Liste

Esistono dei tipi predefiniti in Java per liste e insiemi

Breve riepilogo

Variabili:

1. ogni oggetto si può mettere in `Object`
2. per il passo inverso, serve il cast

Quando la variabile è di un tipo ma l'oggetto è di una sottoclasse:

1. le componenti sono quelle della classe dalla variabile
2. i metodi sono quelli della classe dell'oggetto

Nota: questo discorso vale solo per i metodi e le componenti che esistono in tutte e due le classi

Se un metodo esiste solo nella classe dell'oggetto ma non in quella della variabile, non si può invocare

---

Ridefinizione dei metodi di `Object`

1. si definisce `public String toString()` come un metodo che ritorna una stringa ottenuta concatenando le componenti
2. si definisce `public boolean equals(Object o)` come un metodo che:
    1. confronta `o` con `null`
    2. fa il cast di `o` alla classe
    3. confronta le componenti di `o` e `this`

```
class Studente {
  String nome;
  int anno;

  public String toString() {
    return "["+nome+","+anno+"]";
  }

  public boolean equals(Object o) {
    Studente s;

    if(o==null)
      return false;

    if(this.getClass()!=o.getClass())
      return false;

    s=(Studente) o;

    if(this.nome==null) {
      if(s.nome!=null)
        return false;
    }
    else
      if(!this.nome.equals(s.nome))
```

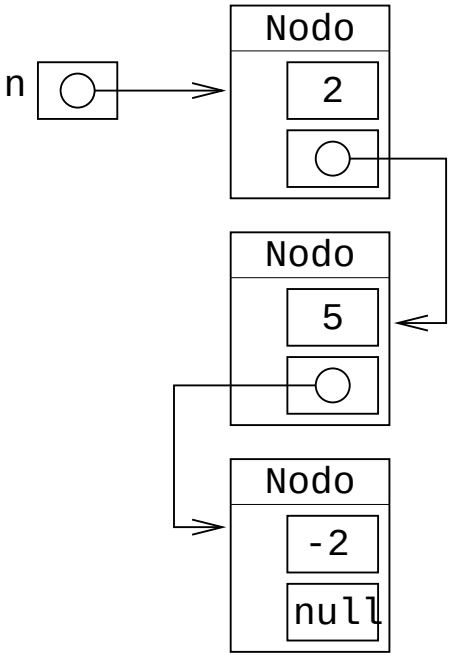
```
        return false;

    if(this.anno!=s.anno)
      return false;

    return true;
  }
}
```

Liste collegate

Si rappresentano sequenze di elementi usando un oggetto per ogni elemento

Ogni oggetto è collegato al successivo

```
class Nodo {
  int info;
  Nodo next;
}
```



Liste di oggetti

Al posto di `int info`, posso mettere un tipo qualsiasi (es. `Point`)

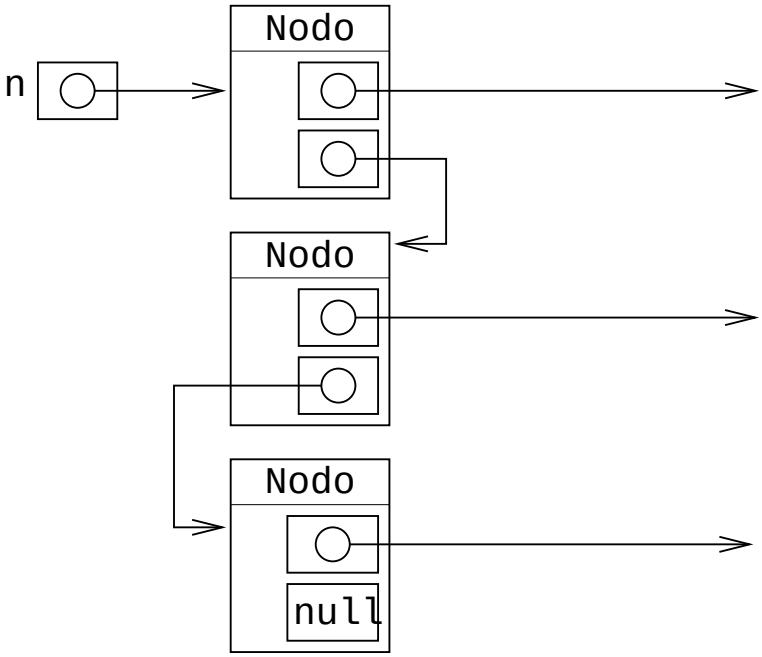
In questo modo, posso rappresentare sequenze di oggetti `Point`

Usando `Object` come tipo di `info`, ho una sequenza di oggetti qualsiasi

```
class Nodo {
  Object info;
  Nodo next;
}
```

