



Sequenze di lunghezza variabile

ArrayList
Operazioni
Ricerche

Vettori a lunghezza variabile

- Abbiamo visto come fare per definire una sequenza di variabili tutte dello stesso tipo tramite gli array
- Una forte limitazione degli array è che quando vengono creati bisogna specificare la loro lunghezza e, dopo la creazione, la dimensione non può essere più cambiata
- In molti casi è molto difficile stimare la dimensione necessaria al momento della creazione dell'array

Vettori a lunghezza variabile

- La libreria standard di Java ci viene in aiuto fornendo classi che definiscono array evoluti
- Due esempi sono `java.util.Vector` e `java.util.ArrayList`
- Le due classi sono uguali tranne che per una caratteristica legata alla concorrenza
- I vettori di `Vector` sono protetti da accessi simultanei, potenzialmente dannosi, da parte di due o più thread (sottoprocessi della JVM) mentre `ArrayList` non ha questo controllo

Vettori a lunghezza variabile

- Se non si gestiscono thread nella nostra applicazione possiamo tranquillamente usare **ArrayList**
- In tutti gli altri casi è meglio usare **Vector**
- Si noti, in particolare, che in ogni applicazione che usa un'interfaccia grafica ci sono almeno due thread: il main e il gestore dell'interfaccia
- Pertanto è bene usare **Vector** se si usa l'interfaccia grafica

ArrayList

- Illustriamo **ArrayList** sapendo che comunque le stesse considerazioni valgono per **Vector**
- Un oggetto di **ArrayList** rappresenta una **lista sequenziale di oggetti generici**
- Gli elementi della sequenza sono numerati a partire dal primo
- Come per gli array la numerazione parte da 0

ArrayList

- Si può accedere in maniera indipendente ad ogni elemento dell'**ArrayList** tramite un indice intero, come per gli array
- Si può sempre inserire un nuovo elemento in fondo a un **ArrayList** tramite il metodo **add**
- Non c'è limite (a parte la disponibilità di memoria fisica) alla dimensione che un'**ArrayList** può raggiungere
- L'implementazione della classe fa in modo di allocare sempre nuovo spazio quando ce n'è bisogno

Purse e Coin

- Consideriamo di nuovo la classe **Purse**
- L'implementazione che abbiamo dato, ogni volta che viene inserita una nuova moneta, semplicemente incrementa di uno un contatore per il tipo di moneta inserita
- Sarebbe più realistico se il borsellino potesse memorizzare proprio le singole monete
- Utilizzeremo un'**ArrayList** per migliorare **Purse** e permettergli di memorizzare un numero, non specificato a priori, di oggetti della classe **Coin**

Operazioni su ArrayList

- Vediamo come si crea un oggetto della classe **ArrayList** e come si aggiungono elementi

```
ArrayList coins = new ArrayList();  
coins.add(new Coin(0.1, "Dime"));  
coins.add(new Coin(0.25,  
                    "Quarter"));  
...
```

Accesso agli elementi

- Per accedere ad un elemento di un **ArrayList** `a` si usa il metodo `a.get(i)` passando come parametro la posizione `i` dell'oggetto richiesto (è l'analogo di `a[i]` per gli array)
- La posizione deve essere tra 0 e la dimensione `a.size() - 1` (in caso contrario viene sollevata un'eccezione)
- Il metodo `size()` è l'analogo del campo `length` degli array

Tipo degli elementi

- Gli elementi sono tutti di tipo **Object** generico
- È possibile inserire in un **ArrayList** anche elementi non omogenei (ma in genere non è una buona pratica di programmazione)
- È necessario fare un cast esplicito per poter utilizzare tutte le funzionalità di un oggetto recuperato dall'**ArrayList** (che altrimenti può essere usato solo come un generico elemento della classe **Object** – vedi API)

Recupero di un elemento

```
Coin aCoin = (Coin) coins.get(0);
```

- `coins.get(0)` restituisce un riferimento ad un `Object` che sarebbe il primo elemento di `coins`
- Sappiamo che questo `Object` in realtà è un oggetto della classe `Coin` e facciamo un cast esplicito per riottenere la possibilità di chiamare i metodi della classe `Coin` sull'elemento recuperato

