

STRUTTURE DATI: OLTRE GLI ARRAY

- le strutture dati progettate per ospitare una *collezione di elementi*, sono variazioni di array
- Ma l'array ha dimensione fissa anche in Java
 - determinata a priori, in linguaggi statici
 - determinata al momento della creazione, se definiti dinamicamente (Java)
- Molti problemi richiedono strutture dati in grado di rappresentare collezioni di elementi *di numero variabile dinamicamente*
- Efficienza nell'uso delle risorse

Liste - 1

LISTE

- Una *lista* è una *collezione di elementi* organizzati concettualmente *in sequenza*
- La dimensione della *lista non è prefissata*

Progetto in Java

- Uso della semantica per riferimento
- Realizzare delle classi corrispondenti

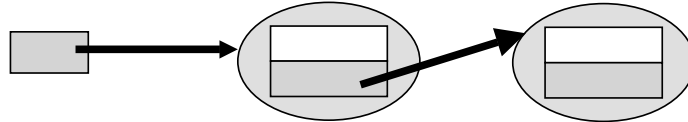
Decisioni progettuali

- Una lista come contenitore o come valore
- Quali operazioni? Quali costruttori?
- Quali operazioni *primitive*?

Liste - 2

LISTE: RAPPRESENTAZIONE

- Una *lista* è spesso rappresentata sotto forma di *sequenza di nodi concatenati*

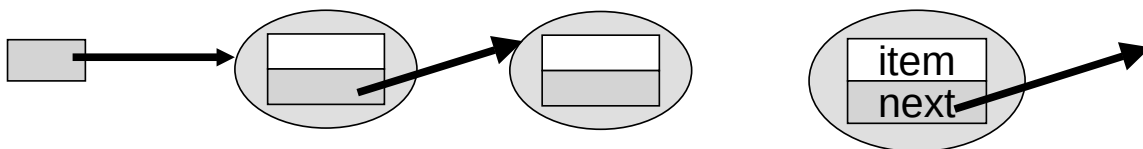


- Ogni nodo della lista è fatto di *due parti*: un *valore* e un *riferimento al nodo successivo*
- La *rappresentazione fisica* può essere:
 - basata su puntatori e allocazione dinamica
 - basata su oggetti creati dinamicamente
 - basata su altre rappresentazioni (file, array, ...)

Liste - 3

LISTE

- Consideriamo un generico elemento

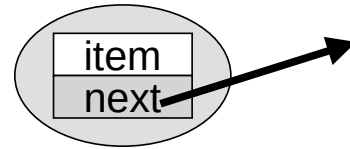


```
public class ListInt {  
    private ListInt next;  
    private Integer item;  
  
    public ListInt()  
    { next = null; item = null; }  
    ...  
}
```

Liste - 4

LISTE: funzionalità

```
public class ListInt {  
    private ListInt next;  
    private Integer item;
```

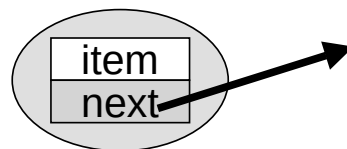


```
public ListInt() {}  
public void insert (int i) {}  
public Integer extract () {}  
  
public boolean isEmpty () {}  
public void printList () {}  
}
```

Liste - 5

LISTE: funzionalità

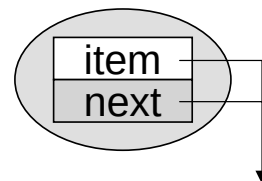
```
public class ListInt {  
    private ListInt next;  
    private Integer item;
```



```
public ListInt() {next = null; item = null;}  
}
```

