QUEUE : considerazioni

- Si è realizzata una struttura dati complessa utilizzandone una primitiva, l'array.
- Il pregio di tale implementazione è il *basso costo computazionale*, mentre il punto critico riguarda l'*uso della memoria*.
- Una sequenza di `enqueue()` può riempire la coda, mentre una sequenza di `dequeue()` può lasciare un numero elevato di posizioni vuote.
- Quindi nasce la necessità di ridimensionare opportunamente l'array.

QUEUE : considerazioni

- Considerazioni analoghe valgono anche per le pile, ma l'uso della classe `Vector` ha reso "trasparente" questa problematica.
- Vediamo un semplice esempio di uso delle code, che mette in evidenza il problema: si riempie una coda con valori interi e poi li si stampa nello stesso ordine con cui si sono inseriti.

QUEUE : esempio

```
ArrayQueue qu = new ArrayQueue(10);
System.out.print("enqueue: ");
for(int i=0;i<8 ;i++){    //for(int i=0;i<18 ;i++){
    qu.enqueue(new Integer(i));
    System.out.print(i+" ");
}
System.out.print("\ndequeue: ");
for(int i=0;!qu.isEmpty();i++)
    System.out.print(qu.dequeue()+" ");
```

```
enqueue: 0 1 2 3 4 5 6 7
dequeue: 0 1 2 3 4 5 6 7
```


QUEUE : esempio

- Nell'ultimo ciclo `for` l'uso di `!qu.isEmpty()` impedisce di estrarre un elemento da una coda vuota (nel qual caso verrebbe lanciata un'eccezione).
- Il fatto grave è che se la coda è piena i valori precedentemente inseriti vengono persi.
- Se si sostituisce il primo ciclo `for` con quello commentato, si ottiene il seguente output:

```
enqueue: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
dequeue: 10 11 12 13 14 15 16 17
```

QUEUE : gestione memoria

- Vediamo come si può modificare l'implementazione della coda in modo tale da ridimensionare opportunamente l'array.
- Poiché una sequenza di `enqueue()` può riempire la coda, mentre una sequenza di `dequeue()` può lasciare un numero elevato di posizioni vuote, questi sono i metodi che devono essere modificati.

QUEUE : gestione memoria

```
public void enqueue(Object el) {
  if (isFull())
     doubleQueue();
   …
protected void doubleQueue(){
  Object[] newdata = new Object[2*size];

  for(int i=first, j=0; j<size; j++, i++,i=i%size){
          newdata[j] = data[i];
  }
  first=0;
  last=size-1;
  size=2*size;
  data=newdata;
}
```


QUEUE : gestione memoria

```
public Object dequeue() {
    int elem;
    if (first<=last)
          elem=last-first+1;
    else
          elem=size-first+last+1;
    if (( elem<size/4) && (elem>=s_size/4))
          halfQueue(elem);

    …
```


QUEUE : gestione memoria

- Il nuovo campo `s_size` memorizza la dimensione iniziale della coda, quindi è inizializzato nel costruttore.
- Il controllo su `elem<size/4` serve per evitare di invocare un dimezzamento della dimensione dell'array non appena si è eseguito un raddoppio della sua dimensione.
- Il controllo su `elem>=s_size/4` assicura di non scendere mai sotto la dimensione iniziale della coda.

QUEUE : gestione memoria

```
protected void halfQueue(int elem){
  Object[] newdata = new Object[size/2];

  for(int i=first, j=0; j<elem; j++,
                           i++,i=i%size ){
          newdata[j] = data[i];
     }
     first=0;
     last=elem-1;
     size=size/2;
     data=newdata;
  }
```


QUEUE : gestione memoria

- Vediamo come si comporta la nuova implementazione della coda nell’esempio precedente:

Avviene un raddoppio della coda: l’array iniziale è sostituito con uno di dimensione doppia.

enqueue: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
dequeue: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Avviene un dimezzamento della coda: l’array corrente è sostituito con uno di dimensione dimezzata.

