccezione

! le stringhe in Java sono oggetti IMMUTABILI ! s.charAt(0) = 'm' NO.

- l'assegnamento (`s = s`) e' tra REFERENCE, non tra oggetti.

Anche l'operatore `==`, per confrontare due oggetti possiamo utilizzare il metodo equals. `s.equals(r);` → return di tipo bool

→ tutte le classi in java hanno un metodo equals.

```
String s = "hello"  
String r = "hello"  
if (s == r) // true
```



ottimizzazione della JVM, s e r contengono la stessa stringa perciò punteranno alla stessa cella di memoria.

cotidiana

- `indexOf` → `s.indexOf(c)` ritorna

0 $s.length() - 1$

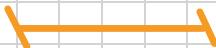
l'indice della prima occorrenza
di c in s

-1 se c non c'e' in s

- `substring` → `s.substring(init, end)` → tocca la sottostringa di s da init a end -1

Se gli indici sono fuori dai limiti
della stringa viene lanciata
un'eccezione

Se end e' omesso tocca la sottostringa
che va da init a `s.length() - 1`



CLASSI WRAPPER → oggetti che contengono i valori primitivi corrispondenti

Byte	Float
Short	Double
Integer	Character
Long	Boolean

Integer x = 3; viene automaticamente tradotto
dal compilatore come:

`Integer x = new Integer(3);`



ma e' una conversione di tipo: AUTOBOXING

si tratta di una sintassi semplificata per nascondere la creazione
di oggetti wrapper

- `x.intValue()`; per recuperare il valore primitivo corrispondente



UNBOXING

`Integer x = new Integer(7)`

`int y = x.intValue() + 2; // 9`

Da java 5.0 le op. di **UNBOXING** e **AUTOBOXING** sono eseguite in automatico:

Integer x = new Integer(7);

int y = x+2 //9

contenitori dei
valori primitivi
corrispondenti

- gli oggetti wrapper sono **IMMUTABILI** e non esiste un modo per modificarli

int j=3

Integer i=j;

i=7;

non modifica il valore dell'oggetto che contiene 3,
ma ne crea uno nuovo.

- "==" confronto fra riferimenti, per confrontare i valori contenuti usiamo **equals**
x.equals(y)

- **toString** → x.toString() ritorna una stringa corrispondente al valore
contenuto in x

presente in tutte le classi if (x.toString().equals("3")) ...

a volte viene invocato automaticamente

- **parseInt** → Integer.parseInt(s) analizza la stringa s e, se possibile, ritorna
il numero intero corrispondente, altrimenti
lancia un'eccezione

int i = Integer.parseInt("13") + 3; // i = 16
int j = Integer.parseInt("13"); // eccezione
int k = Integer.parseInt("2.1"); // eccezione

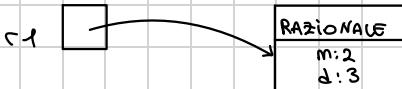
Elementi di istanza e di classe

- elementi di classe

una classe è composta da: • ATTRIBUTI/CAMPPI private int num;

• METODI Raziionale somma(Raziionale other){...}

Ogni **istanza** di una classe (**un oggetto**) ha una propria "copia" degli **attributi**
+ sull'oggetto possono essere "applicate" delle azioni invocando i **metodi**.

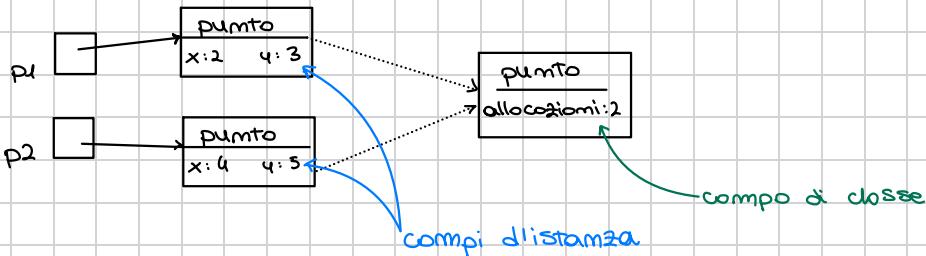


attributi

oggetto

- **CAMPO DI Istanza**: ogni Istanza contiene una copia dei **campi d'istanza**
- **METODI D'ISTANZA**: metodi applicabili ad un'istanza della classe
- **CAMPPI DI CLASSE**: a differenza dei campi di classe, sono condivisi fra tutte le istanze di classe

oggetti



- gli attributi/campi di una classe possono essere:
 - CAMPI D'ISTANZA
 - CAMPI DI CLASSE