Catene di oggetti, in memoria

Il campo `info` contiene l'indirizzo di un oggetto



Al posto di un intero, nei campi `info` ci sono riferimenti ad altri oggetti

Ci posso mettere riferimenti a `Point`, `Rectangle`, `Studente`, ecc

Cosa posso fare con una lista

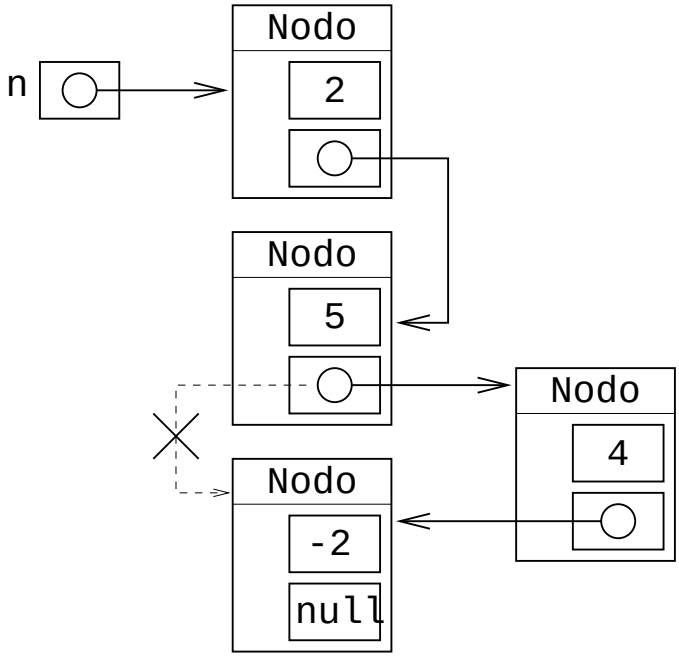
Ci sono varie operazioni che si possono fare:

1. verifica se è vuota
2. trovare la lunghezza
3. trovare l'elemento in una certa posizione
4. inserire un elemento in una certa posizione
5. eliminare l'elemento in una certa posizione

Alcune sono complicate da realizzare

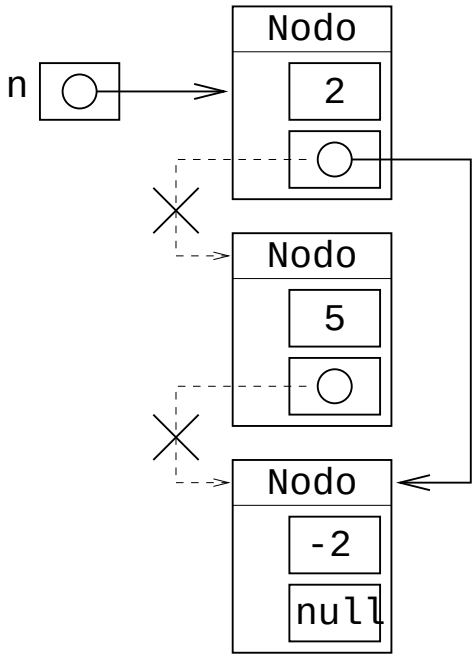
Cosa succede se si inserisce un elemento

Si può mettere all'inizio della catena, alla fine, o anche in mezzo:



Eliminazione di un elemento

Basta fare in modo che la catena di riferimenti ‘‘aggiri’’ l’elemento da eliminare:



Per eliminare il secondo:

```
n.next=n.next.next;
```

In generale, si può eliminare un elemento qualsiasi

Semplicità di realizzazione

Sulle liste collegate, le operazioni di inserimento e cancellazione di un elemento sono semplici

D'altra parte, per arrivare all'i-esimo elemento, devo scandire tutta la lista (sugli array questo non è necessario)

Tipo `LinkedList`

È un tipo di oggetto predefinito di Java

Realizza liste collegate, in cui il campo `info` è `Object`

Sono liste collegate di oggetti

È un tipo predefinito del linguaggio: non c'è bisogno di sapere come sono implementati i metodi

Se si usa una `LinkedList` va fatto:

```
import java.util.*;
```

Come si usa

Una lista è una sequenza di oggetti

Un singolo oggetto `LinkedList` rappresenta una sequenza di oggetti

```
LinkedList l;
l=new LinkedList();
```

Dopo la creazione, la lista è vuota (è una sequenza di zero elementi)

Metodo `add`: inserisce un elemento in fondo alla lista

```
l.add(new Point(12,3));
l.add("abcd");
l.add(new Studente("Pippo", 3));
```

Stampa della lista

Si può stampare una lista:

```
System.out.println(l);
```

Per ogni oggetto della lista, si invoca `toString` e si stampa il risultato

Tutti gli oggetti hanno il metodo `toString`: o è quello di `Object`, oppure è stato ridefinito

---

Stampa della lista

Stampa della lista di sopra se `toString` non viene ridefinito in `Studente`:

```
[java.awt.Point[x=12,y=3], abcd,
Studente@5d87b2]
```

Stampa l'indirizzo dell'oggetto, non i dati dello studente!

Se ridefinisco `toString` di `Studente`:

```
[java.awt.Point[x=12,y=3], abcd,
[Pippo 3]]
```

Adesso vedo i dati dello studente!

Trovare la lunghezza di una lista

Metodo `size`, che ritorna un intero

```
import java.util.*;
import java.awt.*;

class Prova {
  public static void main(String args[]) {
    LinkedList l;
    l=new LinkedList();

    l.add(new Point(12,3));
    l.add("abcd");
    l.add(new Studente("Pippo", 3));

    System.out.print("Lunghezza lista: ");
    System.out.println(l.size());
  }
}
```

Inserire un elemento in mezzo

Il metodo `add` è sovraccarico

```
  // inserisce elemento alla fine
 l.add("efg");
```

C'è una versione con un intero

```
  l.add(2, "efg");
```

Inserisce l'elemento in posizione `2`

Numerazione degli elementi

Attenzione!