QUEUE : considerazioni

- L’uso della struttura dati primitiva array, come rappresentazione interna dei dati della coda, evidenzia due aspetti:
  - La semplicità ed efficienza dell’uso degli indici.
  - Il costo che si paga nella gestione della memoria per ottenere un “array ridimensionabile”.

- Inoltre è evidente il vantaggio di usare una struttura dati astratta (ADT), che nasconde i problemi dell’implementazione all’utilizzatore.

QUEUE : considerazioni

- Considerando gli aspetti legati alle prestazioni, l’implementazione esplicita della coda ha reso evidente il costo computazionale di avere un array ridimensionabile.

- Nel caso peggiore entrambi i metodi, `enqueue()` e `dequeue()`, hanno un costo lineare $O(n)$. Vi è un costo per aumentare la dimensione dell’array, ma anche uno per diminuirla.

PRIORITY QUEUE

- Nelle *code prioritarie* gli elementi vengono estratti in base alla loro priorità e alla loro attuale posizione in coda.
- Per esempio, in una coda di processi può essere necessario che per il corretto funzionamento del sistema il processo P_2 venga eseguito prima del processo P_1, anche se P_1 è stato inserito per primo nella coda.
- Il problema di una coda prioritaria è trovare realizzazioni efficienti che consentano accodamenti ed estrazioni veloci.

PRIORITY QUEUE

- Vi sono diverse soluzioni: in generale, c'è la necessità di mantenere due strutture dati (per es., una ordinata secondo l'ordine di ingresso nella coda e l'altra ordinata secondo la priorità).

- Dal punto di vista della complessità computazionale si ottengono prestazioni da $O(n)$ a $O(n^{1/2})$ in relazione alle diverse implementazioni.

INTRODUZIONE

- Una *collezione* (contenitore) è un oggetto che raggruppa più dati in una singola struttura. L'array è un esempio di collezione. Sono collezioni anche le code e le pile.

- Le collezioni sono usate per memorizzare, leggere, manipolare dati strutturati e per trasmettere tali dati da un metodo ad un altro.

SOMMARIO

- Prima di studiare nuove strutture dati è necessario descrivere alcuni argomenti:

  - Metodo `equals()`.
  - Tipi run-time: `instaceof`.
  - Eccezioni.
  - Iteratori.
  - Classi interne.
  - Nodi: elementi più riferimenti.

METODO EQUALS

- Poiché ogni classe Java è una discendente della classe `Object`, vengono ereditati i metodi di `Object`. Tuttavia per produrre un corretto comportamento tali metodi devono essere sovrascritti (*overridden*) per essere adattati alla specifica classe.
- Un esempio è il metodo `toString()`, che permette di stampare una descrizione dell'oggetto.
- Un importante metodo ereditato è `equals()`: confronta due oggetti e se sono uguali (contengono la stessa informazione) ritorna `true`, altrimenti `false`.

METODO EQUALS

```
public class Libro {
      int ISBN;
      public Libro(int i){
            ISBN=i;
      }
      public boolean equals(Object obj) {
              if (((Libro)obj).ISBN==this.ISBN)
                  return true;
              else
                  return false;
}}
```

```
public static void main(String[] args) {
      Libro b1 = new Libro(1234567890);
      Libro b2 = new Libro(1234567890);
      if (b1.equals(b2))
            System.out.println("true");}
```


METODO EQUALS

- L'ordinamento naturale di una classe è detto consistente con `equals()` se e solo se `(e1.compareTo((Object)e2) == 0)` ha lo stesso valore `boolean` di `e1.equals((Object)e2)` per qualsiasi oggetto `e1` e `e2` di tale classe.
- Il metodo `equals()` ha valori di ritorno di tipo `boolean`, mentre il metodo `compareTo()` ha valori di ritorno di tipo `int`.