Accesso agli elementi

- Vediamo una tipica attività molto comune cioè l'esame in sequenza di tutti gli elementi di un **ArrayList**
- Lo schema di programma che si usa è esattamente analogo a quello usato per un array, adattando solo la notazione

Accesso in sequenza

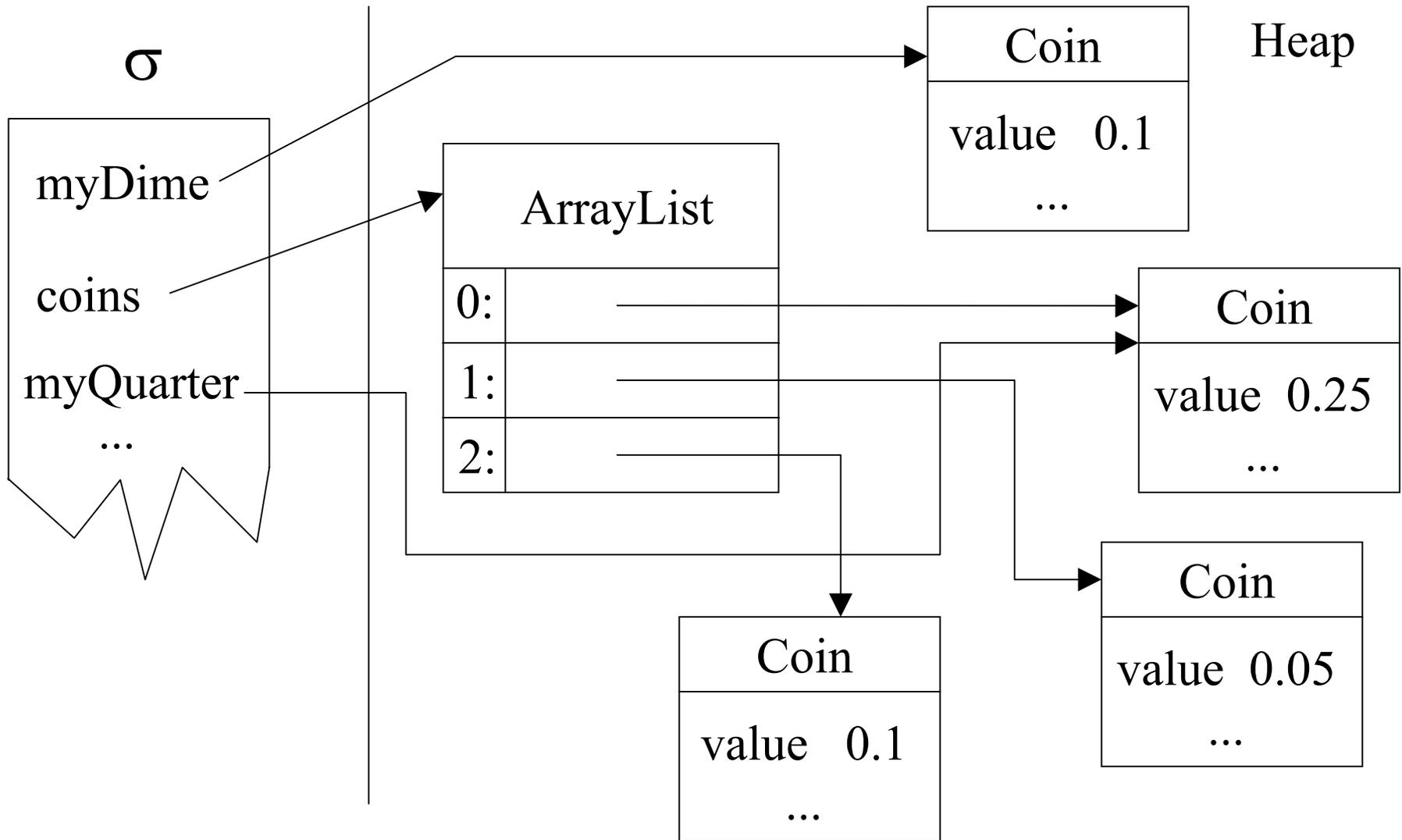
- Calcola il valore totale di tutte le monete attualmente presenti nell'**ArrayList coins**:

```
double total = 0;
for (int i = 0; i < coins.size();
     i++) {
    // Recupero con cast
    Coin aCoin = (Coin) coins.get(i);
    total = total + aCoin.getValue();
}
```

