Lezione XXXIII
Lu 3-Dic-2007

ADT Dizionario

Dizionario

- Un ***dizionario*** è un ADT con le seguenti proprietà
  - è un contenitore
  - consente l'inserimento di ***coppie di dati*** di tipo
    - ***chiave / valore (attributo)***

    con la chiave che deve essere ***confrontabile*** e ***unica*** nell'insieme dei dati memorizzati
    - non possono esistere nel dizionario due valori con identica chiave
  - consente di effettuare in modo efficiente la ricerca e la rimozione di valori ***usando la chiave come identificatore***

Dizionario

- L'analogia con il dizionario di uso comune è molto forte
- In un comune dizionario
  - le chiavi sono le singole parole
  - l'attributo è la definizione della parola nel dizionario
  - tutte le chiavi sono distinte
  - a ogni chiave è associato uno e un solo valore
  - la ricerca di un attributo avviene tramite la sua chiave

Dizionario

```
public interface Dictionary extends Container
{   // l'inserimento va sempre a buon fine;
    // se la chiave non esiste, la coppia
    // key/value viene aggiunta al dizionario;
    // se la chiave esiste già, il valore ad
    // essa associato viene sovrascritto con
    // il nuovo valore
    void insert(Comparable key, Object value);

    // la rimozione della chiave rimuove anche
    // il corrispondente valore
    void remove(Comparable key);

    // la ricerca per chiave restituisce
    // soltanto il valore ad essa associato
    Object find(Comparable key);
}
```

Dizionario in un array

- Un ***dizionario*** può anche essere realizzato con un ***array***
  - ogni cella dell'array contiene un riferimento ad una coppia chiave/valore
    - un ***oggetto*** di tipo **Pair (*Coppia*)**
- Ci sono due strategie possibili
  - mantenere le chiavi ***ordinate*** nell'array
  - mantenere le chiavi ***non ordinate*** nell'array

Dizionario in un array ordinato

- Se le **n** chiavi vengono conservate ***ordinate*** nell'array
  - la **ricerca** ha prestazioni **O(log n)**
    - si può usare la ***ricerca per bisezione***
  - l'**inserimento** ha prestazioni **O(n)**
    - si usa l'ordinamento per ***inserzione in un array ordinato***
    - con altre strategie, occorre invece ordinare l'intero array, con prestazioni, in generale, **O(n log n)**
  - la **rimozione** ha prestazioni **O(n)**
    - bisogna fare una ricerca, e poi spostare ***mediamente*** **n/2** elementi per mantenere l'ordinamento

Dizionario in un array non ordinato

- Se le **n** chiavi vengono conservate ***disordinate*** nell'array
  - la **ricerca** ha prestazioni **O(n)**
    - bisogna usare la ***ricerca lineare***
  - l'**inserimento** ha prestazioni **O(n)**
    - Bisogna fare una ricerca. Se questa ha successo si sovrascrive la coppia, altrimenti si inserisce la nuova coppia all'ultimo posto dell'array, perche' l'ordine non interessa.
  - la **rimozione** ha prestazioni **O(n)**
    - bisogna fare una ricerca, e poi spostare nella posizione trovata l'ultimo elemento dell'array, perché l'ordinamento non interessa

Prestazioni di un dizionario

Dizionario	array ordinato	array non ordinato
ricerca	O(lg n)	O(n)
inserimento	O(n)	O(n)
rimozione	O(n)	O(n)

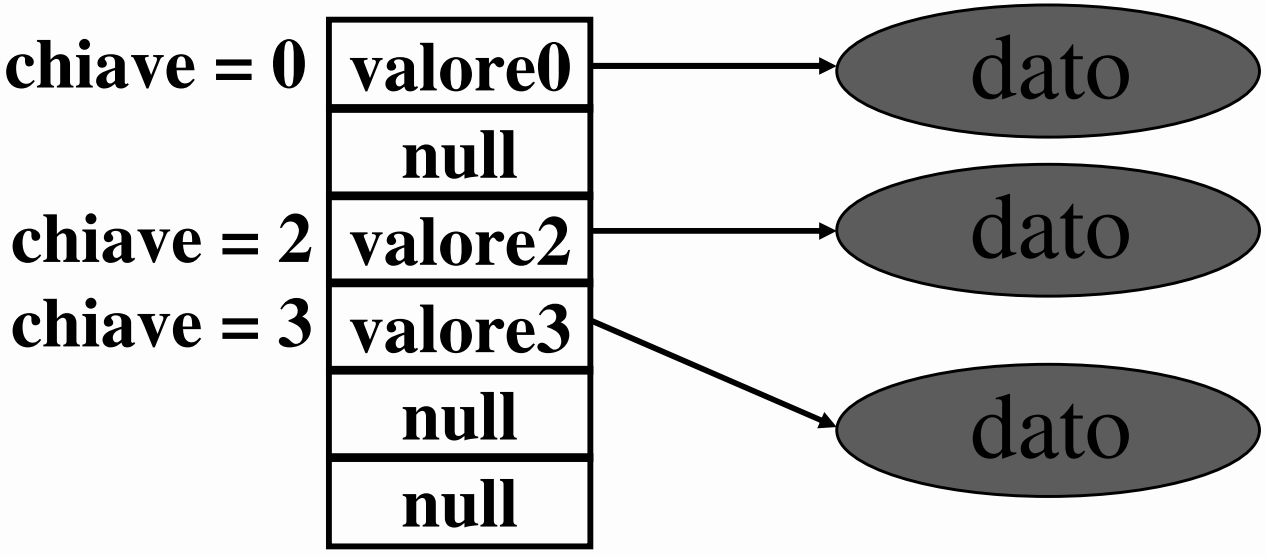
Chiavi numeriche

- Se imponiamo una restrizione al campo di applicazione di un dizionario
  - supponiamo che le chiavi siano ***numeri interi*** appartenenti a un intervallo noto a priori

  allora si può realizzare molto semplicemente un dizionario con prestazioni **O(1)** per tutte le operazioni
- Si usa un array che contiene soltanto i riferimenti ai valori, usando ***le chiavi come indici nell'array***
  - le celle dell'array che hanno come indice una chiave che non appartiene al dizionario hanno il valore **null**

Tabella

- Un dizionario con chiavi numeriche intere viene detto ***tabella*** o tavola (***table***)
- L'analogia è la seguente
  - un valore è una riga nella tabella
  - le righe sono numerate usando le chiavi
  - alcune righe possono essere ***vuote*** (senza valore)