L'elemento viene creato senza riferire
a nulla

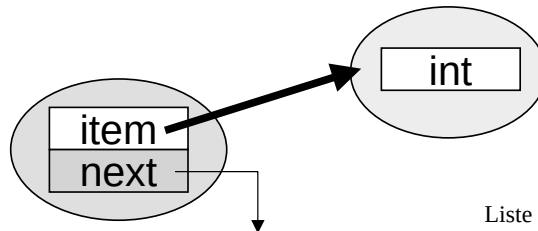
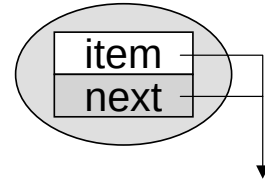
Si noti che consideriamo di riferire
un altro (next) e un wrapper Integer (item)



Liste - 6

LISTE: inserimento

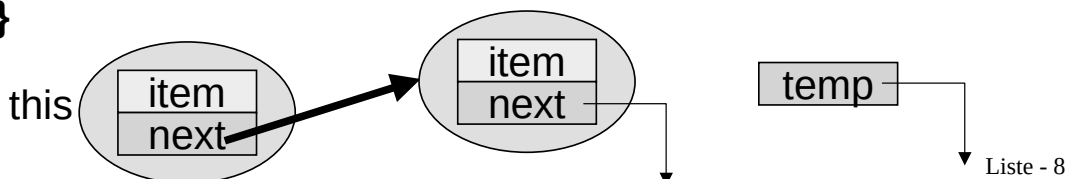
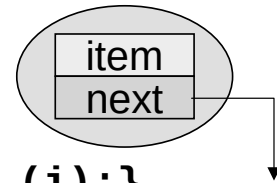
```
public class ListInt {
    private ListInt next;
    private Integer item;
    public void insert (int i) {
        if (item==null){item = new Integer (i);}
        else {
            ListInt temp = next;
            next = new ListInt();
            next.item = item; item = new Integer (i);
            next.next = temp;
        }
    }
}
```



Liste - 7

LISTE: inserimento

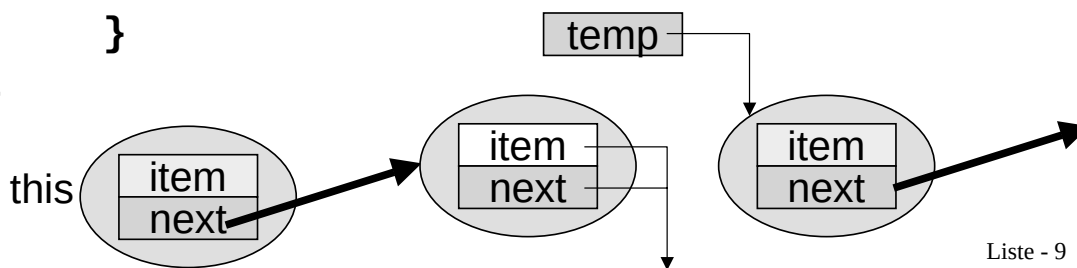
```
public class ListInt {
    public void insert (int i) {
        if (item==null){item = new Integer (i);}
        else { // item non nullo
            ListInt temp = next;
            next = new ListInt();
            next.item = item; item = new Integer (i);
            next.next = temp;
        }
    }
}
```



Liste - 8

LISTE: inserimento

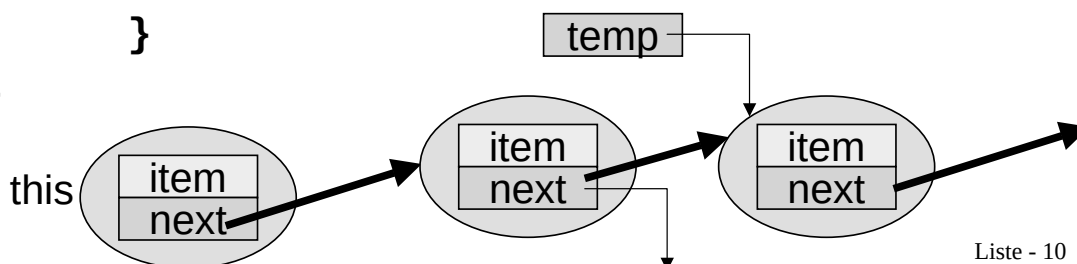
```
public class ListInt {
public void insert (int i) {
    if (item==null){item = new Integer (i);}
    else { // item non nullo
        ListInt temp = next;
        next = new ListInt();
        next.item = item; item = new Integer (i);
        next.next = temp;
    }
}
```