Inserzione

- Il metodo dalla segnatura `void add(Object)` aggiunge un elemento in fondo all'`ArrayList`
- Il metodo dalla segnatura `void add(int, Object)` aggiunge un elemento nella posizione specificata dal primo parametro e fa scorrere l'elemento attualmente in quella posizione e tutti gli eventuali elementi che seguono di una posizione verso destra

Inserzione



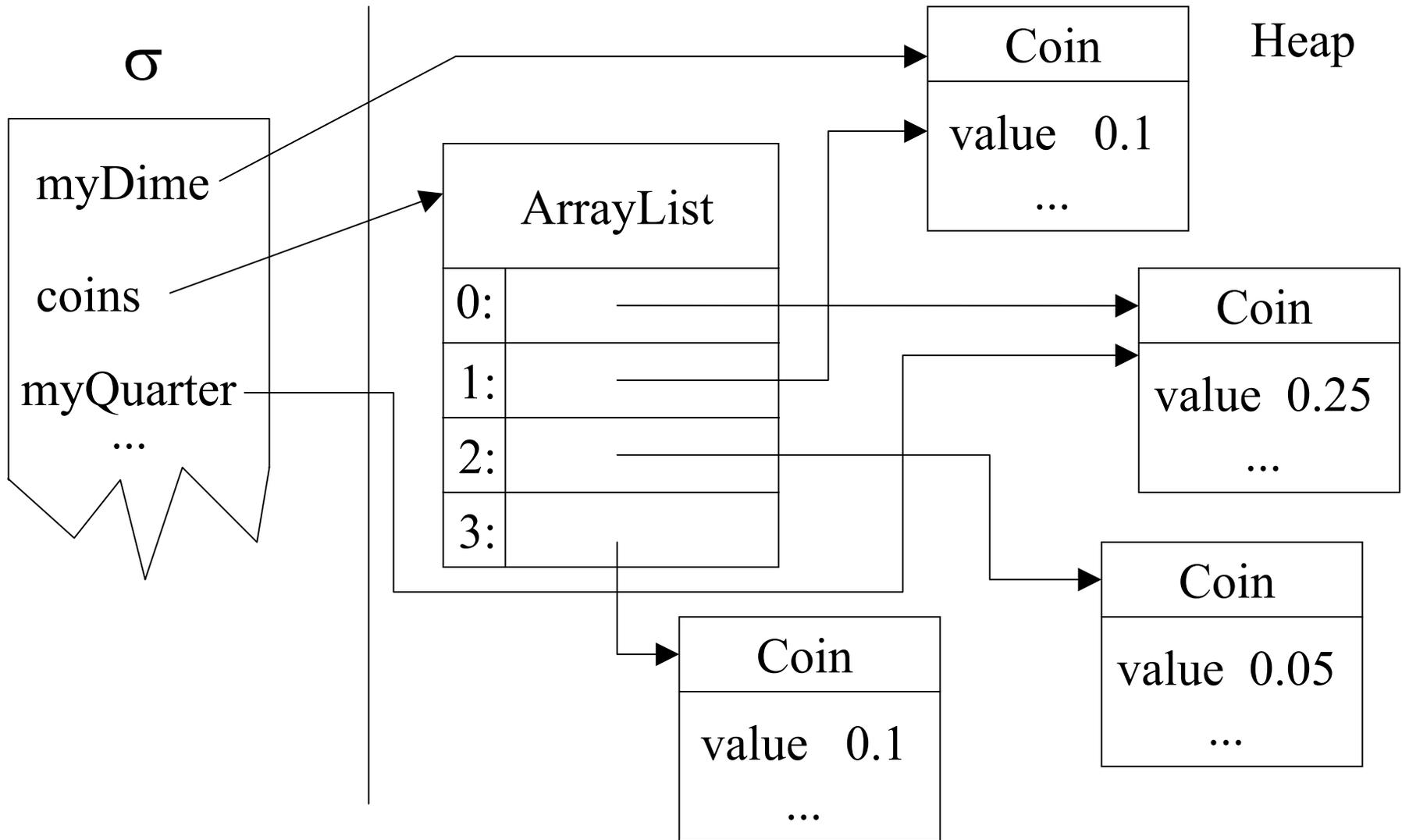
Inserzione

- Supponiamo di eseguire, nello stato appena visto, questa aggiunta:

```
coins.add(1, myDime);
```

- Il risultato è il seguente stato:

Inserzione

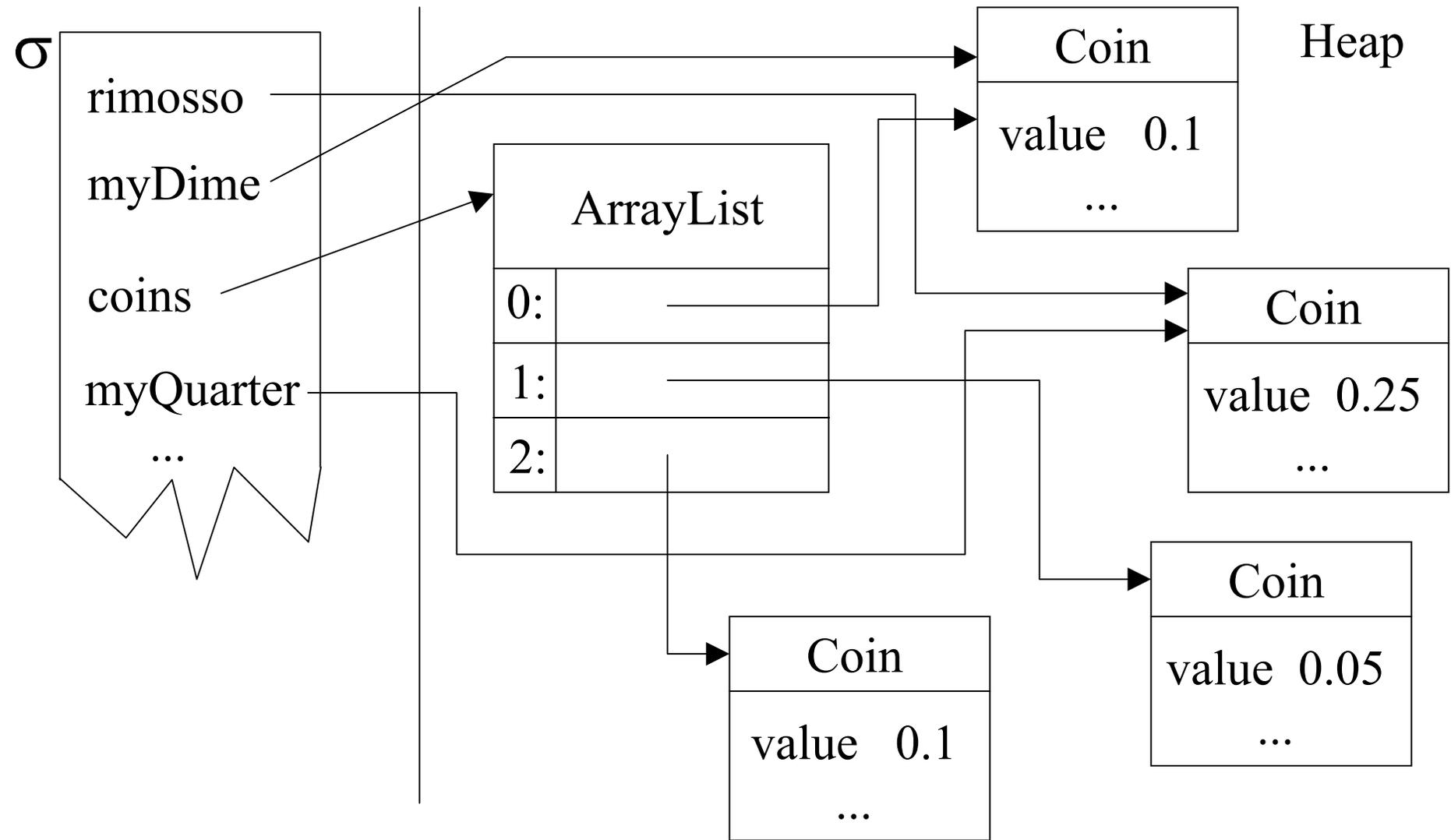


Rimozione

- È possibile rimuovere un elemento in una data posizione con il metodo `Object remove(int)`
- Tutti gli elementi che seguono, se ce ne sono, vengono spostati di una posizione verso sinistra
- Il metodo restituisce un riferimento all'oggetto rimosso