Tabella

- Definiamo il tipo di dati astratto **Table** con un comportamento identico al dizionario
  - l'unica sostanziale differenza è che le chiavi non sono riferimenti a oggetti di tipo **Comparable**, ma sono ***numeri interi*** (che evidentemente sono confrontabili)

```
public interface Table extends Container
{  void insert(int key, Object value);
   void remove(int key);
   Object find(int key);
}
```

Tabella

```
public class ArrayTable100 implements Table
{  private Object[] v;
   private int count; //count rende isEmpty O(1)

   public ArrayTable100()
   {  makeEmpty(); }

   public void makeEmpty()
   {  count = 0;
      v = new Object[100];
   }
   public boolean isEmpty()
   {  return (count == 0); }

   private void check(int key)
   {  if (key < 0 || key >= v.length)
        throw new
           InvalidPositionTableException();
   }
    ...
```

```
public class ArrayTable100 implements Table
{  ...
   public void insert(int key, Object value)
   {  check(key);
      if (v[key] == null)
         count++;

      v[key] = value;
   }

   public void remove(int key)
   {  check(key);
      if (v[key] != null)
      {  count--;
         v[key] = null;
      }
   }

   public Object find(int key)
   {  check(key);
      return v[key];
   }
}
```

Tabella

- La tabella potrebbe avere ***dimensione variabile***, cioè utilizzare un ***array di dimensione crescente*** quando sia necessario
  - l'operazione di ***inserimento*** richiede, però, un tempo **O(n)** ogni volta che è necessario un ridimensionamento
  - in questo caso non si può utilizzare l'analisi ammortizzata perché non si può prevedere quali siano le posizioni richieste dall'utente
  - non è più vero che il ridimensionamento avviene "una volta ogni tanto", può avvenire anche tutte le volte
  - le prestazioni nel caso peggiore sono quindi **O(n)**

Tabella

- La tabella non utilizza la memoria in modo efficiente
  - l'occupazione di memoria richiesta per contenere **n** dati non dipende da **n** in modo lineare
    - come invece avviene per tutti gli altri ADT

    ma dipende dal contenuto informativo presente nei dati
    - in particolare, dal valore della chiave massima
- Può essere necessario un array di milioni di elementi per contenere poche decine di dati
  - si definisce ***fattore di riempimento*** (***load factor***) della tabella il numero di dati contenuti nella tabella diviso per la dimensione della tabella stessa

```
load factor = size() / length
```

Tabella

- La tabella è un dizionario con prestazioni ottime
  - tutte le operazioni sono **O(1)**

  ma con le seguenti limitazioni
  - le ***chiavi*** devono essere ***numeri interi*** (non negativi)
    - in realtà si possono usare anche chiavi negative, sottraendo ad ogni chiave il valore dell'estremo inferiore dell'intervallo di variabilità
  - ***l'intervallo di variabilità delle chiavi deve essere noto a priori***
    - per dimensionare la tabella
  - se il fattore di riempimento è molto basso, si ha un grande ***spreco di memoria***
    - ciò avviene se le chiavi sono molto "disperse" nel loro insieme di variabilità

Tabella

- Cerchiamo di eliminare una delle limitazioni
  - ***l'intervallo di variabilità delle chiavi deve essere noto a priori***
    - per dimensionare la tabella
- Risolviamo il problema in modo diverso
  - fissiamo la dimensione della tabella in modo arbitrario
    - in questo modo si definisce di conseguenza l'intervallo di variabilità delle chiavi utilizzabili
  - per usare chiavi esterne all'intervallo, si usa una ***funzione di trasformazione delle chiavi***
    - ***funzione di hash***

Funzione di hash

- Una funzione di hash ha
  - come ***dominio*** l'insieme delle chiavi che identificano univocamente i dati da inserire nel dizionario
  - come ***codominio*** l'insieme degli indici validi per accedere ad elementi della tabella
    - il risultato dell'applicazione della funzione di hash a una chiave si chiama ***chiave ridotta***
- Se manteniamo la limitazione di usare numeri interi come chiavi
  - una semplice funzione di hash è il calcolo del ***resto della divisione intera tra la chiave (k) e la dimensione della tabella (length)***

$$k_r = k \ \% \ length$$

Funzione di hash

- Per come è definita, la ***funzione di hash*** è generalmente ***non biunivoca***, cioè non è invertibile
  - ***chiavi diverse possono avere lo stesso valore per la funzione di hash***
- Per questo si chiama funzione di ***hash***
  - è una funzione che "***fa confusione***", nel senso che "mescola" dati diversi…
- In generale, non è possibile definire una funzione di hash biunivoca, perché la dimensione del dominio è maggiore della dimensione del codominio

Funzione di hash

- Il fatto che la funzione di hash non sia univoca genera un problema nuovo
  - inserendo un valore nella tabella, può darsi che la sua chiave ridotta sia uguale a quella di un valore già presente nella tabella e avente una ***diversa*** chiave, ma la stessa chiave ridotta
  - usando l'algoritmo già visto per la tabella, il nuovo valore andrebbe a sostituire il vecchio
    - questo non è corretto perché i due valori hanno, in realtà, chiavi diverse
  - questo fenomeno si chiama ***collisione*** nella tabella e si può risolvere in molti modi diversi
    - una tabella che usa chiavi ridotte si chiama ***tabella hash*** (***hash table***)

Risoluzione delle collisioni

- Quando si ha una collisione, bisognerebbe inserire il nuovo valore nella stessa cella della tabella (dell'array) che già contiene un altro valore
  - i due valori hanno chiavi diverse, ma la stessa chiave ridotta
  - ***ciascun valore deve essere memorizzato insieme alla sua vera chiave, per poter fare ricerche correttamente***
- Possiamo risolvere il problema usando una lista per ogni cella dell'array
  - l'array è un array di riferimenti a liste
  - ciascuna lista contiene le coppie ***chiave/valore*** che hanno la stessa chiave ridotta

Risoluzione delle collisioni

- Questo sistema di risoluzione delle collisioni si chiama ***tabella hash con bucket***
  - un ***bucket*** è una delle liste associate a una chiave ridotta