Liste - 9

LISTE: inserimento

```
public class ListInt {
public void insert (int i) {
    if (item==null){item = new Integer (i);}
    else { // item non nullo
        ListInt temp = next;
        next = new ListInt();
        next.item = item; item = new Integer (i);
        next.next = temp;
    }
}
```



Liste - 10

LISTE: estrazione

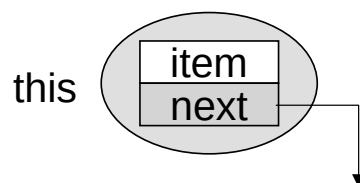
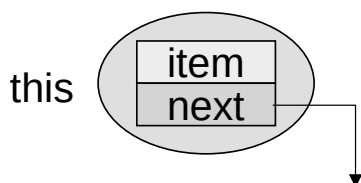
```
public Integer extract () { Integer iI;  
    if (item == null) {return null;}  
    else  
    { iI = item;  
      if (next == null) item = null;  
      else {item = next.item;  
            next = next.next;}  
      return iI;  
    }  
}
```

Si noti il ruolo di Integer (wrapper) per indicare che non si restituisce alcun valore

Liste - 11

LISTE: estrazione

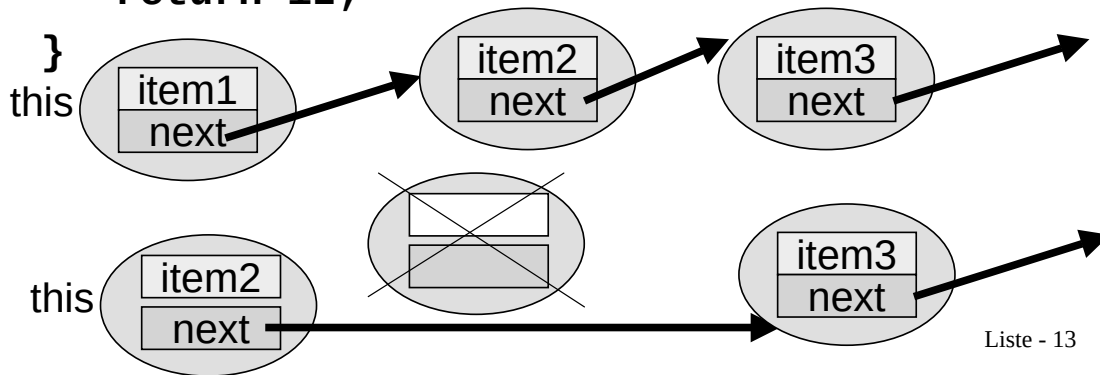
```
public Integer extract () { Integer iI;  
    ... else  
    { iI = item;  
      if (next == null) item = null;  
      else {item = next.item;  
            next = next.next;}  
      return iI;  
    }  
}
```



Liste - 12

LISTE: estrazione

```
public Integer extract () { Integer iI;  
    ... else  
    { iI = item;  
      if (next == null) item = null;  
      else {item = next.item;  
            next = next.next;}  
      return iI;  
    }  
}
```



Liste - 13

LISTE: funzionalità

```
public class ListInt {  
    private ListInt next; private Integer item;  
    public ListInt() {...}  
    public void insert (int i) {...}  
    public Integer extract () {...}  
  
    public boolean isEmpty ()  
        {return (item == null);}  
    public void printList () {}  
    // stampa la lista in formato esterno  
}
```

