Lezione XXVII
Lu 20-Nov-2006

File e Flussi in Java

1

Gestione di file in Java

- Finora abbiamo visto programmi Java che interagiscono con l'utente soltanto tramite i ***flussi*** standard di ingresso e di uscita (***System.in***, ***System.out***)
  - ciascuno di tali flussi può essere collegato a un file con un comando di sistema operativo (***re-indirizzamento***)
- Ci chiediamo: è possibile leggere e scrivere file in un programma Java?
  - con la redirezione di input/output, ad esempio, non possiamo leggere da due file o scrivere su due file...

2

Gestione di file in Java

- Limitiamoci inizialmente ad affrontare il problema della ***gestione di file di testo***
  - ***file contenenti caratteri***
- In un ***file di testo*** i numeri vengono memorizzati come stringhe di caratteri numerici decimali e punti: 123,45 diventa "123.45"
  - esistono anche i ***file binari***, che contengono semplicemente configurazioni di bit (byte) che rappresentano qualsiasi tipo di dati. Nei ***file binari*** i numeri sono memorizzati con la loro ***rappresentazione numerica***. Es.: l'intero 32 diventa la sequenza di 4 byte 00000000 00000000 000000000 00100000
- La gestione dei file avviene interagendo con il sistema operativo mediante classi del pacchetto ***java.io*** della libreria standard

3

Gestione di file in Java

- Per leggere/scrivere ***file di testo*** si usano oggetti creati con le classi
  - ***java.io.FileReader***
  - ***java.io.FileWriter***
- In generale le classi **reader** e **writer** sono in grado di gestire ***flussi di caratteri***
- Per leggere/scrivere ***file binari*** si usano oggetti delle classi
  - ***java.io.FileInputStream***
  - ***java.io.FileOutputStream***
- In generale le classi con suffisso ***Stream*** sono in grado di gestire flussi di byte

4

Lettura di file di testo

- Prima di ***leggere caratteri*** da un file (esistente) occorre ***aprire*** il file in ***lettura***
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileReader**

```
FileReader reader = new FileReader("file.txt");
```

  - il costruttore necessita del nome del file sotto forma di stringa: `"file.txt"`
  - se il file non esiste, viene lanciata l'eccezione **FileNotFoundException** a gestione obbligatoria

5

Lettura di file di testo

- Con l'oggetto di tipo ***FileReader*** si può invocare il metodo ***read()*** che restituisce ***un intero*** a ogni invocazione, iniziando dal primo carattere del file e procedendo fino alla fine del file stesso

```
FileReader reader = new FileReader("file.txt");
while(true)
{  int x = reader.read(); // read restituisce un
   if (x == -1) break;    // intero che vale -1
   char c = (char) x;     // se il file è finito
   // elabora c
} // il metodo lancia IOException, da gestire!
```

- Non è possibile tornare indietro e rileggere caratteri già letti
  - bisogna creare un nuovo oggetto di tipo ***FileReader***

6

Lettura con java.util.Scanner

- In alternativa, si può costruire un'esemplare della classe ***java.util.Scanner***, con il quale leggere righe di testo dal file

```
FileReader reader = new FileReader("file.txt");
Scanner in = new Scanner(reader);
while(in.hasNextLine())
{  String line = in.nextLine();
   ... // elabora
}
// FileReader lancia FileNotFoundException
//(IOException), da gestire obbligatoriamente!
```

- Quando il file finisce, il metodo nextLine() della classe Scanner ritorna **false**!

7

Lettura di file di testo

- Al termine della lettura del file (che non necessariamente deve procedere fino alla fine…) occorre ***chiudere*** il file

```
FileReader reader = new FileReader("file.txt");
...
reader.close();
```

- Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- Se il file non viene chiuso non si ha un errore, ma una potenziale situazione di instabilità per il sistema operativo

8

Scrittura di file di testo

- Prima di scrivere caratteri in un ***file di testo*** occorre ***aprire*** il file in scrittura
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo **PrintWriter**

Java 5.0

```
PrintWriter writer = new PrintWriter("file.txt");
```

  - il costruttore necessita del nome del file sotto forma di stringa e può lanciare l'eccezione **IOException**, che deve essere gestita
    - ***se il file non esiste, viene creato***
    - ***se il file esiste, il suo contenuto viene sovrascritto con i nuovi contenuti***
    - ***e se vogliamo aggiungere in coda al file senza cancellare il testo gia' contenuto?***

boolean append

```
FileWriter writer = new
    FileWriter("file.txt", true);
```

9

Scrittura di file di testo

- L'oggetto di tipo *FileWriter* non ha i comodi metodi **print()/println()**
  - è comodo creare un oggetto di tipo **PrintWriter** che incapsula l'esemplare di **FileWriter**, aggiungendo la possibilità di invocare **print()/println()** con qualsiasi argomento

```
FileWriter writer = new FileWriter("file.txt");
PrintWriter pw = new PrintWriter(writer);
pw.println("Ciao");
...
```

10

Scrittura di file di testo

- Al termine della scrittura del file occorre ***chiudere*** il file

```
PrintWriter writer = new FileWriter("file.txt");
...
writer.close();
```

- Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- ***Se non viene invocato non si ha un errore, ma è possibile che la scrittura del file non venga ultimata prima della terminazione del programma, lasciando il file incompleto***

11

```
/** Copia1.java -- copia un file di testo una riga alla
    volta
    i nomi dei file sono passati come parametri dalla
    riga di comando
*/
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Scanner;
public class Copia1
{ public static void main (String[] arg) throws
     IOException
  {
    Scanner in = new Scanner(new FileReader(arg[0]));
    PrintWriter out = new PrintWriter(new
      FileWriter(arg[1]));

    while (in.hasNextLine())
       out.println(in.nextLine());

    out.close(); //chiusura scrittore
    in. close(); //chiusura lettore
  }
}
```

12

Esempio:numerare le righe

```
/**
  NumeratoreRighe.java crea la copia di un file di testo
  numerandone le righe. I nomi dei file sono passati
  come parametri dalla riga di comando
*/
import java.io.IOException;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.Scanner;
public class NumeraRighe
{ public static void main (String[] args) throws
     IOException
  {
    FileReader reader = new FileReader(args[0]);
    Scanner in = new Scanner(reader);
    PrintWriter out = new PrintWriter(args[1]);
    int numRiga = 0;
    while (in.hasNextLine())
    { numRiga++;
      out.println(numRiga + " " + in.nextLine());
    }
    out.close(); //chiusura scrittore
    in. close(); //chiusura lettore
  }
}
```

13

Copiare a caratteri un file di testo

```
/** Copia2.java -- copia un file di testo un carattere
    alla volta
    fare attenzione a non cercare di copiare un file
    binario con questo programma
 */
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class Copia2
{ public static void main (String[] arg) throws
     IOException
  {  FileReader in = new FileReader(arg[0]);
     FileWriter out = new FileWriter(arg[1]);
     int c = 0;
     while ((c = in.read()) != -1)
       out.write(c);
     out.close(); //chiusura scritture
     in.close();  //chiusura lettore
  }
}
```

14

Scrittura di file di testo

- Al termine della scrittura del file occorre ***chiudere*** il file

```
PrintWriter writer = new FileWriter("file.txt");
...
writer.close();
```

- Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- ***Se non viene invocato non si ha un errore, ma è possibile che la scrittura del file non venga ultimata prima della terminazione del programma, lasciando il file incompleto***

15

Scrittura di file di testo

- Al termine della scrittura del file occorre ***chiudere*** il file

```
PrintWriter writer = new FileWriter("file.txt");
...
writer.close();
```

- Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- ***Se non viene invocato non si ha un errore, ma è possibile che la scrittura del file non venga ultimata prima della terminazione del programma, lasciando il file incompleto***

16

Lettura di file binari

- Prima di ***leggere byte*** da un file (esistente) occorre ***aprire*** il file in ***lettura***
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileInputStream**

```
FileInputStream inputStream = new
   FileInputStream("file.txt");
```

  - il costruttore necessita del nome del file sotto forma di stringa
  - se il file non esiste, viene lanciata l'eccezione **FileNotFoundException** che deve essere gestita

17

Lettura di file binari

- Con l'oggetto di tipo **FileInputStream** si può invocare il metodo **read()** che restituisce un ***intero*** (fra 0 e 255 o -1) a ogni invocazione, iniziando dal primo byte del file e procedendo fino alla fine del file stesso

```
FileInputStream in = new
   FileInputStream("file.txt");
while(true)
{  int x = in.read();     // read restituisce un
   if (x == -1) break;    // intero che vale -1
   byte b = (byte) x;     // se il file è finito
   // elabora b
} // il metodo lancia IOException, da gestire
```

- Non è possibile tornare indietro e rileggere caratteri già letti
  - bisogna creare un nuovo oggetto di tipo **FileInputStream**

18

Scrittura di file binari

- Prima di scrivere byte in un *file binario* occorre ***aprire*** il file in scrittura
  - questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileOutputStream**

```
FileOutputStream outputStream = new
   FileOutputStream("file.txt");
```

  - il costruttore necessita del nome del file sotto forma di stringa e può lanciare l'eccezione **IOException**, che deve essere gestita
    - ***se il file non esiste, viene creato***
    - ***se il file esiste, il suo contenuto viene sovrascritto con i nuovi contenuti***
    - ***e se vogliamo aggiungere in coda al file senza cancellare il testo gia' contenuto?***

```
boolean append

FileOutputStream outputStream = new
    FileOutputStream("file.txt", true);
```

19

Scrittura di file binario

- L'oggetto di tipo **FileOutputStream** ha il metodo write() che scrive un byte alla volta
- Al termine della scrittura del file occorre ***chiudere*** il file

```
FileOutputStream outStream = new
   FileOutputStream("file.txt");
...
outStream.close();
```

- Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- ***Se non viene invocato non si ha un errore, ma è possibile che la scrittura del file non venga ultimata prima della terminazione del programma, lasciando il file incompleto***

20

Gestione di file in Java

- Usando le classi
  - ***FileReader***, ***FileWriter*** e PrintWriter del pacchetto **java.io** è possibile manipolare, all'interno di un programma Java, più ***file di testo*** in lettura e/o più file in scrittura
  - ***FileInputStream*** e ***FileOutputStream*** del pacchetto **java.io** è possibile manipolare, all'interno di un programma Java, più ***file binari*** in lettura e/o più file in scrittura
  - I metodi read() e write() di queste classi sono gli unici metodi basilari di lettura/scrittura su file della libreria standard
  - Per elaborare linee di testo (o addirittura interi oggetti in binario) i flussi e i lettori devono essere combinati con altre classi, ad esempio Scanner o PrintWriter

21

Copiare un file binario

```
/**
 Copia3.java - copia un file binario un byte alla volta
*/
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class Copia3
{
  public static void main (String[] arg) throws
    IOException
  {
    FileInputStream in = new FileInputStream(arg[0]);
    FileOutputStream out = new
       FileOutputStream(arg[1]);
    int c = 0;
    while ((c = in.read()) != -1)
       out.write(c);
    out.close(); //chiusura scrittore
    in.close();  //chiusura lettore
  }
}
```

22

Fine riga e EOF

23

Che cosa è il *fineriga*?

- Un file di testo è un file composto da caratteri (nel nostro caso da caratteri di un byte secondo la codifica ASCII)
- Il file è diviso in *righe* cioè in sequenze di caratteri terminati dalla *stringa fineriga*
- Il ***fineriga*** dipende dal sistema operativo
  - "\r\n" Windows
  - "\n" Unix – Linux
  - "\r" MAC OS
- L'uso di uno ***`Scanner`*** e del metodo ***`nextLine()`*** in ingresso, di un ***`PrintWriter`*** e del metodo ***`println()`*** in uscita, consentono di ignorare il problema della dipendenza del ***fineriga*** dall'ambiente in cui un'applicazione è utilizzata
- Se si elabora un file di testo un carattere alla volta è necessario gestire le differenze fra i diversi ambienti, in particolare il fatto che in due casi la stringa fineriga è di un carattere e nell'altro e composta da due caratteri

24

Che cosa è l'*End Of File* (EOF)

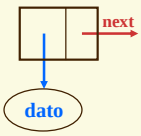
- La fine di un file (EOF) corrisponde alla fine dei dati che compongono il file
- L'EOF non è un carattere contenuto nel file!
- La condizione di EOF viene segnalata quando si cerca di leggere un byte o un carattere dopo la fine fisica del file
- Per segnalare la fine di un file da tastiera si usa una sequenza di controllo
  - Ctr z Windows
  - Ctr d Unix

25

Lista concatenata (Linked List)

26

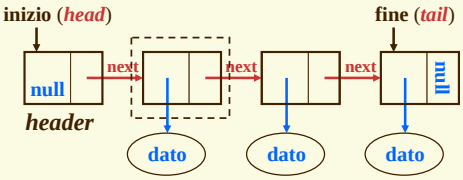
Lista concatenata (*linked list*)



- La ***catena*** o ***lista concatenata*** (***linked list***) è una **struttura dati** alternativa all'array (eventualmente) riempito solo in parte per la realizzazione di classi
- Una catena è un insieme ***ordinato*** di ***nodi***
  - ogni nodo è un oggetto che contiene
    - un riferimento a un elemento (***il dato***)
    - un riferimento al nodo ***successivo*** nella catena (**next**)

27

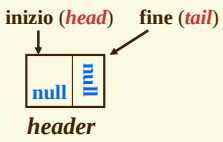
Lista Concatenata



- Per agire sulla catena è sufficiente memorizzare il ***riferimento al suo primo nodo***
  - è comodo avere anche un riferimento all'ultimo nodo
- Il campo **next** dell'ultimo nodo contiene **null**
- Vedremo che è comodo avere un primo nodo senza dati, chiamato ***header***
  - si dice ***lista concatenata con nodo sentinella***
- ***head*** e' il riferimento al ***primo nodo*** (header), ***tail*** all'ultimo nodo

28

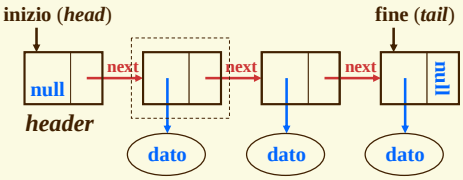
Lista concatenata vuota



- Per capire bene il funzionamento della catena con nodo sentinella, è necessario avere ben chiara la rappresentazione della ***catena vuota***
  - contiene il solo nodo ***header***, che ha **null** in entrambi i suoi campi
  - **head** e **tail** puntano entrambi a tale ***header***

29

Lista Concatenata



- Per accedere in sequenza a tutti i nodi della catena si parte dal riferimento ***head*** e si seguono i riferimenti contenuti nel campo **next** di ciascun nodo
  - non è possibile scorrere la lista in senso inverso
  - la scansione termina quando si trova il nodo con il valore **null** nel campo **next**
- L'Accesso nella lista concatenata è sequenziale

30

Nodo di una Lista Concatenata

```
public class ListNode
{  private Object element;
   private ListNode next; //stranezza

   public ListNode(Object e, ListNode n)
   {  element = e;
      next = n;
   }

   public ListNode()
   {  this(null, null);
   }

   public Object getElement() { return element; }

   public ListNode getNext()  { return next; }

   public void setElement(Object e) { element = e; }

   public void setNext(ListNode n) {  next = n; }
}
```

Auto-riferimento

```
public class ListNode
{  ...
   private ListNode next; //stranezza
}
```

- Nella definizione della classe **ListNode** notiamo una "stranezza"
  - la classe definisce e usa ***riferimenti a oggetti del tipo che sta definendo***
- Ciò è perfettamente lecito e ***si usa molto spesso*** quando si rappresentano "strutture a definizione ricorsiva" come la catena

32

Incapsulamento eccessivo?

- A cosa serve l'incapsulamento in classi che hanno lo stato completamente accessibile tramite metodi?
  - ***apparentemente a niente***…
- Supponiamo di essere in fase di debugging e di aver bisogno della visualizzazione di un messaggio ogni volta che viene modificato il valore di una variabile di un nodo
  - se non abbiamo usato l'incapsulamento, occorre aggiungere enunciati in tutti i punti del codice dove vengono usati i nodi…
  - elevata probabilità di errori o dimenticanze

33

Incapsulamento eccessivo?

- Se invece usiamo l'incapsulamento
  - è sufficiente inserire l'enunciato di visualizzazione all'interno dei metodi **set()** che interessano
  - le variabili di esemplare possono essere modificate ***SOLTANTO*** mediante l'invocazione del corrispondente metodo **set()**
  - terminato il debugging, per eliminare le visualizzazioni è sufficiente modificare il solo metodo **set()**, senza modificare di nuovo moltissime linee di codice

34

Lezione XXVIII
Ma 21-Nov-2006

Lista Concatenata

35

Lista Concatenata

- I metodi utili per una catena sono
  - **addFirst()** per inserire un oggetto all'inizio della catena
  - **addLast()** per inserire un oggetto alla fine della catena
  - **removeFirst( )** per eliminare il primo oggetto della catena
  - **removeLast()** per eliminare l'ultimo oggetto della catena
- Spesso si aggiungono anche i metodi
  - **getFirst()** per esaminare il primo oggetto
  - **getLast()** per esaminare l'ultimo oggetto
- Si osservi che non vengono ***mai restituiti né ricevuti*** riferimenti ai ***nodi***, ma sempre ai ***dati*** contenuti nei nodi

36

Lista Concatenata

- Infine, dato che anche la catena è un contenitore, ci sono i metodi
  - **isEmpty()** per sapere se la catena è vuota
  - **makeEmpty()** per rendere vuota la catena
  - **size()** che restituisce il numero di elementi nel contenitore. Per ora ***non realizziamo*** il metodo nella lista concatenata.
- Si definisce l'eccezione **EmptyLinkedListException**

37

Eccezione EmptyLinkedListException

```
public class EmptyLinkedListException
   extends RuntimeException
{
   public EmptyLinkedListException()
   {
   }

   public EmptyLinkedListException(String err)
   {
      super(err);
   }
}
```

- Estendiamo l'eccezione ***java.lang.RuntimeException***, così non siamo costretti a gestirla

38

Interfaccia Container

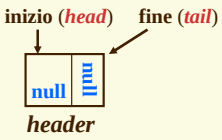
```
public interface Container
{
   /**
      verifica se il contenitore e' vuoto
      @return true se il contenitore e' vuoto,
      false altrimenti
   */
   boolean isEmpty();

   /**
      rende vuoto il contenitore
   */
   void makeEmpty();

}
```

Non inseriamo il metodo size()!