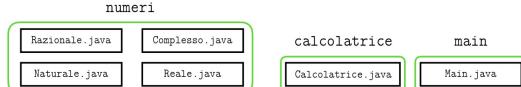
(ognuno con la propria visibilità)

• **METODI DI CLASSE**: si applicano alla classe (non ad un oggetto come i met. d'istanza)

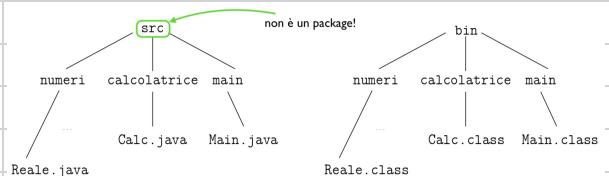
- quando invocato, non ha un riferimento a this.
- non è possibile accedere ai campi d'istanza
- il METODO MAIN:
 - punto d'ingresso di un programma
 - viene chiamato dall'ambiente esterno al programma
 - non ha bisogno di creare un oggetto della classe che lo contiene

PACKAGE

- le classi logicamente correlate possono essere raggruppate in un PACKAGE
- × specificare che una classe è parte di un package la prima riga di codice deve essere: `package <nome>;`



- questo package definisce il proprio spazio di nomi (es. Naturale)
- una classe di un package può essere utilizzata:
 - 1) numerici.Complesso cl package close
 - 2) import numerici.Complesso package close
 - 3) import numerici.*; → importa tutte le classi del package



• package Standard

Package	Classe
non richiede nessun import	java.lang String Integer, Boolean, ... System
java.io	InputStream OutputStream IOException ...
java.util	Scanner Vector ...

• visibilità delle classi

- public: la classe è visibile ovunque, anche all'esterno del package di appartenenza
- Default (visibilità di package): la classe è visibile solamente all'interno del package in cui è contenuta
- private: classi private, possono essere dichiarate esclusivamente come classi INTERNE o INNESTATE (una classe privata C non può essere creata nel file C.java, ma esclusivamente dentro un'altra classe)

• visibilità dei campi

- public: il campo è visibile ovunque
- private: il campo è visibile nella classe in cui è dichiarato
- default (visibilità di package): il campo è visibile nella classe in cui è dichiarato e nelle classi dello stesso package a cui la classe appartiene

ES.1 16/06/23

```
template <class T>
class Stack {
private:
    T* stack;
    int dim;
    int top;
    void enlarge() {
        T* newStack = new T[dim*2];
        for (int i=0; i<dim; i++)
            newStack[i] = this->stack[i];
        this->dim *= 2;
        delete [] this->stack;
        this->stack = newStack;
    }
public:
    Stack() {
        this->dim = 10;
        this->stack = new T[dim];
        this->top = 0;
    }
    ~Stack() {
        delete [] this->stack;
    }
    Stack(const Stack& other) {
        this->dim = other.dim;
        this->top = other.top;
        this->stack = new T[other.dim];
        for (int i=0; i<other.top; i++)
            this->stack[i] = other.stack[i];
    }
}
```

```
Stack& operator=(const Stack& other) {
    if (this != &other) {
        delete [] this->stack;
        this->dim = other.dim;
        this->top = other.top;
        this->stack = new T[dim];
        for (int i=0; i<top; i++)
            this->stack[i] = other.stack[i];
    }
    return *this;
}

void push(T elem) {
    if (dim == top)
        enlarge();
    this->stack[top++] = elem;
}

T pop() {
    if (isEmpty())
        throw runtime_error("pila vacia");
    else
        top--;
    return this->stack[top];
}

bool isEmpty() const {
    return top == 0;
}

int size() const {
    return top;
}
```

```

public interface Immatricolabile {
    public String getMatricola();
}

public class StudenteOrdinario implements Immatricolabile {
    private String matricola;
    private String scuolaSuperiore;
    public StudenteOrdinario(String mat, String scuola) {
        this.matricola = matricola;
        this.scuolaSuperiore = scuola;
    }
    public String getMatricola() {
        return this.matricola;
    }
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        else if (obj == null)
            return false;
        else if (getClass() != obj.getClass())
            return false;
        StudenteOrdinario other = (StudenteOrdinario) obj;
        return this.matricola.equals(other.matricola);
    }
}

public class StudenteLavoratore implements Immatricolabile {
    private String matricola;
    private String azienda;
    public StudenteLavoratore(String mat, String azienda) {
        this.matricola = mat;
        this.azienda = azienda;
    }
    public String getMatricola() {
        return this.matricola;
    }
}

