

Lezione_09_DeA

3. Ripasso di Java

Per scrivere un programma stand-alone, chiamato MyProgram.java è necessario che la public class abbia lo stesso nome del file e che abbia il metodo main, dal quale inizia l'esecuzione.

```
import ... ; // import a package or a class
public class MyProgram {
    public static void main(String[] args) {
        /*...*/ // corpo del metodo main
    }
    /* eventuali altri metodi */
}
```

Per compilare il programma si usa il comando `javac MyProgram.java` da terminale, che crea un *bytecode* `MyProgram.class` eseguibile sulla *Java Virtual Machine* (JVM). Per eseguirlo si usa il comando `java MyProgram`, che richiede che la directory corrente sia nel `CLASSPATH`, altrimenti si usa `java -cp - MyProgram`.

In alternativa, si può usare in *Integrated Development Environment* (IDE), come IntelliJ IDEA, Eclipse, Jbuilder.

3.1. Caratteristiche di Java e dell'approccio Object-Oriented

Le caratteristiche principali sono la modularità, l'astrazione e encapsulamento (information hiding) e l'ereditarietà (Inheritance).

3.1.1. Modularità

Le classi vengono viste come un insieme di oggetti che interagiscono insieme; Main è la classe principale, mentre le altre rappresentano tipi di oggetti. Un'applicazione è un insieme di *oggetti interagenti*.

Un *oggetto* è un'*istanza di una classe* che definisce variabili di istanza (in inglese "instance variables" o "fields") e metodi ("methods").

Per definire una variabile `i` di tipo `Integer`, alla quale assegno un oggetto di tipo `Integer`, dopo averlo creato, scrivo: `Integer i = new Integer(5);`.

3.1.2. Astrazione e encapsulamento (information hiding)

Con l'*information hiding* si astrae la *specificità* delle funzionalità, separandola dalla loro *implementazione*, quindi si possono usare le funzionalità solamente in base alla specifica.

Visto che l'implementazione delle funzionalità è nascosta, gli errori sono più confinati (si ha *robustezza*) e c'è la possibilità di cambiare l'implementazione senza cambiare la specifica (si ha *adattabilità*).

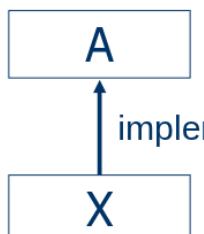
Nel caso delle strutture dati, questo approccio dà origine alla nozione di *Abstract Data Type* (ADT), che in Java si realizza tramite l'utilizzo di interfacce, classi e classi astratte.

Un'*interfaccia* è un insieme di dichiarazioni di metodi senza corpo e di costanti.

Una *classe* è la definizione di costanti, variabili e metodi con corpo.

Una *classe astratta* è una via di mezzo tra un'*interfaccia* e una classe (ci sono alcuni metodi senza corpo).

```
public interface A {  
    public int a1();  
    public boolean a2();  
}  
  
public class X implements A {  
    public int a1() { /*...*/ }  
    public boolean a2() { /*...*/ }  
    public void b() { /*...*/ }  
}  
  
A var1 = new A(); //NO!  
A var1 = new X(); //OK!
```



Ad esempio: `java.lang.String implements java.lang.Comparable`.

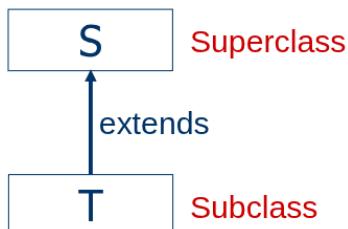
⚠ Attenzione

Non si può creare un oggetto di un'interfaccia, sono legati solo alle classi.

3.1.3. Ereditarietà (inheritance)

Con l'ereditarietà si *importano* in una classe le caratteristiche di un'altra classe, aggiungendone di nuove e/o specializzandone alcune. In Java una *sottoclasse estende una superclasse*

```
public class S {  
    public void a() { /*...*/ }  
    public void b() { /*...*/ }  
    public int c() { /*...*/ }  
}  
  
public class T extends S {  
    /* inherits b() and c() */  
    float y; // added  
    public void a() { /*...*/ } // specialized  
    public boolean d() { /*...*/ } // added  
}
```



① Osservazione

Ogni classe estende implicitamente `java.lang.Object`.