39

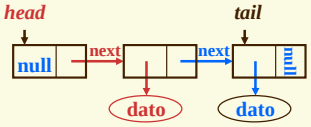
Lista Concatenata

```
public class LinkedList implements Container
{
   // parte privata
   private ListNode head, tail;

   public LinkedList()
   {  makeEmpty();
   }

   public void makeEmpty()
   {  head = tail = new ListNode();
   }

   public boolean isEmpty()
   {  return (head == tail);
   }
...
}
```



Lista Cancatenata

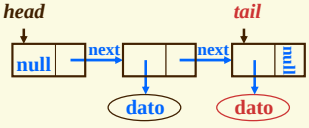
```
public class LinkedList implements Container
{
   ...
   public Object getFirst() // operazione O(1)
   {
      if (isEmpty())
         throw new EmptyLinkedListException();

      return head.getNext().getElement();
   }
...
}
```



41

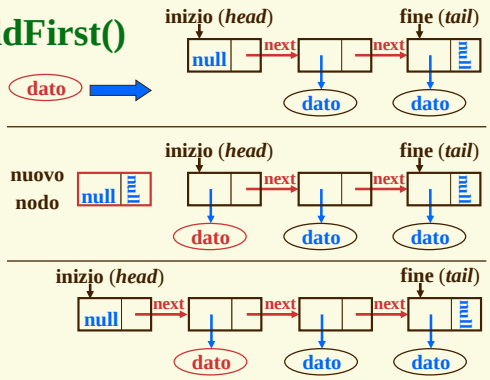
Lista Cancatenata

```
public class LinkedList implements Container
{
   ...
   public Object getLast()  // operazione O(1)
   {
      if (isEmpty())
         throw new EmptyLinkedListException();

      return tail.getElement();
   }
   ...
}
```



42

addFirst()



43

addFirst()

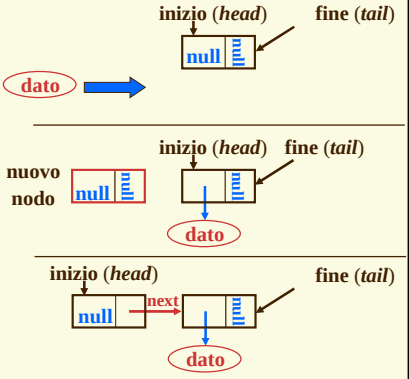
```
public class LinkedList ...
{  ...
  public void addFirst(Object e) {
    // inserisco il dato nell'header attuale
    head.setElement(e);
    // creo un nodo con due riferimenti null
    ListNode newNode = new ListNode();
    // collego il nuovo nodo all'header attuale
    newNode.setNext(head);
    // il nuovo nodo diventa il nodo header
    head = newNode;
    // tail non viene modificato
  }
}
```

- Non esiste il problema di "catena piena"
- L'operazione è **O(1)**

44

addFirst()

- Verifichiamo che tutto sia corretto anche inserendo in una ***catena vuota***
- ***Fare sempre attenzione ai casi limite***



45

addFirst()

```
public void addFirst(Object e) {
   head.setElement(e);
   ListNode n = new ListNode();
   n.setNext(head);
   head = n;
}
```

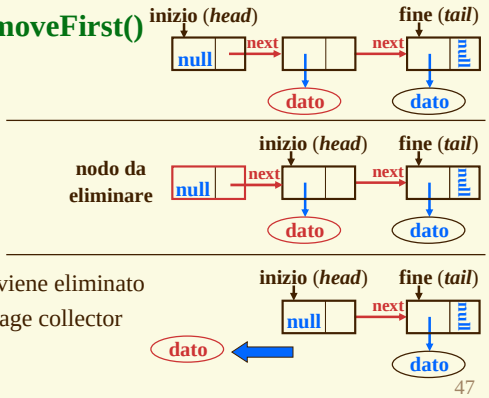
- Il codice di questo metodo si può esprimere anche in modo più conciso

```
public void addFirst(Object e) {
   head.setElement(e);
   // funziona perché prima head viene USATO
   // (a destra) e solo successivamente viene
   // MODIFICATO (a sinistra)
   head = new ListNode(null, head);
}
```

- È più "professionale", anche se meno leggibile

46

removeFirst()



Il nodo viene eliminato dal garbage collector

47

removeFirst()

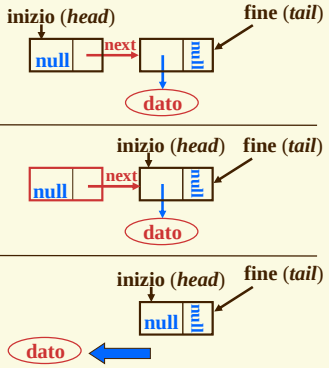
```
public class LinkedList ...
{  ...
   public Object removeFirst() {
      // delega a getFirst il
      // controllo di lista vuota
      Object e = getFirst();

      // aggiorno l'header
      head = head.getNext();
      head.setElement(null);
      return e;
   }
}
```

- L'operazione è **O(1)**

48

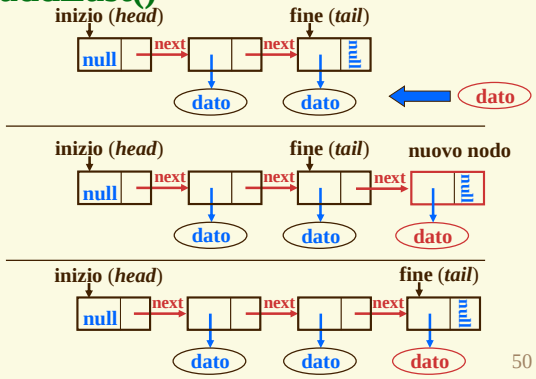
removeFirst()

- Verifichiamo che tutto sia corretto anche rimanendo con una *catena vuota*
- *Fare sempre attenzione ai casi limite*



49

addLast()



50

addLast()

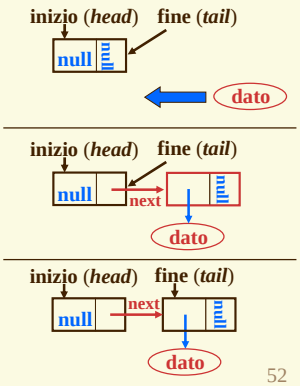
```
public class LinkedList ...
{  ...
   public void addLast(Object e) {
      tail.setNext(new ListNode(e, null));
      tail = tail.getNext();
   }
}
```

- Non esiste il problema di "catena piena"
- Anche questa operazione è `O(1)`

51

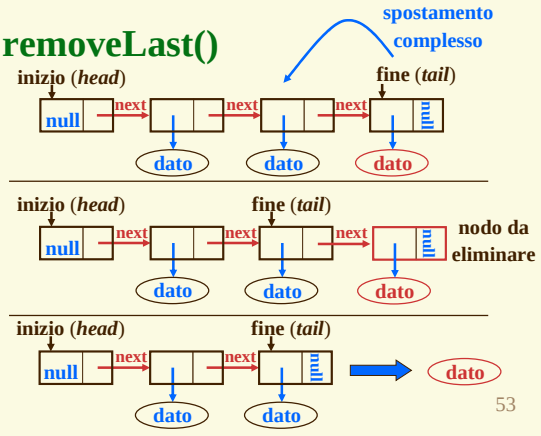
addLast()

- Verifichiamo che tutto sia corretto anche inserendo in una *catena vuota*
- *Fare sempre attenzione ai casi limite*



52

removeLast()



53

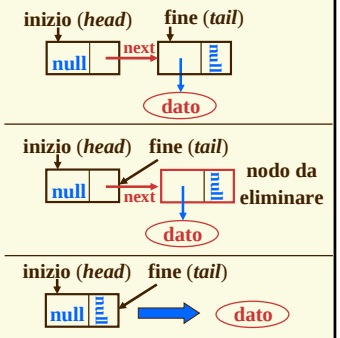
removeLast()

```
public class LinkedList ...
{  ...
  public Object removeLast() {
    Object e = getLast();
    // bisogna cercare il penultimo nodo
    // partendo dall'inizio e finché non si
    // arriva alla fine della catena
    ListNode temp = head;
    while (temp.getNext() != tail)
       temp = temp.getNext();
    // a questo punto temp si riferisce al
    // penultimo nodo
    tail = temp;
    tail.setNext(null);
    return e;
  }
}
```

Operazione `O(n)`

removeLast()

- Verifichiamo che tutto sia corretto anche rimanendo con una *catena vuota*
- *Fare sempre attenzione ai casi limite*



55

Header della lista concatenata

- La presenza del nodo ***header*** nella catena rende più semplici i metodi della catena stessa
  - in questo modo, non è necessario gestire i casi limite in modo diverso dalle situazioni ordinarie
- Senza usare il nodo ***header***, le prestazioni asintotiche rimangono comunque le stesse
- Usando il nodo ***header*** si "spreca" un nodo
  - per valori elevati del numero di dati nella catena questo spreco, in percentuale, è trascurabile

56

Prestazioni lista concatenata

- Tutte le operazioni sulla lista ***cancatenata*** sono `O(1)` tranne **removeLast()** che è `O(n)`
  - si potrebbe pensare di tenere un riferimento anche al ***penultimo*** nodo, ma per ***aggiornare*** tale riferimento sarebbe comunque necessario un tempo `O(n)`
- Se si usa una catena con il solo riferimento **head**, anche **addLast()** diventa `O(n)`
  - per questo è utile usare il riferimento **tail**, che migliora le prestazioni di **addLast()** senza peggiorare le altre e non richiede molto spazio di memoria

57

Prestazioni della lista concatenata

- Non esiste il problema di "catena piena"
  - non bisogna mai "ridimensionare" la catena
  - la JVM lancia l'eccezione **OutOfMemoryError** se viene esaurita la memoria disponibile (***java heap***)
- Non c'è spazio di memoria sprecato (come negli array "riempiti solo in parte")
  - un nodo occupa però più spazio di una cella di array, almeno il doppio (contiene due riferimenti anziché uno)

58

Riepilogo delle prestazioni della lista cancatenata

```
{
private ListNode head, tail;
public LinkedList(){… }
public void makeEmpty(){ }            // O(1)
public boolean isEmpty(){ }           // O(1)
public Object getFirst(){ }           // O(1)
public Object getLast() { }           // O(1)
public void addFirst(Object obj) { }  // O(1)
public Object removeFirst() { }       // O(1)
public void addLast(Object obj) { }   // O(1)
public Object removeLast(){ }         // O(n)
}
```