- Ad esempio supponiamo di eseguire la seguente istruzione a partire dallo stato appena disegnato:

```
Object rimosso = coins.remove(0);
```

Rimozione



Assegnamento

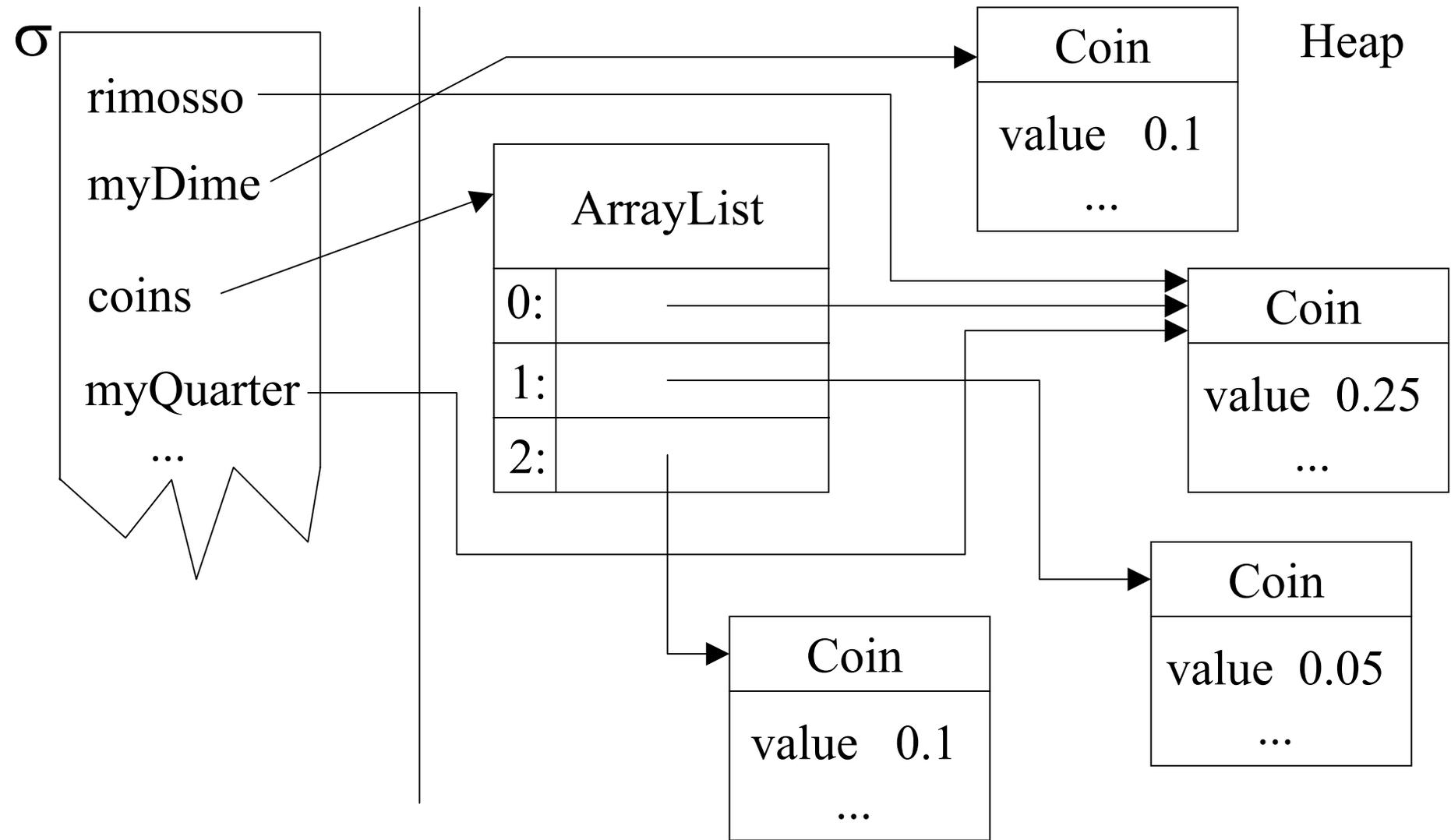
- Per cambiare un elemento in una certa posizione con un altro si usa il metodo **void set(int, Object)**