Gli elementi sono numerati da `0` a `size()-1`

Quando faccio `l.add(2, "efg")`, l'elemento `"efg"` viene inserito nella posizione `2`, ossia in terza

Gli elementi successivi vengono scalati (in ordine):

```
Prima:   [abcd, ert, wwww, zzzz]
faccio   l.add(2, "nuova");
Dopo:    [abcd, ert, nuova, wwww, zzzz]
```

L'elemento viene inserito in posizione `2` (la terza)

Tutti quelli dopo sono spostati in ordine

Regola degli indici

È la stessa degli array:

l'elemento di indice `0` è il primo

Vale per gli array, per le liste, e per varie altre strutture dati

Regola della dimensione:

l'ultimo elemento ha indice `num_elementi-1`

Questa regola vale anche in altri linguaggi

È facile sbagliare quando si fanno cicli su una parte della sequenza
(es. stampare tutti gli elementi di una lista tranne gli ultimi `n`)

Dove va l'elemento?

```
l.add(i, oggetto);
```

Regola dell'inserimento:

l'oggetto diventa l'oggetto in posizione `i` nella lista

Esempio:

`l.add(3, "abcd");`
l'oggetto `"abcd"` diventa il nuovo elemento di indice `3` della lista (il quarto)
`l.add(0, "efgh");`
l'oggetto `"efgh"` diventa il nuovo elemento in posizione `0` (diventa il nuovo primo elemento)

Mettere `null`

Posso anche aggiungere `null` come elemento di una lista

```
l.add(null);
```

Concetto: una `LinkedList` è una sequenza di variabili `Object`

In queste variabili posso anche mettere `null`

Trovare un elemento

Ci sono tre metodi:

`getFirst()`
: trova il primo oggetto della lista

`getLast()`
: trova l'ultimo oggetto della lista

`get(int index)`
: trova l'elemento di indice `index`

Tutti e tre restituiscono un `Object`

Cast sugli elementi

Se so che sono di un tipo specifico, posso fare il cast:

```
String s;

s=(String) l.getFirst();
```

Dà errore se poi l'oggetto non è di tipo stringa!

È un errore in esecuzione

Di solito, si fanno liste che contengono tutti elementi dello stesso tipo

---

Perchè devo fare il cast?

```
Point p, q;
...

l.add(p);
q=(Point) l.getLast();
```

Domanda:

- Quando metto dentro `p`, è un `Point`
- perchè, quando lo tiro fuori, devo fare il cast per rimetterlo in una variabile `Point`?

Cast: risposta

Concettualmente, le liste sono sequenze di oggetti `Object`

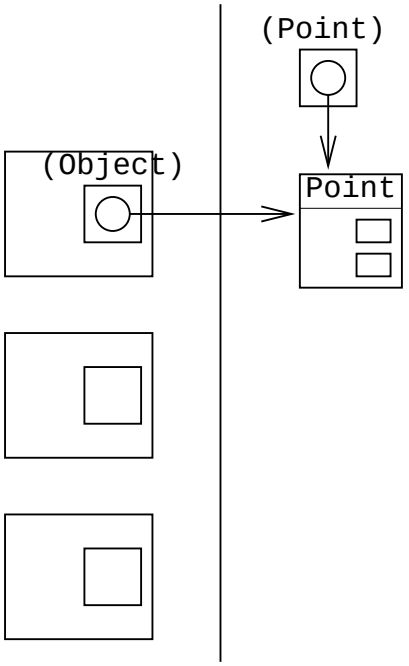
L'implementazione Java: una lista è una sequenza di variabili (d'istanza) `Object`

inserire
: in una variabile `Object` posso mettere un qualsiasi riferimento a oggetto

trovare
: se ho un `Point` in una variabile `Object`, devo fare il cast per poterlo rimettere in un `Point`

Graficamente

Quando inserisco un oggetto in una `LinkedList`, viene messo in una variabile `Object`



Inserimento:
: faccio `Object=Point` non ho bisogno del cast

Trovare elemento:
: per fare `Point=Object` devo fare il cast, anche se l'oggetto è un `Point`

Notare l'incapsulamento: non so come è fatta la `LinkedList`

È come se ci fosse una ''barriera'', che mi impedisce di vedere la struttura degli oggetti

So solo che dentro ci sono delle variabili `Object` per memorizzare gli oggetti.

Queste variabili sono componenti di oggetti collegati fra loro, ma la cosa non mi interessa.

Esercizio

Stampare gli elementi di una lista, uno per linea

Notare che `println(lista)` li stampa tutti su una linea

```
static void stampaLista(LinkedList l) {
  ...
}
```

Soluzione

`l.get(i)` trova l'oggetto in posizione `i`

Gli elementi della lista hanno indici che vanno da `0` a `l.size()-1`

```
static void stampaLista(LinkedList l) {
  int i;
  Object o;

  for(i=0; i<l.size(); i++) {
    o=l.get(i);

    System.out.println(o.toString());
  }
}
```

Osservazione

`l.get(i)` ritorna un `Object`, che metto in `o`

`o.toString()` è l'invocazione del metodo `toString` dell'oggetto il cui riferimento sta in `o`

Se questo oggetto è un `Point`, allora `o.toString()` è l'invocazione del metodo `toString` di `Point`, anche se la variabile `o` è un `Object`