59

Cenni alle Classi Interne

60

Lista Concatenata

```
public class LinkedList
{
   private ListNode head, tail;

   public LinkedList(){… }
   public void makeEmpty(){ }  // O(1)
   public boolean isEmpty(){ } // O(1)
   public Object getFirst(){ } // O(1)
   public Object getLast() { } // O(1)
   public void addFirst(Object obj) { } // O(1)
   public Object removeFirst() { }      // O(1)
   public void addLast(Object obj) { }  // O(1)
   public Object removeLast(){ }        // O(n)
}
```

61

Classi interne

- Osserviamo che la classe **ListNode**, usata dalla catena, non viene usata al di fuori della catena stessa
  - la catena non restituisce mai riferimenti a **ListNode**
  - la catena non riceve mai riferimenti a **ListNode**
- Per il principio dell'incapsulamento (***information hiding***) sarebbe preferibile che questa classe e i suoi dettagli non fossero visibili all'esterno della catena
  - in questo modo una modifica della struttura interna della catena e/o di **ListNode** non avrebbe ripercussioni sul codice scritto da chi usa la catena

62

Classi interne

- Il linguaggio Java consente di ***definire classi all'interno di un'altra classe***
  - tali classi si chiamano ***classi interne*** (***inner classes***)
- L'argomento è molto vasto
- A noi interessa solo il fatto che se una classe interna viene definita dentro un'altra classe essa è ***accessibile*** (in tutti i sensi) ***soltanto all'interno della classe*** in cui è definita
  - all'esterno non è nemmeno possibile creare oggetti di tale classe interna

```
public class LinkedList ...
{  ...
   class ListNode
   {  ... }
}
```

Lista Concatenata

```
public class LinkedList
{  private ListNode head, tail;

   public LinkedList(){ }
   public void makeEmpty(){ }  // O(1)
   public boolean isEmpty(){ } // O(1)
   public Object getFirst(){ } // O(1)
   public Object getLast() { } // O(1)
   public void addFirst(Object obj) { } // O(1)
   public Object removeFirst() { }      // O(1)
   public void addLast(Object obj) { }  // O(1)
   public Object removeLast(){ }        // O(n)

   class ListNode
   {
   . . .
   }
}
```

Lista doppiamente concatenata (*doubly linked list*)

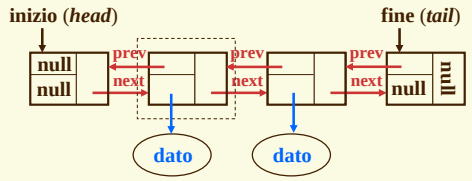
65

Catena doppia

- La ***catena doppia*** (lista doppiamente concatenata, ***doubly linked list***) è una struttura dati.
- Una catena doppia è un insieme ***ordinato*** di ***nodi***
  - ogni nodo è un oggetto che contiene
    - un riferimento ad un elemento (***il dato***)
    - un riferimento al nodo successivo della lista (**next**)
    - un riferimento al nodo precedente della lista (**prev**)

66

Catena doppia



- Dato che la struttura è ora simmetrica, si usano due nodi che non contengono dati, uno a ciascun estremo della catena

67

Catena doppia

- Tutto quanto detto per la catena (semplice) può essere agevolmente esteso alla catena doppia
- Il metodo **removeLast()** diventa **O(1)** come gli altri metodi
- I metodi di inserimento e rimozione si complicano

68

Tipi di dati astratti e strutture dati

69

Strutture dati

- Una ***struttura dati*** (***data structure***) è un ***modo sistematico*** di ***organizzare*** i dati in un ***contenitore*** e di ***controllare l'accesso*** ai dati
- In Java una ***struttura dati*** viene definita tramite ***una classe, ad esempio***
  - la ***lista concatenata*** è una ***struttura dati***
    - è un ***contenitore*** che ***memorizza dati*** in modo ***sistematico (nei nodi)***
    - l'***accesso*** ai dati è ***controllato*** tramite metodi di accesso (***addFirst()***, ***addLast()***, ***getFirst()***...)

70

Strutture dati

- Anche ***java.util.Vector*** è un'esempio di ***struttura dati***:
  - realizza un array che può crescere. Come negli array, agli elementi si può accedere tramite un ***indice***. La dimensione dell'array può aumentare o diminuire, secondo le necessità, per permettere l'inserimento o la rimozione degli elementi dopo che il vettore è stato creato.

71

Tipi di dati astratti

- Un ***tipo di dati astratto*** (**ADT**, ***Abstract Data Type***) è una rappresentazione ***astratta*** di una struttura dati, un modello che specifica:
  - il tipo di dati memorizzati
  - le operazioni che si possono eseguire sui dati
  - il tipo delle informazioni necessarie per eseguire le operazioni

72

Tipi di dati astratti

- In Java si definisce un ***tipo di dati astratto*** con una ***interfaccia***
- Come sappiamo, un'interfaccia descrive un ***comportamento*** che sarà assunto da una classe che realizza l'interfaccia
  - è proprio quello che serve per definire un ADT
- Un ADT definisce ***cosa*** si può fare con una struttura dati che realizza l'interfaccia
  - la classe che rappresenta concretamente la struttura dati definisce invece ***come*** vengono eseguite le operazioni

73

Tipi di dati astratti

- Un ***tipo di dati astratto*** mette in generale a disposizione metodi per svolgere le seguenti azioni
  - ***inserimento*** di un elemento
  - ***rimozione*** di un elemento
  - ***ispezione*** degli elementi contenuti nella struttura
    - ***ricerca*** di un elemento all'interno della struttura
- I diversi ADT che vedremo si differenziano per le modalità di funzionamento di queste tre azioni

74

Il pacchetto `java.util`

- Il pacchetto **java.util** della libreria standard contiene molte definizioni di ADT come interfacce e loro realizzazioni (***strutture dati***) come classi
- La nomenclatura e le convenzioni usate in questo pacchetto sono, però, piuttosto diverse da quelle tradizionalmente utilizzate nella teoria dell'informazione (*purtroppo e stranamente...*)
- Quindi, proporremo un'esposizione teorica di ADT usando la terminologia tradizionale, senza usare il pacchetto **java.util** della libreria standard

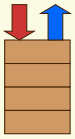
75

Lezione XXIX
Me 22-Nov-2006

ADT Pila (*Stack*)

76

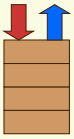
Pila (*stack*)



- In una ***pila*** (***stack***) gli oggetti possono essere inseriti ed estratti secondo un comportamento definito **LIFO** (***Last In, First Out***)
  - l'ultimo oggetto inserito è il primo a essere estratto
  - il nome è stato scelto in analogia con una ***pila*** di piatti
- L'unico oggetto che può essere ispezionato è quello che si trova in cima alla pila
- Esistono molti possibili utilizzi di una struttura dati con questo comportamento
  - la JVM usa una pila per memorizzare l'elenco dei metodi in attesa durante l'esecuzione in un dato istante

77

Pila (*stack*)



- I browser per internet memorizzano gli indirizzi dei siti visitati recentemente in una struttura di tipo ***pila***. Quando l'utente visita un sito, l'indirizzo è inserito (***push***) nella pila. Il browser permette all'utente di saltare indietro (***pop***) al sito precedente tramite il pulsante "indietro"
- Gli editor forniscono generalmente un meccanismo di "***undo***" che cancella operazioni di modifica recente e ripristina precedenti stati del testo. Questa funzione di "undo" è realizzata memorizzando le modifiche in una struttura di tipo pila.

78

Pila (*stack*)

- I metodi che caratterizzano una pila sono
  - **push()** per inserire un oggetto in cima alla pila
  - **pop()** per eliminare l'oggetto che si trova in cima alla pila. Genera l'eccezione ***EmptyStackException*** se la pila è vuota
  - **top()** per ispezionare l'oggetto che si trova in cima alla pila, senza estrarlo. Genera l'eccezione ***EmptyStackException*** se la pila è vuota
- Infine, ogni ADT di tipo "contenitore" ha i metodi
  - **isEmpty()** per sapere se il contenitore è vuoto
  - **makeEmpty()** per vuotare il contenitore
  - **size()** per conoscere il numero di elementi contenuti nella struttura dati

79

Estendere un'interfaccia

```
public interface Stack extends Container
{  ... // push, pop, top }
```

- Anche le ***interfacce***, come le classi, possono essere "***estese***"
- Un'interfaccia eredita tutti i metodi della sua ***super-interfaccia***
- Nel realizzare ***un'interfaccia estesa*** occorre realizzare anche i metodo della sua ***super-interfaccia***

```
public interface Container
{ /**
   @return true se il contenitore e'vuoto, false
    altrimenti
  */
  boolean isEmpty();

  /** rende vuoto il contenitore */
  void makeEmpty();

  /** @return il numero di oggetti nel contenitore */
  int size();
}
```

Solita definizione dell'interfaccia Container

80

EmptyStackException

```
public class EmptyStackException
   extends RuntimeException
{
   public EmptyStackException()
   {
   }

   public EmptyStackException(String err)
   {
      super(err);
   }
}
```

- Estendiamo l'eccezione ***java.lang.RuntimeException***, così non siamo costretti a gestirla

81

Pila (*stack*)

Definiremo tutti gli ADT in modo che possano genericamente contenere oggetti di tipo **Object**

- in questo modo potremo inserire nel contenitore oggetti di qualsiasi tipo, perché qualsiasi oggetto può essere assegnato a un riferimento di tipo **Object**