3.1.3.1. Ereditarietà multipla per interfacce

Un'Interfaccia (*non* una classe) può estender più interfacce.

```
public interface A { /*...*/ }  
public interface B { /*...*/ }  
public interface C extends A, B { /*...*/ }
```

① Osservazione

A e B non possono contenere metodi con la stessa firma.

```
public class D implements A, B {  
    // deve implementare tutti i metodi di A e B  
}  
  
public class D implements C {  
    // deve implementare tutti i metodi di A e B  
    // e quelli (eventuali) aggiuntivi di C  
}
```

3.2. Programmazione generica (generics)

La programmazione generica è un tipo di polimorfismo, introdotto da Java SE5. Permette l'uso di *variabili di tipo* nella definizione di classi, interfacce e metodi. Si parla in questo caso di *classi/interfacce generiche* e *metodi generici*.

Nel creare un'istanza di una classe generica si specifica il tipo (*actual type*) da sostituire con la variabile di tipo. L'*actual type* non può essere un tipo primitivo (come per esempio un *intero*), ma deve essere una classe che estende `Object`.

Nell'invocazione di un metodo generico, la sostituzione della variabile di tipo con un *actual type* è fatta automaticamente in base ai parametri passati.

L'uso delle classi/interfacce generiche evita il ricorso a parametri di tipo `Object` e l'uso frequente di cast.

```
public interface MyInterface<E> {  
    public int size();  
    public boolean method1 (E var1);  
    public E method2 ();  
}  
  
public class MyClass<E> implements MyInterface<E>{  
    E var;  
    public int size() { /*...*/ };  
    public boolean method1 (E var1) { /*...*/ };  
    public E method2 () { /*...*/ };  
    public E method3 (float var2) { /*...*/ };  
}  
  
MyInterface<Integer> x = new MyClass<Integer>();
```

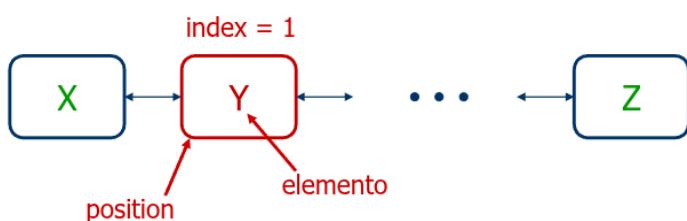
```
public class MyClass1<E extends S> { /*...*/ } // E può essere sostituito  
solo da oggetti di tipo che estende/implementa la classe/interfaccia S
```

3.3. ADT elementari

3.3.1. Lista (list)

Una *lista* (*list*) è una collezione di elementi organizzati secondo un ordine lineare tale per cui è identificabile il primo elemento, il secondo e così via fino all'ultimo.

- In una lista *index-based* un elemento può essere acceduto tramite un *indice intero* che indica il numero di elementi che lo precedono;
- In una lista *position-based* ogni elemento è contenuto in un contenitore detto *nodo* (*position*).



3.3.1.1. Lista index-based

```
public interface List<E> {  
    /** Returns the number of elements in this list. */  
    public int size();  
    /** Returns whether the list is empty. */  
    public boolean isEmpty();  
    /** Inserts an element e to be at index i, shifting all elements  
    after this. */  
    public void add(int i, E e);  
    /** Returns the element at index i, without removing it. */  
    public E get(int i);  
    /** Replaces the element at index i with e, returning the previous  
    element at i. */  
    public E set(int i, E e);  
    /** Removes and returns the element at index i shifting left  
    subsequent elements. */  
    public E remove(int i)  
}
```

! Osservazione

I metodi `public int size()` e `public boolean isEmpty()` si troveranno in ogni ADT che vedremo.