Nota: questo vale anche se faccio `l.get(i).toString()`: il risultato di `l.get(i)` è di tipo `Object`, per cui è come se faccessi `Object o=l.get(i); o.toString();`

Esercizio

Scrivere un metodo statico che verifica la presenza di un oggetto in una lista

```
static boolean
   presente(LinkedList l, Object o) {
  ...
}
```

Voglio sapere se la lista contiene un oggetto che è `equals` ad `o`

Soluzione

Per ogni elemento della lista, se è `equals` ad `o`, ritorna `true`

Se si arriva alla fine senza aver trovato l'elemento, si ritorna `false`

```
static boolean presente(LinkedList l, Object o) {
  int i;

  for(i=0; i<l.size(); i++) {
    if(o.equals(l.get(i)))
      return true;
  }

  return false;
}
```

Attenzione! Quando faccio `o.equals` viene invocato l'`equals` dell'oggetto!

Se l'oggetto è un `Point`, viene invocato `equals` di `Point`, anche se `o` è una variabile `Object`

Più facile

Nella classe `LinkedList` c'è il metodo `contains`:

```
boolean contains(Object)
```

Verifica se l'elemento sta nella lista oppure no

Verifica coordinate negative

Scrivere un metodo statico che verifica se una lista di punti contiene un punto con coordinata `x` negativa

```
static boolean puntiNegativi(LinkedList l) {
  ...
}
```

Soluzione sbagliata

Algoritmo simile: se si trova un elemento uguale, si ritorna `true`

```
static boolean puntiNegativi(LinkedList l) {
  int i;

  for(i=0; i<l.size(); i++)
    if(l.get(i).x<0)
      return true;

  return false;
}
```

Quando si compila, genera un errore

Quale?

Non esiste la componente `x`

L'errore che viene dato è questo:

```
Negative.java:9: cannot resolve symbol
symbol  : variable x
location: class java.lang.Object
      if(l.get(i).x<0)
               ^
1 error
```

Significa: l'oggetto `l.get(i)` non ha la componente `x`

È vero?

Variabili e oggetti

L'oggetto `l.get(i)` è di tipo `Object`

La classe `Object` non ha la componente `x`

`l.get(i).x` è un errore

Verifiche statiche e dinamiche

Se `o` è una variabile `Object`, vengono fatti questi controlli:

staticamente:
: `o` ha solo le componenti e metodi di `Object`

dinamicamente:
: si può fare il cast `(Classe) o` solo se l'oggetto che si trova *attualmente* nella variabile è di un tipo che si può mettere in una variabile `Classe`

Principio: in fase di compilazione, non si può sapere con certezza il tipo dell'oggetto che sta in una variabile

Si assume quindi che abbia solo le componenti e i metodi del tipo della variabile

---

Variabili, metodi ed espressioni

Lo stesso discorso vale per:

- variabili `Object`
- metodi che ritornano un `Object`
- espressioni che hanno come risultato un `Object`

Valgono le stesse regole che abbiamo detto per le variabili `Object`

Cast

Dato che l'oggetto è di tipo `Point`, posso fare il cast a `Point`

```
Point e=(Point) l.get(i);
```

Ora posso fare `e.x` ecc.

Soluzione corretta

Prima faccio il cast, e poi vedo la componente `x`

```
static boolean puntiNegativi(LinkedList l) {
  int i;
  Point p;

  for(i=0; i<l.size(); i++) {
    p=(Point) l.get(i);

    if(p.x<0)
      return true;
  }

  return false;
}
```

Si poteva anche fare:

```
if(((Point) l.get(i)).x<0)
```

L'espressione `((Point) l.get(i)).x` ritorna un `Point`, quindi si può fare `.x`

Osservazione

Se nella lista ci metto un rettangolo, e poi invoco il metodo, viene dato errore

Il cast `(Point) oggetto` si può fare solo se la variabile contiene effettivamente un `Point`

testo del problema:
‘‘data una lista di punti’’

in Java
non esiste il tipo ‘‘lista di punti’’

È una affermazione su come verrà invocato il metodo: solo su liste in cui tutti gli oggetti sono di tipo `Point`

Quindi, il cast `e=(Point) l.get(i)` non dà errore se la lista passata rispetta la specifica

Non c’è modo di generare un errore in compilazione se si passa una lista con oggetti che non sono `Point`

Altro esercizio

Realizzare un metodo statico che prende una lista di punti, che sono in ordine di `x` crescente, e inserisce un nuovo punto nella posizione giusta

```
static void inserisciPuntoOrdine(
  LinkedList l, Point p) {
  ...
}
```

Soluzione

Primo livello di raffinamento dell’algoritmo:

1. trova la posizione in cui va messo l’elemento
2. inseriscilo

La seconda parte è facile

Versione che non funziona

Viene dato un errore:

```
static void inserisciPuntoOrdine(
  LinkedList l, Point p) {

  int i;

  for(i=0; i<l.size(); i++) {
    if(l.get(i).x>=p.x) {
      l.add(i, p);
      return;
    }
  }

  l.add(p);
}
```

È un errore in compilazione: quale?