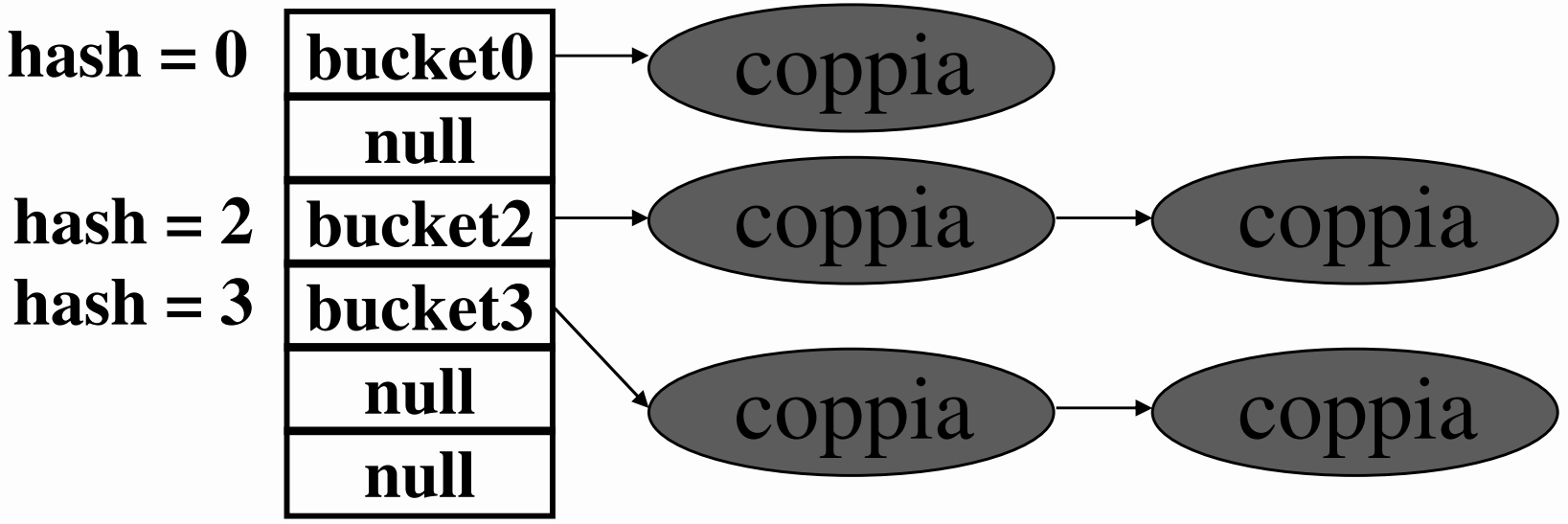


Tabella hash con bucket

- Le prestazioni della tabella hash con bucket non sono più, ovviamente, **O(1)** per tutte le operazioni
- ***Le prestazioni dipendono fortemente dalle caratteristiche della funzione di hash***
  - **caso peggiore**: la funzione di hash restituisce sempre la stessa chiave ridotta, per ogni chiave possibile
  - tutti i dati vengono inseriti in un'unica lista
    - le prestazioni della tabella hash degenerano in quelle di una lista
    - tutte le operazioni sono **O(n)**

Tabella hash con bucket

- **caso migliore**: la funzione di hash restituisce chiavi ridotte che si ***distribuiscono uniformemente*** nella tabella
  - tutte le liste hanno la stessa lunghezza media
    - se **M** è la dimensione della tabella
      - la lunghezza media di ciascuna lista è **n/M**
    - tutte le operazioni sono **O(n/M)**
  - per avere prestazioni **O(1)** occorre dimensionare la tabella in modo che **M** sia dello stesso ordine di grandezza di **n**

Tabella hash con bucket

- Riassumendo, in una ***tabella hash con bucket*** si ottengono prestazioni ottimali (tempo-costanti) se
  - ***la dimensione della tabella è circa uguale al numero di dati che saranno memorizzati nella tabella***
    - fattore di riempimento circa unitario
      - così si riduce al minimo anche lo spreco di memoria
  - ***la funzione di hash genera chiavi ridotte uniformemente distribuite***
    - liste di lunghezza quasi uguale alla lunghezza media
    - le liste hanno quasi tutte lunghezza uno!
- Se le chiavi vere sono uniformemente distribuite
  - la funzione di hash che "***calcola il resto della divisione intera***" genera chiavi ridotte uniformemente distribuite

Tabella hash
con chiave generica

Tabella hash con chiave generica

- Rimane da risolvere un solo problema nella tabella
  - le ***chiavi*** devono essere ***numeri interi*** (non negativi)
- Vogliamo cercare di realizzare una tabella hash con bucket che possa gestire coppie chiave/valore in cui ***la chiave*** non è un numero intero, ma ***un dato generico***
  - ad esempio, una stringa
  - applicazione
    - contenitore di oggetti di tipo **PersonaFisica**
    - chiave: ***codice fiscale***

Tabella hash con chiave generica

- Se vogliamo usare ***chiavi generiche*** (qualsiasi tipo di oggetto, in teoria…)
  - è sufficiente progettare una ***adeguata funzione di hash***
    - se la funzione di hash ha come dominio l'insieme delle chiavi generiche che interessano e come codominio un sottoinsieme dei numeri interi
      - cioè se ***le chiavi ridotte sono numeri interi***

      allora la tabella hash con bucket funziona correttamente senza modifiche

Funzione di hash per chiave generica

- ***Come si può trasformare una stringa in un numero intero?***
- Ricordiamo il significato della notazione posizionale nella rappresentazione di un numero intero

$$434 = 4 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$

- Ciascuna cifra rappresenta un numero che dipende
  - dal suo valore intrinseco
  - dalla sua posizione
- Possiamo usare la stessa convenzione per una stringa, dove ciascun carattere rappresenta un numero che dipende
  - dal suo valore intrinseco come carattere
    - nella codifica Unicode
  - dalla sua posizione nella stringa

Funzione di hash per chiave generica

- Possiamo usare la stessa convenzione per una stringa, dove ciascun carattere rappresenta un numero che dipende
  - dal suo valore intrinseco come carattere
    - nella codifica Unicode
  - dalla sua posizione nella stringa
- La base del sistema sarà il numero di simboli diversi (come nella base decimale), che per la codifica Unicode è 65536

$$\texttt{"ABC"} \Rightarrow \texttt{'A'} \cdot 65536^2 + \texttt{'B'} \cdot 65536^1 + \texttt{'C'} \cdot 65536^0$$

- In Java è molto semplice, perché un **char** è anche un numero intero...