82

```
/**
   Tipo di dati astratto con modalità di accesso LIFO
   @see Container
   @see EmptyStackException
*/
public interface Stack extends Container
{  /**
      inserisce un elemento in cima alla pila
      @param obj l'elemento da inserire
   */
   void push(Object obj);

   /**
      rimuove l'elemento dalla cima della pila
      @return l'elemento rimosso
      @throws EmptyStackException se la pila e' vuota
   */
   Object pop() throws EmptyStackException;

   /**
      ispeziona l'elemento in cima alla pila
      @return l'elemento in cima alla pila
      @throws EmptyStackException se la pila e' vuota
   */
   Object top() throws EmptyStackException;
}
```

Osservazioni

```
@see Container
@see EmptyStackException
```

- **@see** è una direttiva per il programma di generazione automatica della documentazione ***javadoc***.
- Nella documentazione viene segnalato di consultare l'oggetto che segue la direttiva

84

Osservazioni

```
Object pop() throws EmptyStackException
```

- L'eccezione EmptyStackException è a gestione facoltativa, quindi non è obbligatorio segnalarla nell'intestazione del metodo
- Spesso lo si fa per segnalare chiaramente che il metodo può lanciare l'eccezione

pop

```
java.lang.Object pop()
                  throws EmptyStackException
```

rimuove l'elemento dalla cima della pila

Returns:
l'elemento rimosso
Throws:
EmptyStackException - se la pila e' vuota

85

Utilizzo della pila

- Per evidenziare la potenza della definizione di tipi di dati astratti come interfacce, supponiamo che qualcun altro abbia progettato la seguente classe

```
public class StackX implements Stack
{  ...
}
```

- Senza sapere come sia realizzata **StackX**, possiamo usarne un esemplare mediante il suo comportamento astratto definito in **Stack**
- Allo stesso modo, possiamo usare un esemplare di un'altra classe **StackY** che realizza **Stack**

86

```
public class StackReverser // UN ESEMPIO
{ public static void main(String[] args)
  {  Stack st = new StackX();
     st.push("Pippo");
     st.push("Pluto");
     st.push("Paperino");
     printAndClear(st);
     System.out.println();
     st.push("Pippo");
     st.push("Pluto");
     st.push("Paperino");
     printAndClear(reverseAndClear(st));
  }
  private static Stack reverseAndClear(Stack s)
  {  Stack p = new StackY();
     while (!s.isEmpty())
        p.push(s.pop());
     return p;
  }
  private static void printAndClear(Stack s)
  {  while (!s.isEmpty())
       System.out.println(s.pop());
  }
}
```

```
Paperino
Pluto
Pippo

Pippo
Pluto
Paperino
```

87

Realizzazione della pila

- Per ***realizzare una pila*** è facile ed efficiente usare una struttura di tipo ***array*** "riempito solo in parte"
- Il solo problema che si pone è ***cosa fare quando l'array è pieno*** e viene invocato il metodo **push()**
  - la prima soluzione proposta prevede il ***lancio di un'eccezione***
  - la seconda soluzione proposta usa il ***ridimensionamento dell'array***

88

Eccezioni nella pila

- Definiamo la classe **FullStackException**, come estensione di **RuntimeException**, in modo che chi usa la pila ***non sia obbligato a gestirla***

```
public class FullStackException
       extends RuntimeException
{
}
```

Realizzazione minima di un'interfaccia

89

Realizzazione minima

```
public class FullStackException
       extends RuntimeException
{  }
```

- La classe è **vuota**, a cosa serve?
  - serve a definire un nuovo tipo di dato (un'eccezione) che ha le stesse identiche caratteristiche della superclasse da cui viene derivato, ma di cui interessa porre in evidenza il ***nome***, che contiene l'informazione di identificazione dell'errore
- Con le eccezioni si fa talvolta così
  - in realtà la classe non è vuota, perché contiene tutto ciò che eredita dalla sua superclasse

90

```
public class FixedArrayStack implements Stack
{
  private final int CAPACITY = 100;

  // v è un array riempito solo in parte
  protected Object[] v;  // considerare
  protected int vSize;   // protected
                         // uguale a private
  public FixedArrayStack()
  {
    /* per rendere vuota la struttura, invoco
       il metodo makeEmpty; è sempre meglio
       evitare di scrivere codice ripetuto
    */
    makeEmpty();
  }
  ...
```

91

```
  /** rende vuoto il contenitore */
  public void makeEmpty()
  {  vSize = 0;
     v = new Object[CAPACITY];
  }
  /**
    verifica se il contenitore e' vuoto
    @return true se vuota, false altrimenti
  */
  public boolean isEmpty()
  {  return (vSize == 0);
  }
  /**
     calcola il numero di elementi nel contenitore
     @return il numero di elementi nel contenitore
  */
  public int size()
  {  return vSize;
  }
   ...
}
```

CAPACITY
dimensione scelta per il contenitore

dato che Stack estende Container, occorre realizzare anche i suoi metodi

```
public class FixedArrayStack implements Stack
{  ...
  public void push(Object obj) throws FullStackException
  {  if (vSize >= CAPACITY)
        throw new FullStackException();
     v[vSize++] = obj;
  }
  public Object top() throws EmptyStackException
  {  if (isEmpty())
        throw new EmptyStackException();
     return v[vSize - 1];
  }
  public Object pop() throws EmptyStackException
  { Object obj = top(); // lascia a top() l'eccezione
     vSize--;
     v[vSize] = null; //garbage collection
     return obj;
  }
}
```

93

Pila con ridimensionamento

- Definiamo una pila che non generi mai l'eccezione **FullStackException**

```
public class GrowingArrayStack implements Stack
{
   public void push(Object obj)
   {  if (vSize >= v.length)
         v = resize(v, 2*vSize);
      v[vSize++] = obj;
   }

   private static Object[] resize(Object[] a, int length)
   {...}
   ... // tutto il resto è identico!
}
```

- Possiamo evitare di riscrivere tutto il codice di **FixedArrayStack** in **GrowingArrayStack**?

94

Pila con ridimensionamento

```
public class GrowingArrayStack extends FixedArrayStack
{
   public void push(Object obj)
   {  if (vSize == v.length)
         v = resize(v, 2 * vSize);
      super.push(obj);
   }
   private static Object[] resize(Object[] a, int length)
   {...}
}
```

- Il metodo **push()** sovrascritto deve poter accedere alle variabili di esemplare della superclasse
- Questo è consentito dalla definizione **protected**
  - alle variabili **protected** ***si può accedere dalle classi derivate***
  - ***ma anche dalle classi dello stesso pacchetto!!!***
- Se le variabili fossero state private, non sarebbe stato possibile ridimensionare dinamicamente nella classe GrowingArrayStack

95

Accesso protected

- Il progettista della superclasse decide se rendere accessibile in modo **protected** lo stato della classe (o una sua parte…)
- È una violazione dell'incapsulamento, che avviene in modo consapevole ed esplicito
- Anche i metodi possono essere definiti **protected**
  - possono essere invocati soltanto all'interno della classe in cui sono definiti (come i metodi **private**) ***e all'interno delle classi derivate da essa***

In questo corso si scoraggia l'uso dello specificatore di accesso `protected`!!!

- Talvolta può, però, essere utile per rendere più semplice (ma meno sicuro) il codice.

96

Prestazioni della pila

- Il tempo di esecuzione di ogni operazione su una ***pila realizzata con array di dimensioni fisse*** è ***costante***, cioè non dipende dalla dimensione **n** della struttura dati stessa (non ci sono cicli…)
  - si noti che ***le prestazioni dipendono dalla realizzazione della struttura dati*** e non dalla sua interfaccia…
  - per valutare le prestazioni è necessario conoscere il codice che realizza le operazioni!

97

Prestazioni della pila

- Un'operazione eseguita in un tempo costante, cioè in un tempo che non dipende dalle dimensioni del problema, ha un andamento asintotico **O(1)**, perché
  - eventuali costanti moltiplicative vengono trascurate
- Ogni operazione eseguita su **FixedArrayStack** è quindi **O(1)**

98

Prestazioni della pila

- Nella realizzazione ***con array ridimensionabile***, l'unica cosa che cambia è l'operazione **push()**
  - "**alcune volte**" richiede un tempo **O(n)**
    - tale tempo è necessario per copiare tutti gli elementi nel nuovo array, all'interno del metodo **resize()**
    - il ridimensionamento viene fatto ogni **n** operazioni
  - cerchiamo di valutare il ***costo medio*** di ciascuna operazione
    - tale metodo di stima si chiama ***analisi ammortizzata*** delle prestazioni asintotiche

99

Analisi ammortizzata

- Dobbiamo calcolare il valore medio di **n** operazioni, delle quali
  - **n-1** richiedono un tempo **O(1)**
  - **una** richiede un tempo **O(n)**

```
<T(n)> = [(n-1)*O(1) + O(n)]/n
       = O(n)/n = O(1)
```

- Distribuendo il tempo speso per il ridimensiomento in parti uguali a tutte le operazioni **push()**, si ottiene quindi ancora **O(1)**

100

Analisi ammortizzata

- Le prestazioni medie di **push()** con ridimensionamento rimangono **O(1)** per qualsiasi costante moltiplicativa usata per calcolare la nuova dimensione, anche diversa da 2
- Se, invece, si usa una costante ***additiva***, cioè la dimensione passa da **n** a **n+k**, si osserva che su **n** operazioni di inserimento quelle "lente" sono **n/k**

```
<T(n)>= [(n-n/k)*O(1)+(n/k)*O(n)]/n
      = [O(n) + n*O(n)]/n
      = O(n)/n + O(n)
      = O(1) + O(n) = O(n)
```

101

Pila realizzata con una catena

- Una pila può essere realizzata efficientemente anche usando una lista concatenata invece di un array
- Si noti che entrambe le estremità di una catena hanno, prese singolarmente, il comportamento di una pila
  - si può quindi realizzare una pila usando una delle due estremità della catena
  - è più efficiente usare l'***inizio*** della catena, perché le operazioni su tale estremità sono **O(1)**
    - removeFirst() è O(1)
    - removeLast() è O(n)!