Non esiste la componente `x`

L’errore è:

```
Prova.java:30: cannot resolve symbol
symbol  : variable x
location: class java.lang.Object
      if(l.get(i).x>=p.x) {
               ^
1 error
```

Motivo: `l.get(i)` è di tipo `Object`, che non ha il campo `x`

Variabile e oggetto

variabile
: `l.get(i)` è di tipo `Object`

oggetto
: l'oggetto il cui indirizzo sta in `l.get(i)` è un `Point`

Regola: il controllo statico assume che in una variabile `Object` ci sia un `Object`

Più preciso: in una variabile `Object` potrebbe esserci un `Object`

`Object` non ha la componente `x`

È giusto che il compilatore segnali l'errore

Cast

Dato che l'oggetto è di tipo `Point`, posso fare il cast a `Point`

```
Point e=(Point) l.get(i);
```

Ora posso fare `e.x` ecc.

Soluzione corretta

Anche se una variabile `o` di tipo `Object` contiene l'indirizzo di un `Point`, per poter usare le componenti `x` ed `y` devo fare il cast

```
static void inserisciPuntoOrdine(
  LinkedList l, Point p) {

  int i;
  Point e;

  for(i=0; i<l.size(); i++) {
    e=(Point) l.get(i);
    if(e.x>=p.x) {
      l.add(i, p);
      return;
    }
  }

  l.add(p);
}
```

Metodi non ridefiniti

Questa soluzione non funziona:

```
int x=(int) (l.get(i).getX());
```

È vero che i metodi sono quelli dell'oggetto

È vero che `Point` ha il metodo `getX()`

Però vale solo per i metodi che sono *ridefiniti*

`Object` non ha il metodo `getX()`

Non posso fare `o.getX()`, anche se poi `o` contiene l'indirizzo di un `Point`

Metodi e componenti

Variabile e oggetto della stessa classe: metodi e componenti della classe

Variabile di un tipo e oggetto di una sottoclasse:

non ridefiniti
: regola solita: ogni variabile accede ai suoi metodi e componenti

ridefiniti
: le componenti sono quelle della variabile, i metodi dell'oggetto

La regola da ricordare vale solo per componenti e metodi che sono ridefiniti, e soltanto quando la varibile e l'oggetto sono di tipo diverso

---

Caso tipico

La variabile è un `Object`, ma l'oggetto è di un altro tipo:

- non ha componenti (devo fare il cast, se mi servono le componenti dell'oggetto)
- ha i metodi `equals` e `toString`: se sono ridefiniti, vengono invocati quelli dell'oggetto

Se una variabile è `Object`, si possono solo invocare i metodi di `Object`, e non i metodi dell'oggetto (es `move`)

Programma di prova

Creare una lista che contiene i punti di coordinate `(0,0)`, ..., `(9,9)`

Inserire il punto di coordinate `(2,4)`

Soluzione

Faccio un ciclo

A ogni passo, creo un punto e lo aggiungo in fondo alla lista

```
    LinkedList l;
    l=new LinkedList();

    for(int i=0; i<10; i++)
      l.add(new Point(i,i));

    stampaLista(l);
    inserisciPuntoOrdine(l, new Point(2,4));
    stampaLista(l);
  }
```

Per verificare la correttezza: stampare la lista sia prima che dopo

Se la stampate solo dopo, non sapete se è venuto bene oppure no

---

Esercizio

Data una lista, creare quella che ha gli stessi elementi, in ordine inverso

---

Soluzione

Due soluzioni possibili:

1. scansione diretta della lista di partenza: ogni elemento lo aggiungo all'inizio della nuova lista
2. scansione inversa della lista, con aggiunta di elementi in fondo alla nuova lista

---

Soluzione 1

Verifica. All'inizio:

```
vecchia: 1 2 3 4 5
nuova:
```

Prendo il primo elemento, e lo metto all'inizio

```
vecchia: 1 2 3 4 5
nuova:   1
```

Secondo elemento della lista di partenza, all'inizio:

```
vecchia: 1 2 3 4 5
nuova:   2 1
```

Terzo elemento, sempre all'inizio:

```
vecchia: 1 2 3 4 5
nuova:   3 2 1
```

Ecc.

Soluzione 2

Verifica. All'inizio:

```
vecchia: 1 2 3 4 5
nuova:
```

Prendo l'ultimo elemento e lo metto in coda alla nuova lista:

```
vecchia: 1 2 3 4 5
nuova:   5
```

Penultimo elemento in coda:

```
vecchia: 1 2 3 4 5
nuova:   5 4
```

Terz'ultimo elemento in coda:

```
vecchia: 1 2 3 4 5
nuova:   5 4 3
```

Ecc.

Codice della soluzione 1

Inserire all'inizio: il nuovo elemento deve diventare quello di indice `0`

```
  static LinkedList invertiLista(LinkedList l) {
    LinkedList nuova;
    nuova=new LinkedList();

    int i;

    for(i=0; i<l.size(); i++) {
      Object o;
      o=l.get(i);

      nuova.add(0, o);
    }

    return nuova;
  }
```

Esiste anche il metodo `addFirst(Object)`

Non è necessario saperlo:
`add(0, oggetto)` fa la stessa cosa

Codice della soluzione 2

```
static LinkedList invertiLista(LinkedList l) {
  LinkedList nuova;
  nuova=new LinkedList();

  int i;

  for(i=l.size()-1; i>=0; i--) {
    Object o;
    o=l.get(i);

    nuova.add(o);
  }

  return nuova;
}
```