Funzione di hash per stringa

"ABC" $\Rightarrow$ 'A'·65536^2 + 'B'·65536^1 + 'C'·65536^0

```
public int hashCode()
//ritorna un intero
{
   int final BASE = CHARACTER.MAX_VALUE + 1;
   int h = 0;
   for (int i = 0; i < length(); i++)
      h = BASE * h + charAt(i);

   return h;
}
```

e l'overflow?
Una volta tanto e' utile!

Funzione di hash per stringa

- La libreria standard usa, invece, per le stringhe la seguente funzione di hash

```
public int hashCode()
{
  final int HASH_MULTIPLIER = 31;
  int h = 0
  for (int i=0; i < length(); i++)
    h = HASH_MULTIPLIER * h + charAt(i);
  return h;
}
```

- Si usa un **numero primo** come moltiplicatore: simulazioni numeriche mostrano che questo migliora le proprieta' di uniformita' della funzione.

Attenzione l'hash code puo' essere negativo a causa dell'overfolw!

Funzione di hash per chiave generica

- Come fare per oggetti generici?
  - la classe **Object** mette a disposizione il metodo **hashCode()** che restituisce un **int** con buone proprietà di distribuzione uniforme
  - se il metodo **hashCode()** non viene ridefinito, viene calcolata una ***chiave ridotta*** a partire ***dall'indirizzo dell'oggetto in memoria***
  - l'esistenza di questo metodo rende possibile l'utilizzo di qualsiasi oggetto come chiave in una tabella hash
    - invocando **hashCode()** si ottiene un valore di tipo **int**
    - calcolando il resto della divisione intera del modulo di tale valore per la dimensione della tabella DIM, si ottiene finalmente la chiave ridotta kr

```
int hash = obj.hashCode() % DIM;
Attenzione  hash ∈ [-DIM+1, DIM-1]
int kr = hash >= 0 ? hash : -hash;
```

Funzione di hash per chiave generica

- Per calcolare la funzione di hash di un oggetto che abbia due variabili di esemplare:

  String nome; double d;

- Si fa generalmente come segue

```
public int hashCode()
{
  final int HASH_MULTIPLIER = 29;
  int h1 = nome.hashCode();
  int h2 = (new Double(d)).hashCode();
  int h = HASH_MULTIPLIER * h1 + h2;
  return h;
}
```

- HASH_MULTIPLIER e' come al solito un numero primo
- Se ci sono variabili di esemplare intere, si usa generalmente il numero intero nella generazione del hashCode

Tabella hash per chiave generica

```
public interface HashTable
       extends Container
{
  void insert(Object key, Object value);
  void remove(Object key);
  Object find(Object key);
}
```

Esercizio:
Controllo di parentesi

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

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)

```
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***

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)

```
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;
}
```

Lezione XXXIII
Ma 4-Dic-2007

Esercitazione

Esercitazione: Compito 10.1.2007

- Si veda il diario delle lezioni.

Esercitazione: Compito 24.9.2003

- Frequenza delle parole in un testo

- E' data una interfaccia D che definisce un dizionario di coppie composte da una ***stringa di caratteri*** (la ***chiave***) e da un ***attributo*** (un ***numero intero***).
- Le coppie sono definite della seguente classe

```
// Coppia.java -- coppie per il dizionario D
public class Coppia
{  public String ch;
   public int att;

   public Coppia(String s, int i)
   {
     ch = s;
     att = i;
   }
}
```

- mentre il dizionario è definito dalla seguente interfaccia:

```
// D.java --  dizionario
import java.util.NoSuchElementException
public interface Dizionario
{ /**
      inserisce la coppia <chiave, attributo>
      la chiave e' univoca*/
  void inserisci(String chiave, int attributo);

  /** restituisce vero se la chiave x e' presente, falso
      altrimenti  */
  boolean presente(String x);

  /* restituisce l'attributo associato alla chiave x se
     presente, altrimenti lancia l'eccezione
     java.util.NoSuchElementException */
  int cerca(String x) throws NoSuchElementException;

  /* estrae in un array, di lunghezza pari alle
     dimensioni del dizionario, il contenuto del
     dizionario */
  Coppia[] toArray();
}
```

Esercitazione

- Scrivere il codice di una classe D che realizza l'interfaccia Dizionario.
- Scrivere una classe "Main" di prova che, utilizzando un esemplare della classe D, esegua il conteggio della frequenza delle parole (per "parola" si intende una sequenza di caratteri separati da spazi o fineriga) presenti in un file di testo, utilizzando il seguente algoritmo

  - crea un esemplare d del dizionario vuoto
  - finche' ci sono parole in ingresso
    - estrai una parola
    - se la parola non è presente nel dizionario
      - inserisci in d la coppia <parola, 1>
    - altrimenti
      - inserisci nel dizionario la coppia <parola, n+1>
        essendo n l'attributo corrente di parola
  - estrai il contenuto del dizionario
  - trasferisci sull'uscita standard il valore delle coppie presenti.

Esercitazione

- Al termine della prova il candidato dovra' lasciare nella directory di lavoro i seguenti file:
  - Dizionario.java
  - Coppia.java la classe che realizza la coppia
  - D.java la classe che realizza il dizionario
  - Main.java il programma di prova per il conteggio della frequenza
- Durante la valutazione degli elaborati le classi sviluppate verranno provate eseguendo il conteggio delle parole presenti nel file che contiene il tema d'esame con il comando:
  - **`$java Main < compito.txt`**
- Si possono usare nello svolgimento dell'elaborato solo le seguenti classi della libreria standard:
  - tutte le classi del package java.io
  - le classi Scanner e NoSuchElementException del package java.util

Esercitazione: quale soluzione?

- Soluzioni possibili per il Dizionario

- Array non ordinato
  - inserimento O(n) con verifica di univocità della chiave
  - ricerca O(n)
  - verifica presenza O(n)

- Array Ordinato
  - inserimento O(n)
  - ricerca O(log n)
  - verifica presenza O(log n)

- Tabella hash con bucket
  - inserimento O(n/M)
  - ricerca O(n/M)
  - verifica presenza O(n/M)

Classe Coppia

```
// Coppia.java -- coppie per il dizionario D
public class Coppia
{
   public String ch;
   public int att;