Liste - 14

LISTE: printList

```
public void printList () {
    ListInt temp = next; int i = 1;
    if (item == null) System.out.println (" Lista vuota ");
    else
    {System.out.println(" Elem N.1 vale "+ this.item);
     while (temp != null)
     {
         i++;
         System.out.println (" Elem N." + i +
                               " vale " + temp.item);
         temp = temp.next;
     }
    }}
}
```

Liste - 15

LISTE: toString

```
public String toString () {
    ListInt temp = next; int i = 1; String s;
    if (item == null) return " Lista vuota \n";
    else
    {s = "\n Lista con elementi " + item;
     while (temp != null)
     { s = s + "\t" + temp.item;
       temp = temp.next; i++; }
     return s = s + "\n In totale sono "
               + i + " elementi\n";
    }
}
```

Liste - 16

USO da parte di CLIENTI

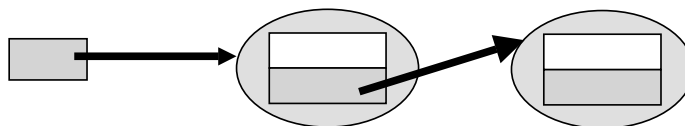
```
public static void main(String[] args) {
    int i; Integer iI;
    ListInt first = new ListInt ();
    first.insert(2); first.insert(6);
    first.insert(17);
    iI = first.extract();
    if (iI != null) { i = iI.intValue(); ...}
    first.insert (19);
    first.printList ();
    while (first.isEmpty ())
        { i = first.extract().intValue();}
}
```

Liste - 17

PROGETTO DI LISTE IN JAVA

List: quale rappresentazione?

- approccio classico:
 - un campo value che rappresenta il valore (Object)
 - un campo next che punta al prossimo nodo (AnyList)



- **semplice, ma con alcuni difetti**
 - la lista non esiste come entità esplicita (coincide con il riferimento al primo nodo)
 - ciò rende difficile avere semantiche di costruzione diverse da quella predefinita (inserimento in testa)

Liste - 18

PROGETTO DI LISTE IN JAVA

List: quale

- approccio

Poiché il valore è un **Object**, la lista può ospitare solo oggetti (non variabili di tipi primitivi)

- un campo **value** che rappresenta il valore (**Object**)
- un campo **next** che punta al prossimo nodo (**AnyList**)

Al posto dei tipi primitivi occorre dunque utilizzare le corrispondenti classi wrapper (**int** → **Integer**, **float** → **Float**, etc.)

- **semplice, ma con alcuni difetti**

- la lista non esiste come entità esplicita (coincide con il riferimento al primo nodo)
- ciò rende difficile avere semantiche di costruzione diverse da quella predefinita (inserimento in testa)

Liste - 19

ANALISI CRITICA DEL PROGETTO

L'approccio classico è soddisfacente?

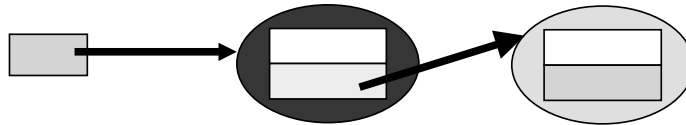
- non del tutto, in quanto la lista non esiste come entità esplicita: coincide con il riferimento al primo nodo
 - la classe **List** in realtà rappresenta il nodo della lista, non la lista in quanto tale!
- ciò rende impossibile avere semantiche di costruzione *diverse da quella predefinita* (inserimento in testa)
 - ad esempio, è impossibile derivare da **List** una classe **SortedList** che costruisca liste ordinate

Liste - 20

ANALISI CRITICA DEL PROGETTO

Perché questo?

- Perché il costruttore di una tale ipotetica classe `SortedList` chiamerebbe come prima cosa il costruttore della classe base `List`, che crea già un nuovo nodo e lo inserisce in testa!