Si implementa questo tipo di lista *tramite array*. I metodi possono essere implementati con complessità $O(1)$, tranne `add` e `remove`, che richiedono complessità $O(n)$ (dove n è il numero di elementi nella lista).

⚠️ Osservazione

Nel caso in cui l'array sia pieno, l'inserimento di un elemento x è implementato creando un nuovo array di capacità maggiore (solitamente doppia), trasferendo tutte le entry dal vecchio al nuovo array, e inserendo l'elemento x nel nuovo array.

3.3.1.2. Lista position-based

```
public interface Position<E> {
    /** Return the element stored at this position */
    public E getElement();
}

public interface PositionalList<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Returns the first node in the list. */
    public Position<E> first();
    /** Returns the last node in the list. */
    public Position<E> last();
    /** Returns the node after a given node in the list. */
    public Position<E> after(Position<E> p);
    /** Returns the node before a given node in the list. */
    public Position<E> before(Position<E> p);
    /** Inserts an element at the front of the list. */
    public void addFirst(E e);
    /** Inserts an element at the back of the list. */
    public void addLast(E e);
    /** Inserts an element after the given node in the list. */
    public void addAfter(Position<E> p, E e);
```

```

    /** Inserts an element before the given node in the list. */
    public void addBefore(Position<E> p, E e);
    /** Removes a node from the list, returning the element stored there.
     */
    public E remove(Position<E> p);
    /** Replaces the element stored at the given node, returning old
     * element. */
    public E set(Position<E> p, E e);
}

```

Si implementa questo tipo di lista tramite *doubly-linked* list. Ogni nodo diventa un oggetto a sé, con variabili di istanza che "puntano" al predecessore e al successore. L'inizio e la fine della lista sono delimitati da dei *nodi sentinella*. I metodi possono essere implementati con complessità $O(1)$, ma la lista può essere scandita solo sequenzialmente,

① Osservazione

È possibile implementare una lista index-based tramite souble-linked list, o una lista position-based tramite array, ma con scarsi vantaggi.

3.3.2. Pila (stack) e coda (queue)

Una *pila (stack)* è una collezione di elementi inseriti e rimossi in base al principio *Last-In First-Out* (LIFO).

```

public interface Stack<E> {
    /** Returns the number of elements in this list. */
    public int size();
    /** Returns whether the list is empty. */
    public boolean isEmpty();
    /** Inserts an element e at the top of the stack. */
    public void push(E e);
    /** Returns the element at the top of the stack without removing it.
     */
    public E top();
    /** Removes and returns the element at the top of the stack. */
    public E pop();
}

```

Una *coda* (*queue*) è una collezione di elementi inseriti e rimossi in base al principio *First-In First-Out* (FIFO).

```
public interface Queue<E> {  
    /** Returns the number of elements in this list. */  
    public int size();  
    /** Returns whether the list is empty. */  
    public boolean isEmpty();  
    /** Inserts an element e at the rear of the queue. */  
    public void enqueue(E e);  
    /** Returns but does not remove the first element of the queue (null  
     * if empty). */  
    public E first();  
    /** Removes and returns the first element of the queue (null if  
     * empty). */  
    public E dequeue();  
}
```

Entrambe le strutture dati possono essere implementate tramite *doubly-linked list*.

La pila effettua inserimento (*push*) e rimozione (*pop*) in testa alla lista, mentre la coda effettua l'inserimento (*enqueue*) in coda e la rimozione (*dequeue*) in testa alla lista.

Tutti i metodi possono essere implementati con complessità $O(1)$ ma, a differenza della lista, non si ha accesso a element intermedi a meno di no rimuovere quelli che li precedono nell'ordine di accesso.

! Osservazione

È possibile usare anche una *singly-linked list*.

3.4. Collection framework di Java

"*Collection*" è un termine generico per indicare una struttura dati.

Il *collection framework* di Java è un'architettura unificata di strutture dati e algoritmi implementato nel package `java.util`.

Esso contiene:

- *interfacce* di varie collection;
- *classi* che implementano le interfacce;

- *algoritmi polimorfi* per operare su collection (*metodi statici della classe Collections*), come ad esempio il sorting;
- ampio uso della *programmazione generica*.