- L'istruzione seguente

```
coins.set(0, myQuarter);
```

eseguita nello stato precedente porta nel seguente stato (che è esattamente la situazione da cui eravamo partiti)

Assegnamento



Ricerca

- La classe **ArrayList** mette a disposizione due metodi per la ricerca di un oggetto in un **ArrayList**
- Il metodo **int indexOf(Object)** cerca il primo elemento dell'**ArrayList** su cui è chiamato che risulta uguale all'**Object** passato come parametro tramite il metodo **equals()**
- Se il risultato è **-1** allora l'oggetto non è presente
- Altrimenti il risultato è la posizione dell'oggetto

Ricerca

- Il metodo `int lastIndexOf (Object)` si comporta come `indexOf`, ma riporta l'ultima posizione dell'oggetto nell'`ArrayList` o `-1`
- Il criterio di ricerca, più precisamente, è il seguente:
- Sia `o` il riferimento all'oggetto passato come parametro
- Viene cercato l'elemento `e` dell'`ArrayList` per cui `o.equals (e)` è vero

- `equals` è un metodo polimorfico
- Il codice che viene effettivamente eseguito dipende dal tipo effettivo dell'oggetto riferito da `o`
- Se la classe a cui `o` appartiene non fornisce un metodo `equals` allora viene usato quello della classe `Object`
- Il metodo `equals` della classe `Object` confronta i riferimenti: due oggetti sono uguali se sono fisicamente lo stesso oggetto allocato, cioè hanno lo stesso riferimento

Ricerca

- Non abbiamo implementato il metodo `equals` per la classe `Coin` quindi verrà usato quello di `Object` nella seguente ricerca:

```
int myQuarterPosition =  
        coins.indexOf(myQuarter) ;
```

- Nello stato in cui eravamo arrivati questa ricerca restituisce l'indice 0, cioè la posizione dell'oggetto puntato da `myQuarter` in `coins`

Ricerca

- Tuttavia la seguente ricerca restituisce -1:

```
int risultato = coins.indexOf(  
    new Coin(0.25, "Quarter"));
```
- L'oggetto non viene trovato perché si usa il metodo `equals` della classe `Object`
- L'oggetto passato come parametro è un oggetto nuovo appena creato e quindi il suo riferimento è diverso da tutti i riferimenti presenti in `coins`

Ricerca

- Possiamo pensare di definire un metodo `equals` per `Coin` in modo che identifichi due monete se hanno lo stesso nome e lo stesso valore
- In questo modo la ricerca effettuata dai metodi `indexOf` e `lastIndexOf` avrà esito positivo se viene trovata una moneta in `coins` che ha lo stesso valore e lo stesso nome di quella passata come parametro

equals

- Il metodo `equals` viene chiamato da `indexOf` o da `lastIndexOf` sull'`Object` che viene loro passato
- Se passiamo un oggetto di tipo `Coin` allora sappiamo che il polimorfismo farà in modo che il metodo `equals` eseguito sia quello della classe `Coin`
- Il metodo `equals` viene chiamato passando come parametri i riferimenti degli elementi dell'`ArrayList` che sono di tipo `Object`