102

```
public class LinkedListStack implements Stack
{  private LinkedList list;
   private int size;     // per contare gli elementi

   public LinkedListStack()      // O(1)
   {  makeEmpty();
   }

   public void makeEmpty()       //O(1)
   {  list = new LinkedList();
      size = 0;
   }

   public boolean isEmpty()      //O(1)
   {  return list.isEmpty();
   }

   public int size()             // O(1)
   {  return size;
   }
...
}
```

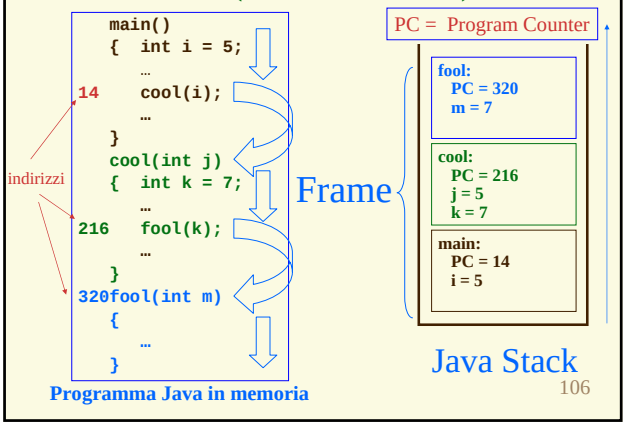
Pila realizzata con una catena

```
public class LinkedListStack implements Stack
{  ...
   public void push(Object obj)  // O(1)
   {  list.addFirst(obj);
      size++;
   }

   public Object top() throws EmptyStackException
   { if (isEmpty())              // O(1)
        throw new EmptyStackException();
     return   list.getFirst();
   }

   public Object pop() throws EmptyStackException //O(1)
   { Object obj = top();  // lascia a top() il lancio
     list.removeFirst();  // dell'eccezione
     size--;
     return obj;
   }
}
```


Pile nella Java Virtual Machine

- Ciascun programma java in esecuzione ha una propria pila chiamata **Java Stack** che viene usata per mantenere traccia delle **variabili locali, dei parametri formali dei metodi** e di altre importanti informazioni relative ai metodi, man mano che questi sono invocati
- Più precisamente, durante l'esecuzione di un programma, JVM mantiene uno **stack** i cui elementi sono **descrittori** dello stato corrente dell'invocazione dei metodi (che non sono terminati)
- I descrittori sono denominati ***Frame.*** A un certo istante durante l'esecuzione, ciascun metodo sospeso ha un ***frame*** nel ***Java Stack***

Java Stack (Runtime Stack)



Pile nella Java Virtual Machine

- Il metodo ***fool*** è chiamato dal metodo ***cool*** che a sua volta è stato chiamato dal metodo ***main***.
- Ad ogni invocazione di metodo in run-time viene inserito un frame nello stack
- Ciascun frame dello stack memorizza i valori del ***program counter***, dei ***parametri*** e delle ***variabili locali*** di una ***invocazione a un metodo***.
- Quando il metodo chiamato è terminato il frame viene estratto ed eliminato
- Quando l'invocazione del metodo ***fool*** termina, l'invocazione del metodo ***cool*** continuerà la sua esecuzione dall'istruzione di indirizzo 217, ottenuto incrementando il valore del program counter contenuto nel frame del metodo cool

Passaggio dei parametri ai metodi

- Il ***java stack*** pressiede al passaggio dei parametri ai metodi.
- Il meccanismo usa il passaggio dei ***parametri per valore***. Significa che il valore corrente di una variabile (o di un'espressione) è ciò che è passato al metodo.
- Nel caso di una variabile x di un ***tipo fondamentale del linguaggio***, come ***int*** o ***double***, il valore corrente di x è semplicemente il numero associato a x.
- Quando questo valore è passato a un metodo, esso è assegnato a una variabile locale nel frame del metodo chiamato.
- Se il metodo chiamato cambia il valore di questa variabile locale, non cambierà il valore della variabile nel metodo chiamante

Passaggio dei parametri per valore

```
    public static void main(String[] args)
    {
       int n = 7;
14     increment(n);
       System.out.println(n);
    }
216 private static void increment(int k)
    {
      k = k + 1;;
    }
```

7

Java Stack

increment:
PC = 216
k = 7 --- > k = 8

main:
PC = 14
n = 7

- Il valore di ***k*** nel frame del metodo increment è cambiato
- Il valore di ***n*** nel frame del metodo main non è cambiato

109

Passaggio dei parametri per valore

- Nel caso la variabile x sia una variabile riferimento ad un oggetto, il valore corrente di x è l'indirizzo in memoria dell'oggetto.
- Quando x viene passato come parametro, ciò che è copiato nel frame del metodo chiamato è l'indirizzo dell'oggetto in memoria.
- Quando questo indirizzo è assegnato a una variabile locale y nel metodo chiamato, y si riferirà allo stesso oggetto a cui si riferisce x.
- Se il metodo chiamato cambia lo stato interno dell'oggetto a cui y si riferisce, cambia anche lo stato dell'oggetto a cui x si riferisce perché è lo stesso oggetto.

110

Passaggio dei parametri per valore

```
    public static void main(String[] args)
    {
       BankAccount acct = new BankAccount(500);
14     sum(acct);
       System.out.println(acct.getBalance());
    }
216 private static void sum(BankAccount b)
    {
      b.deposit(1000);
    }
```

1500

Java Stack

sum:
PC = 216
b = acct

main:
PC = 14
acct = new BankAccount(500)

Java Heap

BankAccount
balance = 500 -- > 1500

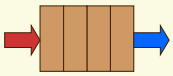
111

Lezione XXX
Gi 23-Nov-2006

ADT Coda (Queue)

112

ADT Coda (*queue*)



- In una ***coda*** (***queue***) gli oggetti possono essere inseriti ed estratti secondo un comportamento definito **FIFO** (***First In, First Out***)
  - il primo oggetto inserito è il primo a essere estratto
  - il nome è stato scelto in analogia con persone in ***coda***
- L'unico oggetto che può essere ispezionato è quello che verrebbe estratto
- Esistono molti possibili utilizzi di una struttura dati con questo comportamento
  - la simulazione del funzionamento di uno sportello bancario con più clienti che arrivano in momenti diversi userà una coda per rispettare la priorità di servizio

113

ADT Coda (*queue*)

- I metodi che caratterizzano una coda sono
  - **enqueue()** per inserire un oggetto nella coda
  - **dequeue()** per esaminare ed eliminare dalla coda l'oggetto che vi si trova da più tempo
  - **getFront()** per esaminare l'oggetto che verrebbe eliminato da **dequeue()**, senza estrarlo
- Infine, ogni ADT di tipo "contenitore" ha i metodi
  - **isEmpty()** per sapere se il contenitore è vuoto
  - **makeEmpty()** per vuotare il contenitore
  - **size()** per contare gli elementi presenti

114

ADT Coda (*queue*)

```
public interface Queue extends Container
{  void enqueue(Object obj);
   Object dequeue()  throws EmptyQueueException;
   Object getFront() throws EmptyQueueException;
}
```

- Si notino le similitudini con la pila
  - **enqueue()** corrisponde a **push()**
  - **dequeue()** corrisponde a **pop()**
  - **getFront()** corrisponde a **top()**

115

ADT Coda (*queue*)

```
public interface Queue extends Container
{  /**
    inserisce l'elemento all'ultimo posto della coda
    @param obj nuovo elemento da inserire
   */
   void enqueue(Object obj);

   /**
    rimuove l'elemento in testa alla coda
    @return elemento rimosso
    @throws EmptyQueueException se la coda e' vuota
   */
   Object dequeue() throws EmptyQueueException;

   /**
    restituisce l'elemento in testa alla coda
    @return elemento in testa alla coda
    @throws EmptyQueueException se la coda e' vuota
   */
   Object getFront() throws EmptyQueueException;
}
```

ADT Coda (*queue*)

- Per ***realizzare una coda*** si può usare una struttura di tipo ***array*** "riempito solo in parte", in modo simile a quanto fatto per realizzare una pila
- Mentre nella pila si inseriva e si estraeva allo stesso estremo dell'array (l'estremo "destro"), qui dobbiamo inserire ed estrarre ai due diversi estremi
  - decidiamo di inserire a destra ed estrarre a sinistra

117

ADT Coda (*queue*)

- Come per la pila, anche per la coda bisognerà segnalare l'errore di accesso a una coda vuota e gestire la situazione di coda piena (segnalando un errore o ridimensionando l'array)
- Definiamo
  - **EmptyQueueException** e **FullQueueException**

```
public class FullQueueException extends
   RuntimeException
{ }
```

```
public class EmptyQueueException extends
   RuntimeException
{ }
```

118

```
public class SlowFixedArrayQueue implements Queue
{
  private static final int CAPACITY = 100;
  private Object[] v;
  private int vSize;

  public SlowFixedArrayQueue()
  { makeEmpty();
  }

  public void makeEmpty()
  { v = new Object[CAPACITY];
    vSize = 0;
  }

  public boolean isEmpty()
  { return (vSize == 0);
  }

  public int size()
  { return vSize;
  }
  …
// continua
```