   // costruttore
   public Coppia (String s, int i)
   {
      ch = s;
      att = i;
   }
}
```

Esercitazione: Soluzione con array non ordinato

Soluzione con array non ordinato

```
// D.java --  dizionario con array non ordinato
import java.util.NoSuchElementException;
public class DconArrayNonOrdinato implements Dizionario
{ private final int ARRAY_DIM = 1000;
  private Coppia[] v;    // array riempito parzialmente
  private int vSize;     // num. coppie nel dizionario

  //array di lunghezza fissa
  public DconArrayNonOrdinato()
  { v = new Coppia[ARRAY_DIM];
    vSize = 0;
  }

  public void inserisci(String chiave,      //O(n)
                        int attributo)
  { v[vSize] = new Coppia(chiave, attributo);
    int i;
    for (i = 0; !chiave.equals(v[i].ch); i++)
      ;
    if (i == vSize)
      vSize++;
    else
      v[i] = v[vSize];
  }
```

Sintassi valida perché le variabili di esemplare della classe Coppia sono pubbliche

Attenzione: non c'è ridimensionamento dinamico

```
  public boolean presente(String x)        //O(n)
  {  for (int i = 0; i < vSize; i++)
       if (x.equals(v[i].ch))
         return true;
     return false;
  }

  public int cerca(String x) throws        //O(n)
    NoSuchElementException
  {
    for (int i = 0; i < vSize; i++)
      if (x.equals(v[i].ch))
        return v[i].att;
    throw new NoSuchElementException();
  }

  public Coppia[] toArray() //ritorna copie! O(n)
  {
    Coppia[] tmp = new Coppia[vSize];
    for (int i = 0; i < vSize; i++)
      tmp[i] = new Coppia(v[i].ch,v[i].att); //copia
    return tmp;
  }
}
```

Copia di una Coppia!

Attenzione: restituire sempre *copie* dei dati interni, a meno che la loro classe non sia *immutabile* (nel nostro caso non e' immutabile!)

Incapsulamento

- Le variabili di esemplare della classe Coppia sono definite pubbliche!
  - è una violazione del principio dell'**incapsulamento**
  - è lecito: Java non obbliga a definire private le variabili di esemplare
  - ma rende il codice **"debole"**

- Classe Coppia: interna o pubblica?
  - la classe Coppia non può essere realizzata come una classe interna alla classe D, perchè il metodo **`toArray()`** ritorna un riferimento a un array di oggetti di classe Coppia; la classe Coppia deve essere quindi una classe accessibile all'esterno della classe D, quindi pubblica o con accessibilità di pacchetto.
  - in generale, quando una classe è usata solo all'interno di un'altra classe, è buona prassi definire la prima come classe interna

- I dati memorizzati nella classe D sono sicuri anche se la classe Coppia ha variabili di esemplare pubbliche?
  - Sì, perchè il metodo toArray() non restituisce i riferimenti agli oggetti di classe Coppia memorizzati internamente, ma restituisce un nuovo array di riferimenti a copie, a oggetti cioè diversi creati allo scopo che hanno lo stesso stato (stessi valori delle variabili di esemplare) degli oggetti del contenitore

Incapsulamento

- E' opportuno definire pubbliche le variabili di esemplare della classe Coppia?
  - **No**, il codice della classe Coppia è molto debole. Se ri-usata in altri contesti (in altre classi, che realizzano metodi che ritornano oggetti di classe Coppia) rende vulnerabili quelle classi.

- E allora, come realizzare la classe Coppia?

```
public class Coppia
{ private String ch;
  private int att;

  public Coppia(String s, int i)
  {  ch = s;
     att = i;
  }

  public String chiave() {return ch;}
  public int attributo() {return att;}
}
```

Il codice va riscritto così

```
public void inserisci(String chiave, int attributo)
{ v[vSize] = new Coppia(chiave, attributo);
  int i;
  for (i = 0; !chiave.equals(v[i].chiave()); i++)
    ;
  if (i == vSize)  vSize++;
  else  v[i] = v[vSize];
}
public boolean presente(String x)
{ for (int i = 0; i < vSize; i++)
    if (x.equals(v[i].chiave()))   return true;
  return false;
}
public int cerca(String x)
  throws NoSuchElementException
{ for (int i = 0; i < vSize; i++)
    if (x.equals(v[i].chiave()))
      return v[i].attributo();
  throw new NoSuchElementException();
}
```

Il codice va riscritto così

```
public Coppia[] toArray()
{
  Coppia[] tmp = new Coppia[vSize];

  for (int i = 0; i < vSize; i++)
    tmp[i] = new Coppia(v[i].chiave(), v[i].attributo());

  return tmp;
}
```

Incapsulamento

- Nel rispetto del principo dell'incapsulamento è accettabile anche la seguente realizzazione:

```
public class Coppia // -- classe immutabile
{ public final String ch;
  public final int att;

  public Coppia(String s, int i)
  {  ch = s;
     att = i;
  }
}
```

- Le variabili di esemplare **costanti** sono inizializzate dal costruttore al momento della creazione dell'esemplare e non possono essere modificate successivamente
- Nella classe non si possono, quindi, scrivere metodi modificatori.
- La classe è, allora, **immutabile**.

Come realizzare il metodo toArray()

- Nel realizzare metodi che restituiscono un array di oggetti, come il metodo toArray() della classe D, e' necessario usare due accorgimenti per rispettare il **principio dell'incapsulamento**:
  1. Non restituire **mai** il riferimento alla struttura in cui sono memorizzati i dati internamente. Non è impedito dal linguaggio (non vengono generati errori in complazione) ma viola il principio dell'**incapsulamento** e rende il codice molto debole!

```
public Coppia[] toArray()
{  v = resize(v, vSize)
   return v;
}
```

viola il principio dell'incapsulamento!

```
public Coppia[] toArray()
{  Coppia[] tmp = new Coppia[vSize];
     ...
   return tmp;
}
```

Così va bene!

Come realizzare il metodo toArray()

2 Non restituire **mai** i riferimenti ai dati interni, se questi sono oggetti, ma restituire una copia. E' sicuro restituire i riferimenti agli oggetti interni se questi sono esemplari di una classe immutabile.

```
public Coppia[] toArray() //ritorna copie! O(n)
{
   Coppia[] tmp = new Coppia[vSize];
   for (int i = 0; i < vSize; i++)
     tmp[i] = v[i];
   return tmp;
}
```

viola il principio dell'incapsulamento!