Il package `java.util` contiene diverse implementazioni di Liste, Pile e Coda. Tra esse si segnalano le seguenti:

- Lista
 - Interfaccia: `List`;
 - Classe `ArrayList`: implementazione index-based;
 - Classe `LinkedList`: implementazione sia index-based che position-based. Tuttavia l'implementazione position-based non utilizza esplicitamente il concetto di position ma si basa sull'uso di iteratori.
- Pila e Coda
 - Interfaccia unificata: `Deque` (Double-ended queue);
 - Classe `LinkedList`: implementazione unificata di Pila e Coda (e Lista);
 - Classe `Stack`: implementazione della Pila basata su `Vector`. Si consiglia tuttavia l'uso di classi, come `LinkedList`, che implementano l'interfaccia `Deque` che è più completa.

Info

I nomi di alcuni metodi differiscono da quelli riportati precedentemente che seguono il libro di testo.

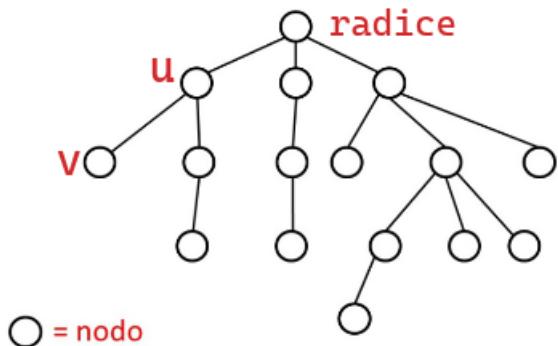
Riepilogo sul ripasso di Java

- Programma stand-alone;
- Caratteristiche di Java e dell'approccio Object-Oriented;
- ADT elementari:
 - Lista (index-based e position-based);
 - Pila
 - Coda
- Collection framework di Java, con particolare attenzione alle interfacce e classi che implementano Lista, Pila e Coda.

4. Alberi

4.1. Alberi generali

Un *albero* è una collezione di *nodi* caratterizzata da una *struttura gerarchica* che si dipana da un nodo *radice* tramite relazioni di tipo padre-figlio.



Nel disegno *u* è padre di *v*, *v* è figlio di *u*.

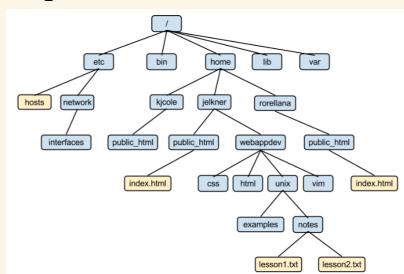
! Osservazioni

Le relazioni padre-figlio costituiscono un insieme di collegamenti minimi che introducono un legame (connessione) tra tutti i nodi.

Una lista è un caso estremo di albero con una struttura gerarchica lineare.

💡 Campi applicativi

1. Strutture dati: mappe, priority queue;
2. Esplorazione risorse: filesystem, siti di e-commerce;



3. Sistemi distribuiti e reti di comunicazione:
sincronizzazione, broadcast, gathering;
4. Analisi di algoritmi: albero della ricorsione;
5. Classificazione: alberi di decisione;
6. Compressione di dati (codici di Huffman);

4.1.1. Definizioni e proposizioni

Un *albero radicato* (*rooted tree*) T è una collezione di nodi che, se non è vuota, soddisfa le seguenti proprietà:

- \exists un nodo speciale $r \in T$ (r è chiamato *radice*);
- $\forall v \in T, v \neq r: \exists! u \in T: u$ è padre di v (v è figlio di u);
- $\forall v \in T, v \neq r: \text{risalendo di padre in padre si arriva a } r$ (ovvero ogni nodo è discendente dalla radice).

Nel libro di testo la terza condizione manca. Senza questa, la seguente collezione con u padre di v e v padre di u sarebbe un albero, che ha poco senso. Questa collezione piuttosto è una foresta di alberi.