Coda realizzata con un array a dimensione fissa

```
  … // continua
  public void enqueue(Object obj)
  { if (vSize ==v.length)
       throw new FullQueueException();
    v[vSize++] = obj;
  }

  public Object getFront() throws EmptyQueueException
  { if (isEmpty())
       throw new EmptyQueueException();
    return v[0];
  }

  public Object dequeue() throws EmptyQueueException
  {
    Object obj = getFront(); // genera eccezione
    for (int i = 0; i < vSize - 1; i++)
       v[i] = v[i+1];
    vSize--;
    return obj;
  }
}
```

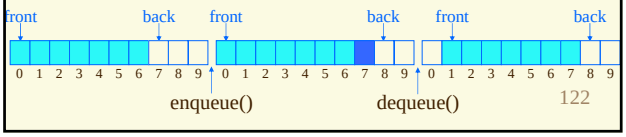
Coda (*queue*)

- Questa semplice realizzazione con array, che abbiamo visto essere molto efficiente per la pila, è al contrario assai inefficiente per la coda
  - il metodo **dequeue()** è `O(n)`, perché bisogna spostare tutti gli oggetti della coda per fare in modo che l'array rimanga "compatto"
  - la differenza rispetto alla pila è dovuta al fatto che nella coda gli inserimenti e le rimozioni avvengono alle due estremità diverse dell'array, mentre nella pila avvengono alla stessa estremità

121

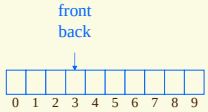
Coda (*queue*) su array con due indici

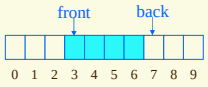
- Per realizzare una coda più efficiente servono ***due indici*** anziché uno soltanto
  - ***front:*** indice che punta al primo elemento nella coda
  - ***back:*** indice che punta al ***primo posto libero*** dopo l'ultimo elemento nella coda
  - quando front raggiunge back l'array è vuoto
- In questo modo, aggiornando opportunamente gli indici, si ottiene la realizzazione di una coda con un "***array riempito solo nella parte centrale***" in cui tutte le operazioni sono `O(1)`
  - la gestione dell'array pieno ha le due solite soluzioni, ridimensionamento o eccezione



122

```
public class FixedArrayQueue implements Queue
{
  static final int CAPACITY = 100;
  protected Object[] v;
  protected int front, back;

  public FixedArrayQueue()
  { makeEmpty();
  }

  public void makeEmpty()
  {  v = new Object[CAPACITY];
     front = back = 0;
  }

  public boolean isEmpty()
  { return (back == front);
  }

  public int size()
  {
    return back – front;
  }
  // continua
```





Coda (*queue*) su array con due indici

```
… // continua
  public void enqueue(Object obj)
     throws FullQueueException
  { if (back >= v.length)
        throw new FullQueueException();
    v[back++] = obj;
  }

  public Object getFront() throws EmptyQueueException
  { if (isEmpty())
        throw new EmptyQueueException();
    return v[front];
  }

  public Object dequeue() throws EmptyQueueException
  { Object obj = getFront();
    v[front] = null;  //garbage collector
    front++;
    return obj;
  }
}
```

Coda (*queue*)

- Per rendere la coda ridimensionabile, usiamo la stessa strategi a vista per la pila, estendendo la classe **FixedArrayQueue** e sovrascrivendo il solo metodo **enqueue()**

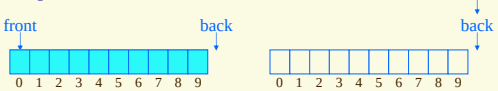
```
public class GrowingArrayQueue extends FixedArrayQueue
{
   public void enqueue(Object obj)
   {  if (back >= v.length)
         v = resize(v, 2 * v.length);
      super.enqueue(obj);
   }

   private static Object[] resize(Object[] a, int length)
   {...}
}
```

125

Prestazioni della coda realizzata con array

- La realizzazione di una coda con un array e due indici ha la massima efficienza in termini di prestazioni temporali, tutte le operazioni sono `O(1)`, ma ha ancora un punto debole
- Se l'array ha n elementi, proviamo a
  - effettuare n operazioni **enqueue()**

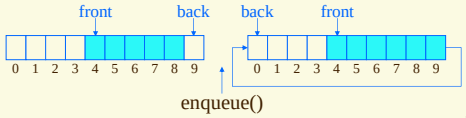
  e poi
  - effettuare n operazioni **dequeue()**
- Ora ***la coda è vuota***, ma alla successiva operazione **enqueue()** ***l'array sarà pieno***
  - lo spazio di memoria non viene riutilizzato efficientemente



126

Coda con array circolare

- Per risolvere quest'ultimo problema si usa una tecnica detta "*array circolare*"
  - i due indici, dopo essere giunti alla fine dell'array, possono ritornare all'inizio se si sono liberate delle posizioni
  - in questo modo l'array risulta pieno solo se la coda ha effettivamente un numero di oggetti uguale alla dimensione dell'array
  - le prestazioni temporali rimangono identiche



127

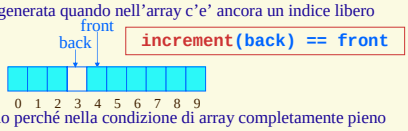
```
public class FixedCircularArrayQueue extends FixedArrayQueue
{ // il metodo increment() fa avanzare un indice di una
  // posizione, tornando all'inizio dell'array se si
  // supera la fine
  protected int increment(int index)
  {  return (index + 1) % v.length;
  }
  public void enqueue(Object obj) throws
    FullQueueException
  {  if (increment(back) == front)
        throw new FullQueueException();
     v[back] = obj;
     back = increment(back);
  }
  public Object dequeue() throws EmptyQueueException
  {  Object obj = getFront();
     v[front] = null;  // garbage collector
     front = increment(front);
     return obj;
  }
  public int size()
  {  return (v.length – front + back) % v.length; }
}
```

non serve sovrascrivere getFront() perché non modifica le variabili back e front

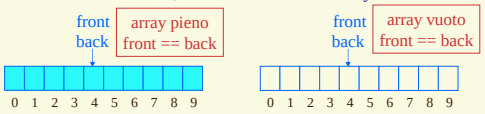
Osservazioni

```
public void enqueue(Object obj)
{  if (increment(back) == front)
      throw new FullQueueException();
   v[back] = obj;
   back = increment(back);
}
```

- L'eccezione viene generata quando nell'array c'e' ancora un indice libero



- Questo e' necessario perché nella condizione di array completamente pieno avremmo front == back, che è la condizione di array vuoto



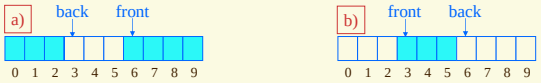
129

Osservazioni

```
public int size()
{ return (v.length – front + back) % v.length;
}
```

- Se back = front l'espressione ritorna 0; è corretto perchè questa è la condizione di array vuoto (si veda la discussione precedente)
- a) se back < front, il numero di elementi è pari a back più gli elementi fra front e la fine dell'array (v.length – front)

```
n = v.length – front + back < v.length =>
=> n = (v.length – front + back) % v.length
```

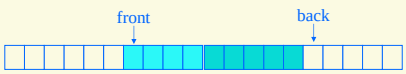


- b) se back > front, allora gli elementi sono back – front

```
n = back – front < v.length =>
=> n = (v.length – front + back) % v.length
```

130

```
public class GrowingCircularArrayQueue
   extends FixedCircularArrayQueue
{ public void enqueue(Object obj)
  { if (increment(back) == front)
    { v = resize(v, 2*v.length);
      // se si ridimensiona l'array e la zona utile
      // della coda si trova attorno alla sua fine,
      // la seconda metà del nuovo array rimane vuota
      // e provoca un malfunzionamento della coda,
      // che si risolve spostandovi la parte della
      // coda che si trova all'inizio dell'array
      if (back < front)
       {  System.arraycopy(v, 0, v, v.length/2, back);
          back += v.length/2;
       }
     }
     super.enqueue(obj);
  }
  private static Object[] resize(Object[] a, int length)
  {…}
}
```



Coda realizzata con una lista concatenata

- Anche una coda può essere realizzata usando una lista concatenata invece di un array
- È sufficiente inserire gli elementi a un'estremità della catena e rimuoverli all'altra estremità per ottenere il comportamento di una coda
- Perché tutte le operazioni siano **O(1)** bisogna ***inserire alla fine*** e ***rimuovere all'inizio***

132

Coda realizzata con una catena

```
public class LinkedListQueue implements Queue
{  private LinkedList list;
   private int size;

   public LinkedListQueue()
   {  makeEmpty();
   }

   public void makeEmpty()
   {  list = new LinkedList();
      size = 0;
   }

   public boolean isEmpty()
   {  return list.isEmpty();
   }

   public int size()
   {  return size;
   }
...
}
```

Coda realizzata con una catena

```
public class LinkedListQueue implements Queue
{  ...
   // O(1)
   public void enqueue(Object obj)
   {  list.addLast(obj);
      size++;
   }
   // O(1)
   public Object getFront() throws EmptyQueueException
   {  if (isEmpty())
         throw new EmptyQueueException();
      return list.getFirst();
   }
   // O(1)
   public Object dequeue() throws EmptyQueueException
   {  Object obj = getFront(); // lancia eccezione
      list.removeFirst();
      size--;
      return obj;
   }
}
```

Estrarre oggetti

- Le strutture dati generiche, definite in termini di **Object**, sono molto comode perché possono contenere oggetti di qualsiasi tipo
- Sono però un po' scomode nel momento in cui effettuiamo l'estrazione (o l'ispezione) di oggetti in esse contenuti
  - viene sempre restituito un riferimento di tipo **Object**, indipendentemente dal tipo di oggetto effettivamente restituito
  - si usa un forzamento per ottenere un riferimento del tipo originario

```
...
st.push("Hello");
Object obj = st.pop();
char c = obj.charAt(0);
...
```

```
...
st.push("Hello");
Object obj = st.pop();
String str = (String)obj;
char c = str.charAt(0);
```

```
ClassName.java:9: cannot find symbol
symbol  : method charAt(int)
location: class java.lang.Object
      char c = obj.charAt(0);
1 error          ^
```

'H'

135

Estrarre oggetti

```
String str = (String)st.pop();
```

- Sappiamo che serve il forzamento perché l'operazione di assegnamento è potenzialmente pericolosa
- Il programmatore si assume la responsabilità di inserire nella struttura dati oggetti del tipo corretto
- Cosa succede se è stato inserito un oggetto che NON sia di tipo **String**?
  - viene lanciata l'eccezione **ClassCastException**
- Possiamo scrivere codice che si comporti in modo più sicuro?