```
public Coppia[] toArray() //ritorna copie! O(n)
 {
   Coppia[] tmp = new Coppia[vSize];
   for (int i = 0; i < vSize; i++)
     tmp[i] = new Coppia(v[i].ch,v[i].att); //copia
   return tmp;
 }
```

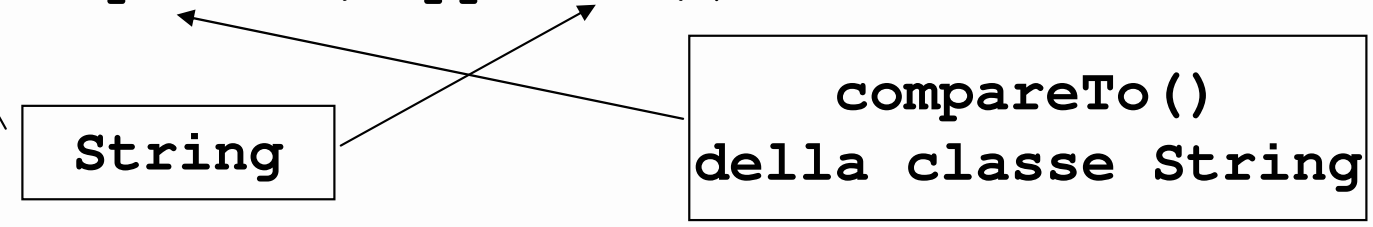
Così va bene!

Esercitazione: Soluzione con array ordinato

Soluzione con array ordinato

Rendiamo la classe Coppia comparabile

```
// Coppia.java -- coppie per il dizionario D
// Ordinamento naturale: per chiave
public class Coppia implements Comparable
{
   public final String ch;
   public final int att;

   public Coppia (String s, int i) {ch = s; att = i; }

   public int compareTo(Object obj)
   {
      Coppia coppia = (Coppia) obj;
      return ch.compareTo(coppia.ch);
   }
}
```



```
// DconArrayOrdinato.java -- dizionario con array ordinato
import java.util.NoSuchElementException;
public class DconArrayOrdinato implements Dizionario
{  private final int ARRAY_DIM = 1000;
   private Coppia[] v;
   private int vSize;

   public DconArrayOrdinato() //costruttore
   { v = new Coppia[ARRAY_DIM];
     vSize = 0;
   }
   // restituisce l'indice della coppia con chiave x
   private int trovaIndice(String x) // O(log n)
   {  int da = 0;
      int a = vSize - 1;
      Coppia coppia = new Coppia(x,1);//coppia da cercare
      while (da <= a) //bisezione iterativa
      {  int m = (da + a) / 2;
         if (coppia.compareTo(v[m]) == 0) //trovato
           return m;
         else if (coppia.compareTo(v[m]) < 0 )
           a = m -1; //cerco a sinistra
         else
           da = m + 1; //cerco a destra
      }
      return -1; //non trovato
   }
```

Soluzione con array ordinato

```
//O(n)
public void inserisci(String chiave, int attributo)
{
  int i = trovaIndice(chiave); //verifica presenza
  Coppia c = new Coppia(chiave, attributo);

  if (i == -1)
  { int j;
    for(j = vSize; j > 0 && c.compareTo(v[j-1]) < 0; j--)
      v[j] = v[j-1];

    v[j] = c;
    vSize++;
  }
  else
    v[i] = c;
}

public boolean presente(String x) // o(log n)
{
   int i = trovaIndice(x);
   return i != -1;
}
```

Soluzione con array ordinato

```
  //O(log n)
  public int cerca(String x) throws NoSuchElementException
  {
    int i = trovaIndice(x);
    if (i != -1)
      return v[i].att;

    throw new NoSuchElementException("Cerca " + x);
  }

  // O(n)
  public Coppia[] toArray() //O(n)
  {
     Coppia[] tmp = new Coppia[vSize];

     for (int i = 0; i < vSize; i++)
       tmp[i] = new Coppia(v[i].ch, v[i].att);

     return tmp;
   }
}
```

Lezione XXXV
Me 5-Dic-2007

Esercitazione: Soluzione con tabella hash con bucket

Soluzione con tabella hash

```
// D.java --  dizionario con tabella hash
import java.util.NoSuchElementException;

public class DconTabellaHash implements Dizionario
{
  // parte privata della classe
  private static final int HASHTABLE_DIM = 97;

  private ListNode[] v;
  private int size; //num. elementi nel dizionario

  public DconTabellaHash() //il costruttore
  {
    v = new ListNode[HASHTABLE_DIM];
    size = 0;

    //iniz. dei bucket con liste concatenate vuote
    for (int i = 0; i < HASHTABLE_DIM; i++)
      v[i] = new ListNode(null, null); //header
  }
```

```
//calcola l'hashCode della stringa String s
private static int hash(String s)
{  final int HASH_MULTIPLIER = 31;

   int h = 0;
   for (int i = 0; i < s.length(); i++)
   {
      h = HASH_MULTIPLIER * h + s.charAt(i);
      h = h % HASHTABLE_DIM;  // mantiene h fra 0
   }                          // HASHTABLE_DIM - 1

   return h;
}

//classe interna
class ListNode
{
   Coppia element;
   ListNode next;

   ListNode(Coppia e, ListNode n)
   {  element = e;
      next = n;
   }
}
```

Ritorna un intero fra zero e HASHTABLE_DIM – 1, compresi

Soluzione con tabella hash