TIPI RUN-TIME: INSTACEOF

- Quando si usa un *cast* su un oggetto di tipo `Object` si suppone che il *tipo run-time* sia quello desiderato, se non è così viene lanciata un'eccezione.
- Vediamolo nell'esempio precedente:

```
public static void main(String[] args) {
      Libro b1 = new Libro(1234567890);
      Libro b2 = new Libro(1234567890);
      if (b1.equals("oggetto stringa"))
            System.out.println("true");}
```

ClassCastException

TIPI RUN-TIME: INSTACEOF

- Si può evitare l'eccezione utilizzando una parola chiave di Java `instanceof` che verifica se il tipo run-time del suo primo argomento è compatibile con quello del suo secondo argomento.
- Con riferimento al precedente esempio:

```
public boolean equals(Object obj) {
      ──────►if (obj instanceof Libro){
              if (((Libro)obj).ISBN==this.ISBN) {
                  return true;
              } else {
                  return false;
              }
            }else{
                  return false;
      }}
```


ECCEZIONI

- Vi possono essere situazioni in cui è necessario segnalare un comportamento anomalo con una *propria eccezione* (senza sfruttare quelle lanciate "da Java").
- Per esempio, un metodo per calcolare se le prenotazioni superano la disponibilità totale: il metodo ritorna un numero intero qualsiasi, ma le prenotazioni possono essere solo un numero intero positivo o nullo. Pertanto un argomento illegale (prenotazioni negative) deve essere segnalato. Non si può sfruttare un valore di ritorno perché sono tutti valori validi.

ECCEZIONI

```
public static void main(String[] args) {
  int risorse_libere;
  risorse_libere=prenotazione(10);
  System.out.println("Risorse libere:"+risorse_libere);
  risorse_libere=prenotazione(-20);
  System.out.println("Risorse libere:"+risorse_libere);
}
static int  prenotazione(int p){
  return elabora(p);
}
```

Risorse libere: 90
Risorse libere: 110

Supponendo di avere 100 connessioni disponibili.

ECCEZIONI

- Si deve definire una propria eccezione: una classe che ne estende una di Java.

```
public class NegativeArgumentException
     ──────►  extends RuntimeException{
  public NegativeArgumentException(){
     super();
  }
  public  NegativeArgumentException(String
                                       msg){
     super(msg);
}}
```

ECCEZIONI

- Il metodo precedente andrebbe definito nel seguente modo:

non è necessario indicare l'eccezione perché di tipo run-time

```
static int  prenotazione(int p){
    if (p<0)
          throw new NegativeArgumentException();
    return elabora(p);
}
```

Risorse libere: 90
Exception in thread "main" NegativeArgumentException

ECCEZIONI

- Oppure più *semplicemente* utilizzare un'eccezione già definita in Java, quindi senza definire una propria classe.

```
static int  prenotazione(int p){
    if (p<0)
          throw new IllegalArgumentException();
    return elabora(p);
}
```

Risorse libere: 90
Exception in thread "main" java.lang.IllegalArgumentException

- Tutte le eccezioni andrebbero gestite con il costrutto:
  `try{` azione `}` `catch(` TipoEccezione e `)` `{` gestore `}`

ITERATORI

- Una tipica operazione che si esegue su collezioni è esaminare tutti gli elementi uno alla volta. Per gli array è semplice scandire tutti gli elementi attraverso un indice e un ciclo `for`.
- In generale, una generica collezione non ha i dati organizzati in una struttura così rigida e lineare.
- Di conseguenza si introduce il concetto di iteratore, *generalizzando la scansione lineare di un array*.

ITERATORI

- Un iteratore è un oggetto di una classe che implementa l'interfaccia `Iterator`.
- Tale interfaccia contiene le firme di tre metodi: `boolean hasNext(); Object next(); void remove();`.
- Un iteratore è restituito da una collezione sulla quale si è invocato il metodo `Iterator iterator();` che consente di scandire i dati della collezione usando i metodi dell'interfaccia.
- Il metodo `hasNext()` restituisce `true` se esiste ancora un elemento nella collezione, `false` altrimenti.