136

Estrarre oggetti

- Ricordiamo che le eccezioni, la cui gestione non è obbligatoria come **ClassCastException**, possono comunque essere gestite!

```
try
{  String str = (String)st.pop();
} catch (ClassCastException e)
{  // gestione dell'errore
}
```

- In alternativa si può usare l'operatore **instanceof**

```
Object obj = st.pop();
String str;
if (obj instanceof String)
   str = (String)obj;
else
   // gestione dell'errore
```

137

Strutture dati contenenti oggetti e numeri

138

Strutture dati di oggetti e di numeri

- Abbiamo visto che la pila e la coda come definite sono in grado di gestire dati di qualsiasi tipo, cioè riferimenti a oggetti di qualsiasi tipo (stringhe, conti bancari…)
- Non sono però in grado di gestire dati dei ***tipi fondamentali*** definiti nel linguaggio Java (**int**, **double**, **char**…)
  - tali tipi di dati NON sono oggetti
- Come possiamo gestire una struttura dati di numeri? Ad esempio una pila?

139

Pile di numeri

- Possiamo ridefinire tutto

```
public interface IntStack
      extends Container
{  void push(int obj);
   int top();
   int pop();
}
```

```
public class FixedArrayIntStack implements IntStack
{  protected int[] v;  protected int vSize;
   public FixedArrayIntStack()
   { makeEmpty(); }
   public void makeEmpty(){v=new int[100]; vSize =0;}
   public boolean isEmpty(){  return (vSize == 0); }
   public int size() { return vSize; }
   public void push(int obj) throws FullStackException
   {  if(vSize == v.length)
         throw new FullStackException();
      v[vSize++] = obj; }
   public int top() throws EmptyStackException
   {  if (isEmpty()) throw new EmptyStackException();
      return v[vSize - 1]; }
   public int pop()
   {  int obj = top(); vSize--; return obj; }
}
```

Pile di numeri

- La ridefinizione della pila per ogni tipo di dato fondamentale ha alcuni svantaggi
  - occorre replicare il codice nove volte (i tipi di dati fondamentali sono otto), con poche modifiche
  - non esiste più il tipo di dati astratto **Stack**, ma esisterà **IntStack**, **DoubleStack**, **CharStack**, **ObjectStack**
- Cerchiamo un'alternativa, ponendoci un problema
  - è possibile "trasformare" un numero intero (o un altro tipo di dato fondamentale di Java) in un oggetto?
- La risposta è affermativa
  - si usano le ***classi involucro*** (***wrapper***)

141

Classi involucro

- Per dati **int** esiste la ***classe involucro* Integer**
  - il costruttore richiede un parametro di tipo **int** e crea un oggetto di tipo **Integer** che contiene il valore intero, "avvolgendolo" con la struttura di un oggetto

```
Integer intObj = new Integer(2);
Object obj = intObj; // lecito
```

  - gli oggetti di tipo **Integer** sono ***immutabili***
  - per conoscere il valore memorizzato all'interno di un oggetto di tipo **Integer** si usa il metodo non statico **intValue**, che restituisce un valore di tipo **int**

```
int x = intObj.intValue(); // x vale 2
```

142

Classi involucro

- Esistono ***classi involucro*** per tutti i tipi di dati fondamentali di Java, con nomi uguali al nome del tipo fondamentale, ma iniziale maiuscola
  - eccezioni: **Integer** e **Character**
- Ogni classe fornisce un metodo per ottenere il valore contenuto al suo interno, con il nome corrispondente al tipo
  - es: ***booleanValue( )***, ***charValue( )***, ***doubleValue( )***
- Tutte le classi involucro si trovano nel pacchetto ***java.lang*** e realizzano l'interfaccia parametrica ***Comparable<T>***

143

Classi Involucro (Wrappers)

- **Boolean**
- **Byte**
- **Character**
- **Short**
- **Integer**
- **Long**
- **Float**
- **Double**

```
// tipo fondamentale del linguaggio
double x = 3.5;

// Oggetto che incapsula un tipo
// fondamentale
Double d = new Double(3.5);
```

```
Double d = new Double(3.5);
// d e' un oggetto di classe Double
```

144

Auto-boxing

- In Java 5.0, se un tipo fondamentale viene assegnato a una variabile della corrispondente classe involucro, viene creato automaticamente un oggetto della classe

```
Double d = 3.5;
// equivale a
Double d = new Double(3.5);
```

- Questo tipo di conversione automatica si indica col nome di ***auto-boxing***
- Non useremo l'auto-boxing, perché tende a confondere i tipi fondamentali con gli oggetti, mentre la loro natura in java è diversa

145

Esercizio: Controllo di parentesi

146

Esercizio: controllo parentesi

- Vogliamo risolvere il problema di verificare se in un'espressione algebrica (ricevuta come **String**) le parentesi tonde, quadre e graffe sono utilizzate in maniera corretta
- In particolare, vogliamo verificare che a ogni parentesi aperta corrisponda una parentesi chiusa dello stesso tipo
- Risolviamo prima il problema nel caso semplice in cui non siano ammesse parentesi annidate

147

Esercizio: controllo parentesi

ALGORITMO

- Inizializza la variabile **status** a **OUT** (vale **IN** quando ci si trova all'interno di una coppia di parentesi)
- Finché la stringa non è finita
  - leggi nella stringa il carattere più a sinistra non ancora letto
  - se è una parentesi aperta
    - se **status** è **OUT** poni **status** = **IN** e memorizza il tipo di parentesi
    - altrimenti errore (parentesi annidate...)
  - se è una parentesi chiusa
    - se **status** è **OUT** errore (parentesi chiusa senza aperta...)
    - altrimenti se corrisponde a quella memorizzata poni **status** = **OUT** (la parentesi è stata chiusa…)
    - altrimenti errore (parentesi non corrispondenti)
- Se **status** è **IN**, errore (parentesi aperta senza chiusa)

148

```
public static int checkWithoutNesting(String s)
{
  final boolean IN = true;
  final boolean OUT = false;

  boolean status = OUT;
  char bracket = '0'; // un valore qualsiasi
  for (int i = 0; i < s.length(); i++)
  {  char c = s.charAt(i);
     if (isOpeningBracket(c))
        if (status == OUT)
        {  status = IN;
           bracket = c;
        }
        else return 1; //Errore: parentesi annidate
     if (isClosingBracket(c))
        if (status == OUT) return 2;//Errore
        else if(areMatching(bracket, c))
           status = OUT;
        else return 3; //Errore      }
     if (status == IN) return 4; //Errore
     return 0; // OK
  }
```

Esercizio: controllo parentesi

```
/* verifica se e' una parentesi aperta
   @param c il carattere oggetto della verifica
   @return true se e' una parentesi aperta */
private static boolean isOpeningBracket(char c)
{  return c == '(' || c == '[' || c == '{';
}

/* verifica se e' una parentesi chiusa
   @param c il carattere oggetto della verifica
   @return true se e' una parentesi chiusa */
private static boolean isClosingBracket(char c)
{  return c == ')' || c == ']' || c == '}';
}

/* verifica se due parentesi sono corrispondenti */
private static boolean areMatching(char c1, char c2)
{  return c1 == '(' && c2 == ')' ||
          c1 == '[' && c2 == ']' ||
          c1 == '{' && c2 == '}';
}
```

Esercizio: Controllo parentesi

- Cerchiamo di risolvere il caso più generale, in cui le parentesi di vario tipo possono essere annidate

```
a + [ c + (g + h) + (f + z) ]
```

- In questo caso non è più sufficiente memorizzare il tipo dell'ultima parentesi che è stata aperta, perché ci possono essere più parentesi aperte che sono in attesa di essere chiuse
  - quando si chiude una parentesi, bisogna controllare se corrisponde al tipo della parentesi in attesa che è stata aperta ***più recentemente***

151

Esercizio: Controllo parentesi

- Possiamo quindi risolvere il problema usando una pila
- Effettuando una scansione della stringa da sinistra a destra
  - inseriamo nella pila le parentesi aperte
  - quando troviamo una parentesi chiusa, estraiamo una parentesi dalla pila (che sarà quindi l'ultima a essere stata inserita) e controlliamo che i tipi corrispondano, segnalando un errore in caso contrario
    - se ci troviamo a dover estrarre da una pila vuota, segnaliamo l'errore (parentesi chiusa senza aperta)
  - se al termine della stringa la pila non è vuota, segnaliamo l'errore (parentesi aperta senza chiusa)

152

```
public static int checkWithNesting(String s)
{
   Stack st = new GrowingArrayStack();
   for (int i = 0; i < s.length(); i++)
   {  char c = s.charAt(i);
      if (isOpeningBracket(c))
         st.push(new Character(c));
      else if (isClosingBracket(c))
      try
      {  Character ch = (Character)st.pop();
         char cc = ch.charValue();
         if (!areMatching(cc, c))
            return 3;
      }
      catch (EmptyStackException e)
      {  return 2; //Errore
      }
   }
   if (!st.isEmpty())
      return 4; //Errore
   return 0;
}
```