```
/*
 se la chiave String x non e' presente, ritorna
 l'ultimo nodo del bucket, altrimenti ritorna il
 riferimento al nodo che precede quello contenente la
 chiave String x.
 NB: se la lista concatenata e' vuota ritorna
  l'header del bucket
*/
private ListNode trovaNodo(String x)
{
   ListNode tmp = v[hash(x)]; //header del bucket

   while (tmp.next != null)
     if ((tmp.next.element.ch).equals(x))
       break;
     else
       tmp = tmp.next;

   return tmp;
}
```

```
public void inserisci(String chiave, int attributo)
{  ListNode nodo = trovaNodo(chiave);
   Coppia nuovaCoppia = new Coppia(chiave, attributo);

   if (nodo.next != null)
      nodo.next.element = nuovaCoppia;
   else // inserimento in coda
   {  nodo.next = new ListNode(nuovaCoppia, null);
      size++;
   }
}

public boolean presente(String x)
{
   ListNode nodo = trovaNodo(x);
   return nodo.next != null;
}

public int cerca(String x) throws NoSuchElementException
{
  ListNode nodo = trovaNodo(x);
    if (nodo.next != null)
      return nodo.next.element.att;

   throw new NoSuchElementException();
}
```

Soluzione con tabella hash

```
 public Coppia[] toArray() // restituisce copie!!!
 {
   Coppia[] c = new Coppia[size];

   int j = 0;
   for (int i = 0; i < HASHTABLE_DIM; i++)
   {
     ListNode nodo = v[i];
     while (nodo.next != null)
     { // scansione dei nodi del bucket
       String chiave = nodo.next.element.ch;
       int attributo = nodo.next.element.att;
       Coppia copia = new Coppia(chiave, attributo);
       c[j++] = copia; //copia della Coppia

       nodo = nodo.next;
     }
   }

   return c;
 }
}
```

Lista Concatenata

- La realizzazione dei bucket che è stata presentata nell'esercizio è **sicura**?
  - **la risposta è sì**, perchè i bucket sono usati solo internamente alla classe DconTabellaHash
    - i metodi pubblici non restituiscono mai riferimenti a bucket e non hanno bucket come parametri espliciti

- In questa condizione, abbiamo potuto realizzare una versione della **lista concatenata** più semplice di quella presentata a lezione

Lista Concatenata

- quando è richiesto che sia una **classe pubblica**, allora deve essere realizzata come abbiamo fatto a lezione,
  - con la complicazione dell'iteratore, se è necessaria la funzionalità di accedere agli elementi centrali della lista
  - senza l'iteratore se si accede in testa e in coda (come nel caso della realizzazione dei contenitori Stack e Queue)

- Quando la lista concatenata può essere definita come **classe interna**, allora è sicuro scrivere il codice come presentato nell'esercizio

Classe di prova: Main

```
import java.util.Scanner;
import java.util.NoSuchElementException;

public class MainD
{ public static void main(String[] args)
  { Scanner in = new Scanner(System.in); //da std input

    Dizionario htd = new DconTabellaHash();

    while (in.hasNext())
    {
      String parola = in.next();

      if (!htd.presente(parola))
        htd.inserisci(parola, 1);
      else
        htd.inserisci(parola, htd.cerca(parola) + 1);
    }

    System.out.println("\nStampa da HashTableD");
    Coppia[] ar = htd.toArray();
    for (int i = 0; i < ar.length; i++)
      System.out.println(ar[i].ch + " " + ar[i].att );
  }
}
```

Riepilogo di ADT
Dizionario e Hash Table

ADT DIZIONARIO

```
public interface Dictionary extends Container
{   void insert(Comparable key, Object value);
    Object find(Comparable key);
    void remove(Comparable key);
}
```

Realizzata con	Complessita' Temporale		
	insert()	find()	remove()
Array non ordinato	O(n)	O(n)	O(n)
Array ordinato	O(n)	O(lgn)	O(n)
Lista Concatenata	O(n)	O(n)	O(n)

java.util.HashMap, interfaccia java.util.Map

ADT TABELLA

```
public interface Table extends Container
{  void insert(int key, Object value);
   Object find(int key);
   void remove(int key);
}
```

Realizzata con	Complessita' Temporale
Array	insert() O(1) find() O(1) remove() O(1)

Possibile uso non ottimale della memoria
fattore di riempimento (dati contenuti/dimensione tabella)

ADT HASH TABLE

```
interface HashTable extends Container
{  void insert(Object key, Object value);
   Object find(Object key);
   void remove(Object key);
}
```

Realizzata con	Complessita' Temporale
Array e Lista Concatenata (bucket)	insert() O(n/M)* find() O(n/M) remove() O(n/M)

* M dimensione della tabella
Nell'ipotesi che la funzione di hash sia uniforme

Interfaccia java.util.Map

Strutture dati: java.util.Hashtable, java.util.HashMap, java.util.LinkedHashMap

Il flusso di errore standard

Il flusso di errore standard

- Abbiamo visto che un programma Java ha sempre due flussi collegati
  - il flusso di ingresso standard: **System.in**
  - il flusso di uscita standard: **System.out**

  che vengono forniti dal sistema operativo

- In realtà esiste un altro flusso, chiamato ***flusso di errore standard*** o ***standard error***, rappresentato dall'oggetto **System.err**
  - **System.err** è di tipo **PrintStream** come **System.out**

```
public class StandardErrorTester
{ public static void main(String[] args)
  {
    System.out.println("Questo va allo standard output");
    System.err.println("Questo va allo standard error");
   }
}
```

Il flusso di errore standard

- La differenza tra **System.out** e **System.err** è solo convenzionale
  - si usa **System.out** per comunicare all'utente i risultati dell'elaborazione o qualunque altro messaggio che sia previsto dal corretto e normale funzionamento del programma
  - si usa **System.err** per comunicare all'utente eventuali condizioni di errore (fatali o non fatali) che si siano verificate durante il funzionamento del programma

Il flusso di errore standard

- In condizioni normali (cioè senza redirezione) lo ***standard error*** finisce ***sullo schermo*** insieme allo standard output
- In genere il sistema operativo consente di effettuare la redirezione dello standard error in modo indipendente dallo standard output
  - in Windows è possibile redirigere ***soltanto lo standard output***, mentre lo standard error rimane verso lo schermo
  - in Unix è possibile redirigere i due flussi verso due file distinti

Il flusso di errore standard

- In Linux, usando l'interprete dei comandi ***bash*** (per conoscere quale interprete dei comandi state usando, digitare da shell il comando: **`$echo $SHELL`**)

```
$ java StandardErrorTester
$Questo va allo standard output
$Questo va allo standard error
```

Senza redirezione

```
$ java StandardErrorTester > out.txt
$Questo va allo standard error
```

Redirezione di standard output

File out.txt

```
Questo va allo standard output
```

```
$ java StandardErrorTester 2> err.txt
$Questo va allo standard output
```

Redirezione di standard error

File err.txt

```
Questo va allo standard error
```

Il flusso di errore standard

- S può anche applicare la redirezione, aggiungerdo in coda a un file (modalità append) con i seguenti comandi dell'interprete ***bash***:

```
// redirezione di standard output
$ java StandardErrorTester >> out.txt
$ java StandardErrorTester >> out.txt
```

Aggiunge in coda al file out.txt