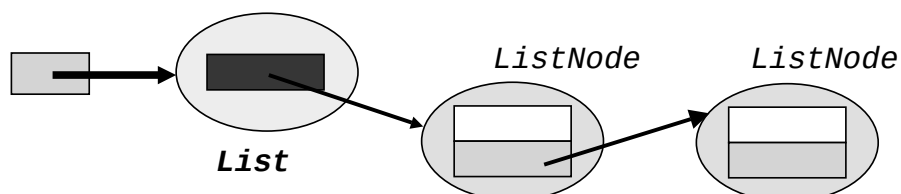
- Nessuno può revocare quel comportamento, indipendentemente dal costruttore di `SortedList`
- Il cliente riceverebbe in ogni caso il riferimento al nuovo nodo creato in testa.

Liste - 21

REVISIONE DEL PROGETTO

Una diversa rappresentazione

- `List` rappresenta la lista: è un oggetto che contiene *la radice della sequenza di nodi*
 - tale radice è *un riferimento a un oggetto `ListNode`*
- `ListNode` rappresenta il nodo della lista
 - contiene il valore e il riferimento al nodo successivo
 - è quello che prima chiamavamo impropriamente lista



Liste - 22

REVISIONE DEL PROGETTO

Perché questo approccio risolve il problema?

- Perché ora creare una *List* *non implica più* creare anche un nodo!
- Quindi, il costruttore di *List* può gestire direttamente la creazione del nodo, *inserendolo dove ritiene più appropriato*
- Ridefinendo il proprio costruttore, una sottoclasse *SortedList* può cambiare tale comportamento
- Il cliente riceve in ogni caso il riferimento alla nuova lista creata, non già a uno specifico nodo!

Liste - 23

Nuovo Progetto

```
public class ListNode implements Iposition {  
    // Implementazione dell'interfaccia "IPosition"  
    protected Object item; protected ListNode next;  
  
    public ListNode(Object o) { item = o; next=null;}  
    public Object getItem(){ return item;}  
  
    public ListNode getNext() { return next;}  
    public Object container() {return null; }  
    // modificare per riferire la lista cui nodo appartiene  
    public void setNext(ListNode next){ this.next=next;}  
}
```

Liste - 24

INTERFACCIA per NODO di LISTA

```
public interface IPosition
{
    public Object getItem ();
    public Object container();
}
```

La Interfaccia consente ad un nodo di trovare il prossimo e di riferire una ed una sola lista

Liste - 25

INTERFACCIA LISTA

```
public interface IList
{
    // Inserimento oggetto a inizio e a fine lista
    public void insert(Object o)
    public void append(Object o)
    // Rimozione primo e ultimo elemento
    public Object removeFirst() throws ListEmptyException
    public Object removeLast() throws ListEmptyException
    // Restituire il primo elemento e ultimo lista
    public Object first() throws ListEmptyException
    public Object last() throws ListEmptyException
    // Numero elementi e predicato di lista vuota
    public int size()                public boolean isEmpty()
```

Liste - 26

LISTA SEMPLICE (link in avanti)

```
public class SimpleList implements IList
{private int size; private ListNode first;
// Inserimento oggetto a inizio e a fine lista
public void insert(Object o);
public void append(Object o);
// Rimozione primo e ultimo elemento
public Object removeFirst() throws ListEmptyException;
public Object removeLast() throws ListEmptyException;
// Restituire il primo elemento e ultimo lista
public Object first() throws ListEmptyException;
public Object last() throws ListEmptyException;
// Numero elementi e predicato di lista vuota
public int size();          public boolean isEmpty();
```

Liste - 27

Inserimento in prima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;

// Inserimento oggetto all'inizio della lista
public void insert(Object o)
{ ListNode node = new ListNode(o);
  node.setNext(first);
  first = node;
  size++; }
```

Liste - 28

Inserimento ultima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;

// Inserimento alla fine della lista
public void append(Object o)
{ ListNode newNode = new ListNode(o);
  ListNode node = first;
  while (node.getNext() != null)
    node = node.getNext();
  node.setNext(newNode);
  size++;
}
```