ITERATORI

- Il metodo `next()` restituisce il successivo elemento da esaminare.
- Il metodo `remove()` cancella dalla collezione l'elemento restituito dall'ultima chiamata a `next()`, può essere invocato solo una volta dopo ogni `next()`.

- Sfruttando queste informazioni possiamo scrivere una porzione di codice che calcola il numero di elementi di qualsiasi collezione.

ITERATORI

```
public int size(Collection c){
  int num=0;
  for(Iterator i = c.iterator(); i.hasNext();
                                  i.next())
     num++;
  return num;
}
```

- Da questo esempio si può intuire la potenza di questo strumento. Tuttavia la sua implementazione ci porta a considerare le *classi interne*. Perché è un oggetto che può operare sui dati privati della collezione ed è ritornato da un metodo della classe della collezione, inoltre si possono creare più oggetti iteratore sulla stessa collezione.

CLASSI INTERNE

- Una classe interna (*inner class*) è definita internamente ad un'altra classe (*outer class*).
- La classe interna è visibile solo dalla classe che la include e può accedere a tutti i membri di tale classe.
- Le istanze della classe interna esistono solo all'interno di una istanza della classe che la include e sono creati solo attraverso la chiamata di un metodo della classe che la include.
- Solitamente la classe interna implementa un'interfaccia che permette di invocare dall'esterno i metodi della classe interna attraverso il riferimento creato da un metodo della classe che la include.

CLASSI INTERNE

- Il fatto di creare una classe interna, invece di creare un'unica classe, ci permette di creare più oggetti della classe interna che operano sullo stesso oggetto della classe esterna.

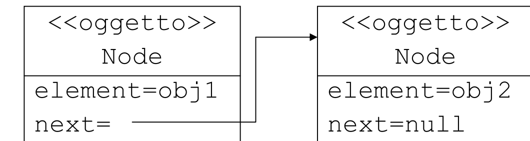
- Un esempio di classe interna è mostrato nella realizzazione di un iteratore per una collezione.

NODI

- Fino ad ora si sono realizzate le strutture dati astratte mediante array (semplici e performanti). Ciò è stato possibile anche perché le caratteristiche delle pile e delle code permettono una *simulazione* del loro comportamento in termini di celle memorizzate sequenzialmente.
- Tuttavia si può pensare di usare realizzazioni in cui l'*ordinamento* dei dati è solo *a livello logico* (senza usare un sottostante ordinamento fisico).
- Questo si può realizzare associando ad ogni elemento anche un *riferimento* all'elemento che lo segue nella sequenza.

NODI

- La struttura dati che sfrutta tale idea è la *lista*.
- Per ora vediamo di realizzare questa idea di *elementi più riferimenti*: un oggetto (nodo) che contenga, oltre all'informazione sull'elemento memorizzato, anche un riferimento al nodo che contiene l'elemento che nella sequenza lo segue.
- Esempio di due nodi:



NODI

- Una possibile definizione della classe `Node`

```
public class Node {
   Object element;
   Node next;
   public Node(Object element){
      this.element=element;
      next=null;
}}
```

- La dichiarazione *ricorsiva* (il campo `next` è di tipo `Node`) di un nodo può sembrare pericolosa, ma è corretta. Ciò che dichiara è un riferimento ad un oggetto di tipo `Node` e non un oggetto stesso.

NODI

- Vediamo un semplice uso di tale oggetto:

```
public static void main(String[] args) {
   Node n1,n2;
   n1=new Node("Primo");
   n2=new Node("Secondo");
   n1.next=n2;
   System.out.println(n1.element);
   System.out.println(n1.next.element);
}
```