```

```

public boolean equals (Object obj) {
    if (this == obj)
        return true;
    else if (obj == null)
        return false;
    else if (getClass() != obj.getClass())
        return false;
    Studentelavoratore other = (Studentelavoratore) obj;
    return this.matricola.equals(other.matricola)
        && this.azienda.equals(other.azienda);
}
}

public class AppelloEsame implements Comparable<AppelloEsame> {
    private String nome;
    private String anno;
    private Set<StudentiOrdinari> iscrittiOrdinari;
    private Set<Studentelavoratori> iscrittiLavoratori;
    public AppelloEsame (String nome, String anno) {
        this.nome = nome;
        this.anno = anno;
        this.iscrittiOrdinari = new HashSet<>();
        this.iscrittiLavoratori = new HashSet<>();
    }
    Immatricolabile
    public void iscrivি (Object studente) throws AppelloEsameException {
        if (iscrittiOrdinari.size() == 50 && iscrittiLavoratori.size() == 50)
            throw new AppelloEsameException("max iscrizioni raggiunto");
        if (studente.getClass() == StudenteOrdinario) → studente instance of Stud.Ord.
            if (iscrittiOrdinari.size() == 50)
                throw new AppelloEsameException("max iscrizione studenti ordinari raggiunta");
            else if (iscrittiOrdinari.contains(studente))
                throw new AppelloEsameException("studente già iscritto");
            iscrittiOrdinari.add ((StudenteOrdinario) studente);
        }
        if (studente.getClass() == Studentelavoratore)
            if (iscrittiLavoratori.size() == 50)
                throw new AppelloEsameException("max studenti lavoratori iscritti raggiunto");
    }
}

```

```
else if ( iscrittilavoratori. contains(studente) )
    throw new AppelloEsameException ("studente già iscritto");
    iscrittilavoratori. add ((Studentelavoratore) studente);
}

public boolean equals ( Object obj) {
    if (this == obj)
        return true;
    else if ( obj == null )
        return false;
    else if ( getClass() != obj. getClass())
        return false;
    AppelloEsame other = (AppelloEsame) obj;
    return this. nome. equals( other. nome) &&
        this. ormai. equals( other. ormai) &&
        this. iscrittiOrdinari. equals( other. iscrittiOrdinari) &&
        this. iscrittilavoratori. equals( other. iscrittilavoratori);
}

public int compareTo (AppelloEsame other) {
    return Integer. compare (this. iscrittilavoratori. size(), other. iscrittilavoratori. size());
}
```

```
public class AppelloEsameException extends Exception {
    public AppelloEsameException (String msg) {
        super (msg);
    }
}
```



```

template < class T >
class Queue {
private:
    struct node {
        T element;
        node* next;
    };
    node* first;
    node* last;
public:
    Queue() {
        this->first = nullptr;
        this->last = nullptr;
    }
    ~Queue() {
        while (!isEmpty())
            dequeue();
    }
    Queue(const Queue & other) {
        this->first = other.first;
        this->last = other.last;
        node* cursor = first;
        while (cursor != nullptr) {
            enqueue(cursor->data);
            cursor = cursor->next;
        }
    }
    Queue& operator=(const Queue & other) {
        if (this != & other) {
            while (!isEmpty())
                dequeue();
            node* cursor = other.first;
            while (cursor != nullptr) {
                enqueue(cursor->data);
                cursor = cursor->next;
            }
        }
        return *this;
    }
};
```

