

Ripasso di Java

Programma Stand-alone

File: MyProgram.java

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

Compilazione:

```
javac MyProgram.java
```

Crea un **bytecode** **MyProgram.class** eseguibile sulla **Java Virtual Machine (JVM)**

Esecuzione:

```
java MyProgram
```

Oss. Richiede che la directory corrente sia nel CLASSPATH. Altrimenti usiamo: `java -cp . MyProgram`

In alternativa si può usare un *Integrated Development Environment (IDE)*, ad es. IntelliJ IDEA, Eclipse, JBuilder

Astrazione e Incapsulamento (Information Hiding):

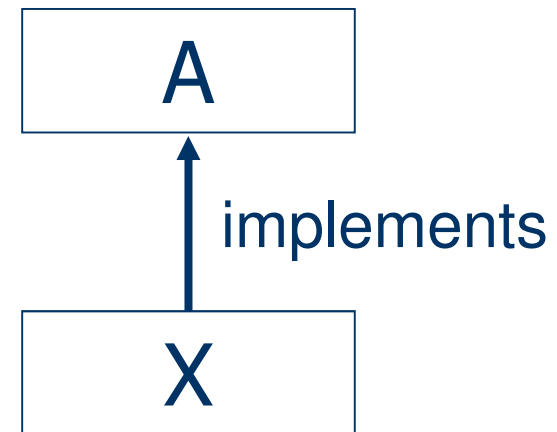
- Si astrae la **specifica** delle funzionalità, separandola dalla loro **implementazione**
- Uso delle funzionalità solo in base alla specifica
- L'implementazione delle funzionalità è nascosta
 - Gli errori sono più confinati (-> *robustezza*)
 - Possibilità di cambiare l'implementazione senza cambiare la specifica (-> *adattabilità*)
- Nel caso delle strutture dati, questo approccio dà origine alla nozione di **Abstract Data Type (ADT)**

Abstract Data Type

- Modello di struttura dati che specifica: il tipo di dati; le operazioni disponibili per accedere a essi e modificarli; e i parametri usati da tali operazioni.
- In Java si realizza principalmente tramite **interfacce** che contengono *solo dichiarazioni di costanti e dichiarazioni di metodi senza corpo*.
- In Java esistono poi anche **classi astratte** che sono classi che possono contenere anche dichiarazioni di metodi senza corpo (come le interfacce)

INTERFACCIA

```
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 es.,
java.lang.String
implements
java.lang.Comparable

Programmazione Generica (*generics*)

- È un tipo di polimorfismo, introdotto da Java SE 5. 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 (ad es., int) 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.

Esempi: interfacce e classi generiche

```
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  
estensioni della classe/interfaccia S
```

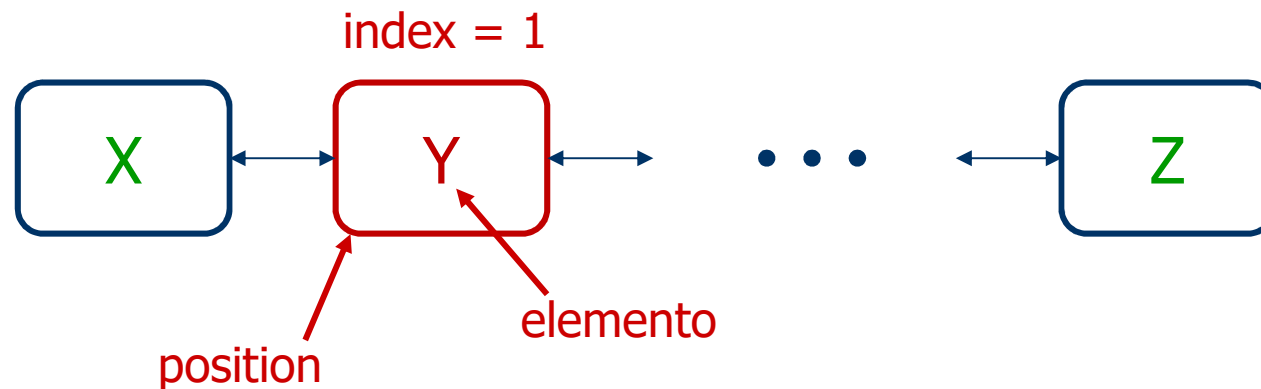

ADT ELEMENTARI

- **LISTA (LIST)**
 - Index-based
 - Position-based
- **PILA (STACK)**
- **CODA (QUEUE)**

Lista

LISTA (LIST): collezione di elementi organizzati secondo un ordine lineare tale per cui è identificabile il primo elemento, il secondo, ... , e l'ultimo.

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



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)  
}
```

In Java: presente nel package *java.util* e implementata da diverse classi tra cui *ArrayList*

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();  
    ....
```

Lista position-based

```
....  
/** 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 and 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);  
}
```

IMPLEMENTAZIONI DELLA LISTA

- **Lista index-based:** tramite array. I metodi possono essere implementati con complessità $O(1)$ tranne *add* e *remove* che richiedono complessità $O(n)$ (n =numero di elementi nella lista)
- **Lista position-based:** tramite doubly-linked list.
 - Ogni nodo diventa un oggetto a sé con variabili di istanza che «puntano» a predecessore e successore
 - Inizio e fine lista delimitati da *nodi sentinella*
 - metodi possono essere implementati con complessità $O(1)$ ma la lista può essere scandita solo sequenzialmente

Oss.: è possibile implementare una lista index-based tramite doubly-linked list, o una lista position-based tramite array ma con scarsi vantaggi.

PILA (STACK): 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();  
}
```

In Java: nel package *java.util* esiste una classe *Stack*, ma si consiglia di usare implementazioni dell'interfaccia *Deque*

Coda

CODA (QUEUE): 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();  
}
```

In Java: nel package *java.util* esiste la interfaccia *Deque* (*Double-ended queue*) che permette inserimento e rimozione in testa e in coda (Pila+Coda).

IMPLEMENTAZIONI DELLA PILA e DELLA CODA

- Entrambe le strutture dati possono essere implementate tramite **doubly-linked list**.
- **PILA**: inserimento (*push*) e rimozione (*pop*) in testa alla lista.
- **CODA**: inserimento (*enqueue*) in coda e 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 elementi intermedi a meno di non rimuovere quelli che li precedono nell'ordine di accesso

Oss.: è possibile usare anche una *singly-linked list*.

Collection Framework

Fa parte del package `java.util` e contiene:

- Interfacce di varie collezioni
- Classi che implementano le interfacce
- Algoritmi (polimorfi) per operare su collezioni (ad es. sorting)
- Ampio uso della programmazione generica

Riepilogo

- Compilazione ed esecuzione di programmi in Java
- Abstract Data Type (ADT), con riferimento a Java
- ADT Elementari:
 - Lista (index-based e position-based)
 - Pila
 - Coda
- Collection Framework

Sul sito del corso viene fornita una cartella compressa contenente l'implementazione in Java del calcolo dei numeri di Fibonacci (iterativo e ricorsivo) e di InsertionSort. La cartella contiene anche un file Readme.pptx che aiuta nella comprensione dei programmi.

Esempio domanda prima parte

1. Definire brevemente il concetto di Abstract Data Type (ADT) caratteristico dell'approccio object-Oriented, e spiegare in che modo Java permette di specificare un ADT.

ERRATA

- Lucidi 15 e 16: corretto il fatto che Deque è un interfaccia non una classe
- Lucido 16: il tipo di ritorno del metodo enqueue è stato aggiornato a void.