equals

- Se definissimo il metodo con la segnatura **boolean equals(Coin)** non sarebbe questo ad essere eseguito, ma sempre quello di **Object** poiché i riferimenti passati come **parametri** sono di tipo **Object** (in questo caso il polimorfismo non si applica)
- Per poter effettuare la ricerca in questo modo dobbiamo quindi definire un metodo con segnatura **boolean equals(Object)** in **Coin**

equals

```
public boolean equals(Object o) {
    if ( o instanceof Coin ) {
        Coin aCoin = (Coin) o;
        if ( this.value == aCoin.value &&
            this.name.equals(aCoin.name) )
            return true;
        else
            return false;
    }
    else
        return false;
}
```

equals

- Possiamo comunque sempre definire un metodo `equals` anche per confronti tra oggetti entrambi **formalmente** di tipo `Coin`:

```
public boolean equals(Coin aCoin) {  
    if ( this.value == aCoin.value &&  
        this.name.equals(aCoin.name) )  
        return true;  
    else  
        return false;  
}
```

Ricerca

- Adesso si avrà che

```
int risultato = coins.indexOf(  
    new Coin(0.25, "Quarter"));
```

- Restituisce la posizione 0, come volevamo

Ricerca

- Non sempre però è conveniente usare i metodi `indexOf` o `lastIndexOf` per cercare un elemento in un `ArrayList`
- Questo tipo di ricerca presenta due inconvenienti:
 1. Bisogna definire il metodo `equals` nel modo che abbiamo visto
 2. Non serve se vogliamo fare una ricerca in base a criteri diversi dall'uguaglianza implementata in `equals`

Ricerca

- In generale una ricerca all'interno di un **ArrayList** può essere implementata nello stesso modo che abbiamo visto per gli array, cioè attraverso gli schemi di ricerca lineare certa o incerta
- Usando questi schemi non siamo costretti a usare per forza un criterio di uguaglianza: possiamo cercare, abbiamo visto, un elemento che soddisfa una proprietà P di qualsiasi tipo

Ricerca

- Vediamo ad esempio una ricerca lineare incerta, in `coins`, della prima moneta che ha un certo valore dato `v`:

```
int i = 0;
boolean trovato = false;
while (i < coins.size() && !trovato) {
    Coin c = (Coin) coins.get(i);
    if (c.getValue() == v )
        trovato = true;
    else i++;
}
```

Ricerca

- Se eseguiamo la ricerca nello stato a cui eravamo pervenuti negli esempi precedenti con $v = 0.1$ il risultato sarà che viene trovata una moneta con valore v in posizione 2 di `coins`

Conteggio

- Vediamo un frammento di codice che serve a contare quante volte un moneta di un certo tipo occorre in `coins`
- Possiamo in questo caso usare il metodo `equals(Coin)` che abbiamo implementato
- Creiamo dapprima un oggetto `Coin` che ci servirà a fare il confronto:

```
Coin modello = new Coin(0.25,  
                        "Quarter");
```

Conteggio

- Poi contiamo le occorrenze di monete che eguagliano il modello secondo la nostra definizione di uguaglianza:

```
int contatore = 0;
for (int i = 0; i < coins.size(); i++) {
    Coin aCoin = (Coin) coins.get(i);
    if (aCoin.equals(modello))
        contatore++;
}
```

- Viene eseguito il metodo con segnatura **boolean equals (Coin)** della classe **Coin**

ArrayList e tipi di base

- Ma come possiamo fare se vogliamo creare un oggetto **ArrayList** contenente interi, booleani, double o altri tipi base?
- Gli elementi di un oggetto **ArrayList** possono essere solo oggetti!
- Ci vengono in aiuto le classi involucro
- A partire da un valore di un tipo base possiamo costruire un oggetto della classe involucro corrispondente contenente il valore

ArrayList e tipi base

- Ad esempio creiamo un **ArrayList** di interi in questo modo:

```
ArrayList interi = new ArrayList();
```

```
interi.add(Integer(0));
```

```
interi.add(Integer(-9));
```

```
interi.add(Integer(15));
```

- Per il recupero:

```
Integer anInt = (Integer) interi.get(1);
```

```
int v = anInt.intValue(); // vale -9
```