Liste - 29

Estrazione dalla prima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;

// Rimozione primo elemento
public Object removeFirst() throws
  ListEmptyException
{ if(isEmpty()) throw new
  ListEmptyException("lista vuota!");
  ListNode node = first;
  first = node.getNext(); size--;
  return node.getItem();
}
```

Liste - 30

Estrazione dalla ultima posizione

```
public class SimpleList implements IList
{private int size; private ListNode first;
public Object removeLast() throws ListEmptyException
{if (isEmpty())
    throw new ListEmptyException("lista vuota!");
if (first.getNext() == null)
{Object o = first. getItem(); first = null; return o;}
else
{ListNode node=first.getNext(); ListNode pnode=first;
while (node.getNext() != null)
    {pnode = node; node= node.getNext();}
Object o=node.getItem (); pnode.setNext(null); size--;
return node.getItem();
}
}
```

Liste - 31

Accessori

```
public Object first() throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException(" vuota!");
return first.getItem();}
```

```
public Object last() throws ListEmptyException
{if(isEmpty()) throw new ListEmptyException("vuota!");
ListNode node = first;
while (node.getNext()!= null) node= node.getNext();
return node.getItem();
}
```

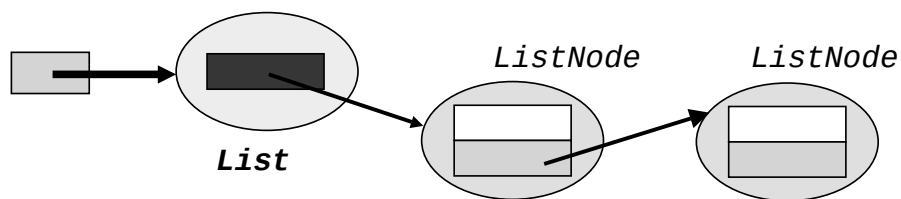
```
public int size(){ return size;}
public boolean isEmpty(){ return first==null;}
```

Liste - 32

IL PROGETTO REVISIONATO

Ora è semplice derivare la classe SortedList

- una SortedList specializza la classe base List nel senso che *il suo costruttore inserisce l'oggetto nella posizione opportuna in base all'ordinamento* anziché sempre in test



Liste - 33

LISTE COME CONTENITORI

- Approccio imperativo: una lista è una sequenza di contenitori di elementi, di lunghezza non stabilita a priori

Conseguenze

- Una lista è un contenitore
- Si può *cambiare* una lista esistente
- È un approccio efficiente (non si ricostruisce inutilmente ciò che già esiste)
- Ma può essere pericoloso in caso di *structure sharing* e *aliasing* (che quindi vanno evitati!)

Liste - 34

JAVA: LE CLASSI DI UTILITÀ

Il package `java.util` definisce:

- ***interfacce*** che rappresentano *tipi di dati astratti di contenitori*
 - `Collection` → `List`,
 - `Collection` → `Set` → `SortedSet`
- ***classi*** che forniscono *contenitori concreti*
 - `LinkedList`, `Vector`, `HashSet`, `TreeSet`, ...

I metodi principali (`add`, `remove`, `contains`, `isEmpty`, ...) sono introdotti da `Collection` e specializzati dalle varie classi

Liste - 35

JAVA: LE LISTE PREDEFINITE

- L'interfaccia `java.util.List` aggiunge ai metodi introdotti da `Collection` altri metodi, specifici del concetto di lista
- La classe `java.util.Vector` implementa `List` definendo un componente che cattura l'idea di *array illimitatamente espandibile*
- La classe `java.util.LinkedList` implementa `List` definendo l'astrazione *lista doppiamente concatenata*

Si veda la documentazione di Java per i dettagli

Liste - 36