Osservazione

Gli oggetti non sono stati copiati

Quando faccio `nuova.add(o)`, sto mettendo nella lista il riferimento all'oggetto

La lista nuova contiene i riferimenti agli oggetti della vecchia lista, in ordine inverso

Non si può fare la copia di un oggetto arbitrario

```
Object a, b;

a=new Object();

b=a.clone();          // errore
```

Ci torneremo

Copia di elementi

Si può fare solo per oggetti specifici:

```
static LinkedList invertiListaPoint(LinkedList l) {
  LinkedList nuova;
  nuova=new LinkedList();

  int i;

  for(i=l.size()-1; i>=0; i--) {
    Point p;
    p=(Point) l.get(i);

    Point q=(Point) p.clone();

    nuova.add(q);
```

```
    }

    return nuova;
  }
```

Eliminazione elementi

Metodo `remove`

Si passa un indice intero

Toglie dalla lista l'elemento:

```
Prima:      [abcd, efgh, eft]
Istruzione: l.remove(1);
Dopo:       [abcd, eft]
```

Viene eliminato l'elemento che sta in posizione `1`

Esercizio

Scrivere un metodo statico che elimina la prima stringa di lunghezza pari da una lista di stringhe

Lista di stringhe=so che tutti gli elementi sono stringhe
(non esiste il tipo ‘‘lista di stringhe’’, si tratta comunque di una lista di `Object`)

Soluzione

Faccio un ciclo di scansione

Quando trovo una stringa di lunghezza pari, la elimino

Codice della soluzione

Quando ho trovato l'elemento, posso anche uscire dal ciclo

```
  static void eliminaPrimaPari(LinkedList l) {
    int i;
    String s;

    for(i=0; i<l.size(); i++) {
      s=(String) l.get(i);

      if(s.length()%2==0) {
        l.remove(i);
        break;
      }
    }
  }
```

Variante

Eliminare tutte le stringhe di lunghezza pari

Sembra più facile

Soluzione che non funziona

Ciclo di scansione con eliminazione di tutte le stringhe di lunghezza pari

```
  static void eliminaPariSbagliato(LinkedList l) {
    int i;
    String s;

    for(i=0; i<l.size(); i++) {
      s=(String) l.get(i);

      if(s.length()%2==0)
        l.remove(i);
    }
  }
```

Sulla lista:

```
[abcde, efg, ef, efgh, xxx]
```

il risultato è:

```
[abcde, efg, ef, efgh, xxx]
```

La stringa `efgh` non è stata eliminata!

Assunzione del ciclo

Quando ho scritto il ciclo, ho fatto queste assunzioni:

1. le stringhe di lunghezza pari che ho già considerato sono state tutte eliminate dalla lista
2. ad ogni iterazione del ciclo, considero una nuova stringa

In questo caso, l'assunzione 2 è sbagliata!

Cosa è successo

Lista di partenza:

```
Lista: [abcde, efg, ef, efgh, xxx]
Indici: 0      1    2   3     4
```

Quando `i=2` ho `s="ef"`, che è di lunghezza pari

La elimino, e ottengo:

```
Lista: [abcde, efg, efgh, xxx]
Indici: 0      1    2     3
```

Ora, il corpo del ciclo è finito, per cui si incrementa `i` e si passa alla prossima iterazione

Prossima iterazione: `i=3`, per cui considero la stringa `s="xxx"`

La stringa `"efgh"` è stata saltata

L’assunzione che non vale

Non è vero che ogni iterazione considera un nuovo elemento

Ci sono elementi che vengono saltati

Quando elimino un elemento, quello successivo viene ignorato, e si passa subito a quello dopo ancora

---

Altra soluzione

Se ho eliminato un elemento, considero anche il successivo:

```
static void eliminaPariSbagliatoDue(LinkedList l) {
  int i;
  String s;

  for(i=0; i<l.size(); i++) {
    s=(String) l.get(i);

    if(s.length()%2==0) {
      l.remove(i);
      s=(String) l.get(i);

      if(s.length()%2==0)
        l.remove(i);
    }
  }
}
```

Non funziona se la lista contiene tre stringhe di lunghezza pari in sequenza

---

Progettazione

L’algoritmo di partenza è corretto:

```
per ogni elemento della lista
  se e’ di lunghezza pari, eliminalo
```

L’errore è che il ciclo non considera tutti gli elementi

Raffinamento dell'algoritmo

La frase `per ogni elemento della lista` la traduco in un ciclo:

```
elemento=primo elemento

finche' non sono alla fine della lista {
  elimina elemento se e' pari
  passa al prossimo elemento
}
```

L'errore è che `passa al prossimo elemento` non è `i++`

Se ho eliminato un elemento, non serve fare `i++` per passare al prossimo

---

Implementazione

Scrivo lo stesso ciclo di prima

Però `i++` lo eseguo solo se non ho eliminato l'elemento

```
static void eliminaPari(LinkedList l) {
  int i;
  String s;

  i=0;
  while(i<l.size()) {
    s=(String) l.get(i);

    if(s.length()%2==0)
      l.remove(i);
    else
      i++;
  }
}
```

Attenzione alla terminazione!

Quando si scrivono cicli `while`, bisogna verificare che il ciclo prima o poi termini

Come: quando si esegue il corpo del ciclo, deve cambiare qualcosa

Alla fine, la condizione deve diventare falsa

Liste di dati scalari

In una `LinkedList` ci posso oggetti qualsiasi, ma non dati scalari (interi, reali, booleani, ...)