Esercizio

- Eseguire le modifiche viste alla classe **Purse**
- In particolare
 - Trasformare i metodi **addNickel** **addDime** e **addQuarter** nel metodo **add(Coin)**
 - cambiare gli altri metodi di conseguenza
 - Aggiungere un metodo **getNumberOf(Coin c)** che dia il numero di monete uguali a **c** attualmente presenti nel borsellino
 - Aggiungere un metodo **toString()** che stampi in sequenza il nome di tutte le monete presenti nel borsellino secondo il loro ordine cronologico di inserimento

Esercizio

- Aggiungere alla classe **Purse** un metodo **reverse ()** che inverta la sequenza di monete in un borsellino
- Ad esempio se il risultato di **toString ()** dell'esercizio precedente è
[Dime, Nickel, Dime, Quarter]
- Dopo la chiamata di **reverse** dovrà essere:
[Quarter, Dime, Nickel, Dime]

Esercizio

- Aggiungere alla classe **Purse** un metodo **public void transfer(Purse other)** che trasferisce in questo (**this**) borsellino il contenuto di un altro borsellino (**other**)
- Ad esempio se un borsellino **a** contiene [Dime, Nickel, Quarter] e un altro borsellino **b** contiene [Dime, Quarter, Quarter], dopo la chiamata **a.transfer(b)** ; **b** diventa vuoto e **a** contiene [Dime, Nickel, Quarter, Dime, Quarter, Quarter]

Esercizio

- Scrivere per la classe `Purse` un metodo `public boolean equals(Object other)` che controlli se i due borsellini contengono le stesse monete nello stesso ordine

Esercizio

- Scrivere per la classe `Purse` un altro metodo `public boolean equals(Object other)` che controlli se i due borsellini hanno le stesse monete in **qualsiasi** ordine
- Ad esempio i borsellini `[Nickel, Dime, Dime, Quarter]` e `[Quarter, Dime, Nickel, Dime]` devono risultare uguali

Esercizio

- Realizzare una classe **Bank** che gestisce un certo numero di conti
- Metodi:
 - `public void addAccount(double initialBalance)`
 - `public void deposit(int account, double amount)`
 - `public void withdraw(int account, double amount)`
 - `public double getBalance(int account)`
- Suggerimento: far coincidere i numeri di conto con gli indici di un **ArrayList**

Esercizio

- Scrivere un programma che produca permutazioni casuali dei numeri da 1 a 10
- Per ogni permutazione casuale si deve riempire un array con i numeri da 1 a 10 in modo che non ve ne siano due uguali
- Realizzare un metodo

```
int[] nextPermutation()
```

che restituisce la prossima permutazione

- Continua.....

Esercizio cont'd

- Per realizzare la permutazione casuale basta riempire un array di 10 posizioni con i numeri da 1 a 10 in ordine
- poi generare un numero casuale i da 0 a 9
- rimuovere dall'array il numero nella posizione i
- aggiungerlo all'array permutazione che si sta costruendo
- Ripetere fino a quando il primo array non si è esaurito

Esercizio

- Scrivere un programma che verifichi se una tabella di numeri $n \times n$ sia un quadrato magico
- Un quadrato magico è tale che contiene tutti numeri diversi e la somma di tutte le righe, di tutte le colonne e delle due diagonali è sempre lo stesso numero
- Prendere in input la tabella sotto forma di $n \times n$ numeri interi

Esercizio

- Scrivere un programma che crea quadrati magici con il seguente algoritmo:
 - Si scelga la dimensione del lato n tale che n sia dispari
 - Mettere il numero iniziale (es 1) al centro dell'ultima riga
 - Sia k l'ultimo numero inserito in una posizione (i,j)
 - Mettere il numero $k+1$ nella casella in basso a destra rispetto a quella di k andando a capo sui bordi
- Continua....

Esercizio cont'd

- Se la casella in basso a destra è già occupata o ci si trova nell'angolo in basso a destra allora scrivere $k+1$ nella casella superiore alla posizione corrente
- Andare avanti incrementando k fino al completamento di tutta la tabella. Esempio:

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15