CURSOR = CURSOR->NEXT;

```

void enqueue(T elem) { → enqueue lavora sul last
    node* newNode = new node;
    newNode → element = elem } creazione di un nuovo nodo
    newNode → next = nullptr; che sarà l'ultimo della coda
    if (isEmpty()) {
        first = newNode; } Se la Coda è vuota first e last coincidono
        last = first; }

last → next = newNode; } altrimenti aggiorna last su nuovo nodo
last = newNode; }

T dequeue() { → lavora sul first
    if (isEmpty())
        throw runtime_error("coda vuota");
    node* tmp = first; → Solvo il nodo da rimuovere
    T result = first → data; → Solviamo il dato del primo nodo, prima di eliminarlo
    first = first → next; → punta al nodo successivo al first, aggiornando first
    if (first == nullptr) { } se dopo la rimozione del primo nodo la coda è vuota
        last = nullptr; viene impostato anche last su nullptr
    delete tmp;
    return result;
}

bool isEmpty()
{
    return first == nullptr;
}

```

```

template < class T >
class Queue {
private:
    struct node {
        T elem;
        node* next;
    };
    node* first;
    node* last;
public:
    Queue() {
        this->first = nullptr;
        this->last = nullptr;
    }
    ~Queue() {
        while (!isEmpty())
            dequeue();
    }
    Queue(const Queue & other) {
        this->first = nullptr;
        this->last = nullptr;
        node* cursor = other.first;
        while (cursor != nullptr)
            enqueue();
        cursor = cursor->next;
    }
}

```

```

Queue & operator=(const Queue & other) {
    if (this != &other) {
        node* cursor = first;
        if (isEmpty())
            while (cursor != nullptr)
                dequeue();
        cursor = cursor->next;
        mode* cursor = other.first;
        while (cursor != nullptr)
            enqueue();
        cursor = cursor->next;
    }
}

```

→ element

```

enqueue(node* cursor) {
    cursor->next = cursor->next->next;
    return *this;
}

void enqueue(T elem) {
    node* newNode = new node;
    newNode->element = elem;
    newNode->next = nullptr;
    if (isEmpty()) {
        last = newNode;
        first = newNode;
    } else {
        last->next = newNode;
        last = newNode;
    }
}

T dequeue() {
    if (isEmpty())
        throw runtime_error("coda vuota");
    node* tmp = first;
    T result = first->element;
    first = first->next;
    if (first == nullptr)
        last = nullptr;
    delete tmp;
    return result;
}

bool isEmpty() const {
    return first == nullptr;
}

void print(ostream & fout) const {
    fout << "{ ";
    node* cursor = first;
    while (cursor != nullptr) {
        fout << cursor->element;
    }
}

```

```
if (cursor->next != nullptr)
    fout << ", ";
* } // cursor = cursor->next;
    fout << " }";
}

};

template < class T >
ostream& operator<< (ostream& fout, const Queue<T>& coda) {
    coda.print(fout);
    return fout;
}
```