Per ogni tipo scalare, esiste una classe corrispondente

Scalare	Classe
boolean	Boolean
int	Integer
double	Double

Ognuno di questi oggetti contiene un dato scalare

creare un oggetto
`i=new Integer(12);` gli passo il valore da mettere dentro

trovare il valore
`i.intValue()` ritorna il valore intero

Il valore memorizzato nell'oggetto non è modificabile

Esempio: creazione di una lista di interi

Ogni volta che devo inserire un valore, devo creare l'oggetto con dentro il valore

```
import java.util.*;

class ListaInteri {
  public static void main(String args[]) {
    LinkedList l;
    l=new LinkedList();

    int i;
    for(i=0; i<10; i++)
      l.add(new Integer(i));

    System.out.println(l);
  }
}
```

Usare i valori

Scrivere un metodo che somma i valori di una lista di interi

Nota: non si possono sommare gli oggetti di tipo `Integer` con `+`

Soluzione

Dato che so che la lista contiene solo interi, posso fare il cast

```
static int somma(LinkedList l) {
  int i;
  int somma=0;
  Integer v;

  for(i=0; i<l.size(); i++) {
    v=(Integer) l.get(i);
    somma=somma+v.intValue();
  }

  return somma;
}
```

Osservazione: `l.get(i).intValue()` è un errore

`l.get(i)` è di tipo `Object`, che non ha il metodo `intValue()`

Esercizio

Data una lista di interi, per ogni elemento di valore (non indice) pari, eliminare il successivo

---

Soluzione provvisoria

Ciclo: per ogni elemento, se è pari, elimino il successivo

```
static void dopoPari(LinkedList l) {
  int i;
  Integer v;

  for(i=0; i<l.size(); i++) {
    v=(Integer) l.get(i);

    if(v.intValue()%2==0)
      l.remove(i+1);
  }
}
```

Attenzione ai casi limite!

Il metodo va quasi bene

Se però l'ultimo elemento della lista è pari, si cerca di eliminare un elemento che non c'è

Questo genera un errore in esecuzione

Varianti

1. faccio il ciclo solo fino al penultimo

```
for(i=0; i<l.size()-1; i++) {
  v=(Integer) l.get(i);

  if(v.intValue()%2==0)
    l.remove(i+1);
}
```

2. controllo prima di eliminare

```
for(i=0; i<l.size(); i++) {
  v=(Integer) l.get(i);

  if(v.intValue()%2==0)
    if(i+1<l.size())
      l.remove(i+1);
}
```

Caso limite sul primo elemento

Se andava eliminato l'elemento precedente a quello pari, bisognava stare attenti al primo elemento

---

Elementi consecutivi

Quando viene enunciata una condizione sugli elementi di una lista, tenere presente che la lista può:

- non contenere elementi che soddisfano la condizione
- avere un elemento in prima o ultima posizione che soddisfa la condizione
- contenere una sequenza di elementi che soddisfano la condizione

Nel nostro caso: se la lista contiene una sequenza di elementi pari consecutivi, vanno eliminati tutti tranne il primo, più il primo dispari successivo

---

Soluzione difficile

Per ogni elemento:
se è pari, memorizzo il successivo, lo elimino, e ripeto

Nel ciclo interno, devo controllare se sono arrivato alla fine della lista

```
static void dopoPariOK(LinkedList l) {
  int i;
  Integer v;

  i=0;
  while(i<l.size()-1) {
    v=(Integer) l.get(i);

    while((i<l.size()-1)&&(v.intValue()%2==0)) {
      v=(Integer) l.get(i+1);
      l.remove(i+1);
    }

    i++;
  }
}
```

Nella condizione `i<l.size()-1`, del while interno, non è `i` che cambia, ma `l.size()`

Soluzione facile

Riformulazione del problema: per ogni elemento, se il *precedente* è pari, elimino l'elemento

Facendo la scansione della lista in ordine inverso, non devo mai tenere conto di elementi già eliminati

```
static void dopoPari(LinkedList l) {
  int i;
  Integer v;

  for(i=l.size()-1; i>=1; i--) {
    v=(Integer) l.get(i-1);
    if(v.intValue()%2==0)
      l.remove(i);
  }
}
```

Altra soluzione

Creare una nuova lista senza gli elementi che vanno eliminati

---

Esercizio facile

Scrivere un metodo statico che conta quante volte un oggetto appare in una lista

```
static int conta(LinkedList l, Object o) {
  ...
}
```

Soluzione

Uso un contatore: ogni volta che trovo un elemento che è `equals` a `o`, lo incremento di uno

```
static int conta(LinkedList l, Object o) {
  int quante=0;
  int i;

  for(i=0; i<l.size(); i++)
    if(o.equals(l.get(i)))
      quante++;

  return quante;
}
```

Esercizio più difficile

Data una lista, verificare se c'è qualche elemento che si ripete più di una volta

```
static boolean ripetizioni(LinkedList l) {
  ...
}
```

Come si risolve:

1. dare la soluzione a parole
2. tradurla in codice

Soluzione a parole

```
per ogni elemento della lista
  se appare piu' di una volta
    ritorna true

ritorna false
```

A questo punto, si tratta di tradurre ogni parte di questo algoritmo

---

Per ogni elemento

È chiaramente un ciclo:

```
for(i=0; i<l.size(); i++)
  se l.get(i) c'e' piu' di una volta
    ritorna true;

ritorna false;
```

I due `ritorna` sono chiaramente dei `return`

Manca solo da tradurre: `sta piu' di una volta nella lista`

Più di una volta?

Posso risolvere il problema cosí:

```
conta quante volte c'e' l'elemento
                      [nella lista
vedi se e' >1
```

Vantaggio:

1. so già come è fatto il metodo per contare
2. il resto è facile

Scrivo il metodo per contare, anche se poi mi serve solo per vedere se un elemento è ripetuto!

Il metodo fa qualcosa in più: va bene lo stesso

Soluzione complessiva

Serve anche il metodo `conta`

```
static int conta(LinkedList l, Object o) {
  int quante=0;
  int i;

  for(i=0; i<l.size(); i++)
    if(o.equals(l.get(i)))
      quante++;

  return quante;
}

static boolean ripetizioni(LinkedList l) {
  int i;

  for(i=0; i<l.size(); i++)
    if(conta(l, l.get(i))>1)
      return true;

  return false;
}
```

Nota sui requisiti

Quando un problema richiede: ‘‘scrivere un metodo che...’’

Si intende: ‘‘ed, eventualmente, tutti i metodi ausiliari necessari’’

Scrivere metodi ausiliari ha due vantaggi:

1. spesso, sono uguali/simili a metodi visti a lezione
2. il codice del metodo da scrivere risulta più semplice

Semplicità del codice

Se il codice è più semplice, è più facile non commettere errori

Codice senza metodo ausiliario:

```
static boolean
    ripetizioniNoAux(LinkedList l) {
  int i, j;
  int conta;

  for(i=0; i<l.size(); i++) {
    conta=0;

    for(j=0; j<l.size(); j++)
      if(l.get(i).equals(l.get(j)))
        conta++;

    if(conta>1)
      return true;
```

```
    }

    return false;
  }
```

Nei cicli nidificati (l'uno dentro l'altro), è facile sbagliare

Esempio:

- usare `i` al posto di `j`,
- mettere `conta=0` fuori da tutti e due i cicli,
- mettere `return true` all'interno di tutti e due i cicli
- ecc.

A volte i cicli nidificati servono

Meglio fare due metodi, se la cosa che fa il ciclo interno si può realizzare con un metodo

---

Unico metodo con doppio ciclo

Altra soluzione

```
per ogni elemento
  se si trova anche in un'altra posizione
    return true;

return false;
```

Implementazione

In questo modo, non funziona:

```
  static boolean ripetizioni(LinkedList l) {
    int i, j;

    for(i=0; i<l.size(); i++)
      if(l.contains(l.get(i)))
        return true;

    return false;
  }
```

Motivo: `l.get(i)` si trova nella lista

Se faccio `l.contains(l.get(i))` ritorna per forza `true`

Altra implementazione

Dato che `l.contains` non va bene, faccio cosí:

> per ogni elemento della lista, faccio un ciclo che, per ogni altro elemento della lista, verifica l'uguaglianza

```
static boolean ripetizioni(LinkedList l) {
  int i, j;

  for(i=0; i<l.size(); i++)
    for(j=0; j<l.size(); j++)
      if(l.get(i).equals(l.get(j)))
        return true;

  return false;
}
```

Non funziona nemmeno questo!

Caso `i=j`: qui `l.get(i).equals(l.get(j))` è sicuramente `true`

Soluzione numero 1

Stesso ciclo, ma controllo che sia `i!=j`

Ossia, la lista contiene due elementi uguali se ci sono due indici *diversi* `i` e `j` che contengono due elementi uguali

```
static boolean ripetizioni(LinkedList l) {
  int i, j;

  for(i=0; i<l.size(); i++)
    for(j=0; j<l.size(); j++)
      if((i!=j) && (l.get(i).equals(l.get(j))))
        return true;

  return false;
}
```

Soluzione numero 2

Per ogni elemento, verifico solo se c'è un elemento uguale nelle posizioni successive

Perchè funziona?

Se sono arrivato all'elemento 4 vuol dire che non ci sono ripetizioni, fino a questo momento

Quindi, non possono esserci elementi prima uguali a questo

Quindi, devo guardare solo gli elementi successivi

```
static boolean ripetizioni(LinkedList l) {
  int i, j;

  for(i=0; i<l.size(); i++)
    for(j=i+1; j<l.size(); j++)
      if(l.get(i).equals(l.get(j)))
        return true;

  return false;
}
```

Nota sui cicli nidificati

Quando ci sono due cicli:

```
for(i=0; i<n, i++) {

  for(j=0; j<m; j++) {
    istruzioni;
  }

}
```

Le istruzioni più interne vengono eseguite `n*m` volte

Dopo aver scritto un programma con due cicli, l'uno dentro l'altro, verificate se effettivamente bisognava fare `n*m` iterazioni

Per vedere se ci sono elementi ripetuti: è necessario. Infatti, per ogni elemento devo fare la scansione di tutta la lista per vedere se c'è un elemento uguale

---

Differenza liste-array

Nelle liste si possono eliminare/aggiungere elementi in mezzo

L'accesso agli elementi di un array è più efficiente (non richiede di guardare gli oggetti precedenti della catena)