ES.2 17/07/23

```
public abstract class Impiegato {
    private String nome;
    private String cognome;
    private int oreAmmuali;
    public Impiegato (String nome, String cognome, int ore) {
        this.nome = nome;
        this.cognome = cognome;
        this.oreAmmuali = ore;
    }
    public String getName () {
        return nome;
    }
    public String getCognome () {
        return cognome;
    }
    public int getOre () {
        return oreAmmuali;
    }
    public boolean equals (Object obj) {
        if (this == obj)
            return true;
        else if (obj == null)
            return false;
        else if (getClass() != obj.getClass())
            return false;
        Impiegato other = (Impiegato) obj;
        return this.nome.equals (other.nome)
            && this.cognome.equals (other.cognome)
            && this.oreAmmuali == other.oreAmmuali;
    }
    public abstract double getStipendioAmmuale ();
}
```

```

public class ImpiegatoBase extends Impiegato {
    private static final double importoOrario = 15.50;
    public ImpiegatoBase (String nome, String cognome, int ore) {
        super (nome, cognome, ore);
    }
    public double getImportoAmmMese () {
        return ore * importoOrario;
    }
    public boolean equals (Object obj) {
        return super.equals (obj);
    }
}

public class Manager extends Impiegato implements Comparable < Manager >, Iterator < Impiegato > {
    private static final double = 25.90;
    private Set < ImpiegatoBase > team;
    public Manager (String nome, String cognome, int ore) {
        super (nome, cognome, ore);
        this.team = new HashSet < () >;
    }
    public boolean equals (Object obj) {
        if (this == obj)
            return true;
        else if (obj == null)
            return false;
        else if (getClass () != obj.getClass ())
            return false;
        Manager other = (Manager) obj;
        return nome.equals (other.nome) &&
               cognome.equals (other.cognome) &&
               team.equals (other.team) &&
               oreAmmMese == other.oreAmmMese;
    }
    public void aggiungiAlTeam (Impiegato impiegato) throws TeamManagerException {
        if (impiegato instanceof Manager)
            throw new TeamManagerException ("Non e' possibile aggiungere un manager al team");
        else if (team.contains (impiegato))
    }
}

```

Bases }

```
        throw new TeamManagerException("Impiegato già presente nel team");
    } else if (impiegato.getOre < 20)
        throw new TeamManagerException("Impiegato ha poche ore di lavoro");
    team.add((ImpiegatoBase) impiegato);
}

public int compareTo (Manager other){
    return Integer.compare(team.size(), other.team.size());
}

public Iterator<ImpiegatoBase> iterator(){
    return team.iterator();
}

public class TeamManagerException extends Exception {
    public TeamManagerException (String msg){
        super(msg);
    }
}
```

ES. 1 12/10/124

```
template < class T >
class Set {
private:
    T* container;
    int dim;
    int top;
    void enlarge() {
        T* tmp = new T[dim * 2];
        for (int i=0; i < dim; i++)
            tmp[i] = container[i];
        delete [] this->container;
        this->container = tmp;
        this->dim *= 2;
    }
public:
    Set() {
        this->dim = 10;
        this->container = new T[dim];
        this->top = 0;
    }
    ~Set() {
        delete [] this->container;
    }
    Set (const Set& other) {
        this->dim = other.dim;
        this->container = new T[dim];
        this->top = other.top;
        for (int i=0; i < top; i++)
            container[i] = other.container[i];
    }
    Set& operator=(const Set& other) {
        if (this != &other) {
            delete [] this->container;
            this->dim = other.dim;
            this->top = other.top;
        }
    }
}
```

```

this -> container = new T[dim];
for (int i=0; i < top; i++)
    container[i] = other.container[i];
}

return *this;
}

const T& elem
void add (T elem) {
if (!contains(elem))
    if (top == dim)
        enlarge();
    this->container[top++] = elem;
} else
    throw runtime_error ("elemento già presente nel Set");
}

const T& elem
bool contains (T elem) const {
for (int i=0; i < top; i++)
    if (container[i] == elem)
        return true;
return false;
}

int getSize () const {
return top;
}

Set operator- (const Set& other)
const

Set result,
int count = 0;
int newDim = (*this->top > other.top ? *this->top : other.top);
for (int i=0; i < newDim; i++) {
    if (container[i] == other.container[i])
        result.add(container[i]);
    count++;
}
result.top = count;
return result;
}

```

```

for (int i=0; i < top; i++) {
    if (other.contains(container[i]))
        result.add(container[i]);
}
}

```

```
Void print( ostream& fout ) const {
    fout << "{ ";
    for( int i=0; i<top; i++ ){
        fout << container[i];
        if( i!=top-1 )
            fout << ", ";
    }
    fout << " } " << endl;
}
};

template < class T >
ostream& operator << ( ostream& fout, const Set<T>& set ) {
    set.print(fout);
    return fout;
}
```