```
File out.txt
Questo va allo standard output
Questo va allo standard output
```

```
// redirezione di standard error
$ java StandardErrorTester 2>> err.txt
$ java StandardErrorTester 2>> err.txt
```

Aggiunge in coda al file err.txt

```
File err.txt
Questo va allo standard error
Questo va allo standard error
```

Cenni alla Programmazione Generica

Cenni alla Programmazione Generica

- In java per realizzare contenitori in grado di memorizzare oggetti di qualsiasi tipo esistono **due** meccanismi **Ereditarietà** e **Programmazione Generica**
- 1/ Ereditarietà
  - `Object[] v = new Object[...];`
  - è, ad esempio una struttura dati in cui possiamo inserire riferimenti a oggetti qualsiasi, perchè tutti gli oggetti derivano dalla classe generica Object attraverso il meccanismo dell'ereditarietà

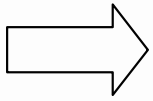
```
Object[] v = new Object[100];
String hello = "ciao"; // stringa
v[0] = hello;

Studente rossi = new Studente("rossi", "12345"); //stud.
v[1] = rossi;
```

  - Con questa tecnica abbiamo costruito i nostri contenitori come, ad esempio, ArStack e LinkedListStack

Cenni alla Programmazione Generica

- Quando si estraggono elementi dai contenitori, per invocare sull'oggetto i metodi specifici della classe cui appartiene si effettua una **conversione forzata**

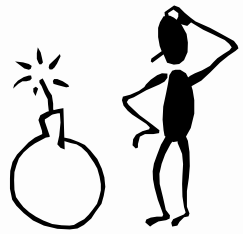
```
Stack s = new ArStack();
s.push("Hello, World!");
String str = (String) s.pop();
System.out.println(str.substring(7));
```



```
World!
```

- Se la conversione forzata è errata, in compilazione non viene segnalato alcun errore, ma in esecuzione viene generata l'eccezione **ClassCastException**

```
...
Stack s = new ArStack();
s.push(new Integer(1));
...
String str = (String) s.pop();
System.out.println(str.substring(7));
```

```
Class
Cast
Exception!
```

Cenni alla Programmazione Generica

- 2/ **Programmazione Generica java 5.0**
- Nella libreria standard sono definite delle classi generiche (Generics) che possono contenere oggetti qualsiasi ma tutti della stessa classe
- Ad esempio: java.util.Vector

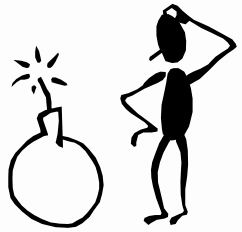
```
...
Vector<String> vStr = new Vector<String>();

Vector<Integer> vInt = new Vector<Integer>();
```

- Nella lista **`vStr`** possono essere inseriti solo riferimenti a oggetti di classe **`String`**
- Nella lista **`vInt`** possono essere inseriti solo riferimenti a oggetti di classe **`Integer`**
- La classe degli oggetti che possono essere contenuti viene scritta fra ***parentesi angolari***, dei tipi e dei costruttori.

Cenni alla Programmazione Generica

- Se in un contenitore particolarizzato per una certa classe si cerca di inserire un oggetto di classe diversa, l'errore viene segnalato dal compilatore

```
...
Vector<String> vStr = new Vector<String>();

Integer zero = new Integer(0);
vStr.add(zero);
...
```



```
cannot find symbol
symbol: java.util.Vector(java.lang.Integer)
location: java.util.Vector(java.lang.String)
        vStr.add(new Integer(0));
        ^
```

Scrivere classi generiche - Esempio

- Riscriviamo l'interfaccia Stack e la sua realizzazione nella classe ArrayStack, in modo che possa contenere solo esemplari di una classe specifica

```
public interface Stack<T> extends Container
{
   void push(T o);
   T top() throws EmptyStackException;
   T pop()throws EmptyStackException;
}
```

- T rappresenta un riferimento a un tipo generico, che verrà specificato quando si userà la struttura dati generica

Scrivere classi generiche

```
public class ArrayStack<T> implements Stack<T>
{
  private static int INITIAL_DIM = 1;
  private Object[] v; // array di Object!!!
  private int vSize;

  public ArrayStack()
  { makeEmpty();
  }

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

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

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

Scrivere classi generiche

```
 public void push(T o)
 {  if (vSize >= v.length)
      v = resize(v, 2 * v.length);

    v[vSize] = o;
    vSize++;
 }

 public T top() throws EmptyStackException
 {  if (isEmpty())
      throw new EmptyStackException();
    return (T) v[vSize - 1];
 }

 public T pop() throws EmptyStackException
 {  T tmp = top();
    v[vSize - 1] = null;
    vSize--;
    return tmp;
 }

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

Usare classi generiche

```
public class GenericsArrayStackTester
{
  public static void main(String[] args)
  { final char FIRST = 'A';
    int n = Integer.parseInt(args[0]); // numero passato

    // stack di riferimenti a Integer
    Stack<Integer> intStack = new ArrayStack<Integer>();

    // inserisco 0 1 2 3 ... n-1
    for (int i = 0; i < n; i++)
       intStack.push(new Integer(i));

    // Stack di riferimenti a Stinga
    Stack<String> stringStack = new ArrayStack<String>();

    // inserisco A AB ABC ABCD ...
    for (int i = 1; i < n; i++)
    { String tmpStr = "";
      for (int j = 0; j < i; j++)
        tmpStr = tmpStr + (char)(FIRST + j);
      stringStack.push(tmpStr);
    }
```

Usare classi generiche

```
    // invio a standard output gli Integer
    System.out.println("*** Integer ***");
    while (!intStack.isEmpty())
       System.out.println(intStack.pop());

    // invio a standard output le stringhe
    System.out.println("*** String ***");
    while (!stringStack.isEmpty())
      System.out.println(stringStack.pop());
   }
}
```

Esercitazione
ADT set

Insieme

- Il tipo di dati astratto "**insieme**" (***set***) è un contenitore (eventualmente vuoto) di oggetti **distinti** (cioè non contiene duplicati)
  - senza alcun particolare ordinamento
  - senza memoria dell'ordine temporale in cui gli oggetti vengono inseriti o estratti
  - si comporta come il corrispondente concetto matematico

Insieme

```
public interface Set extends Container
{ void add(Object obj);
  boolean contains(Object obj);
  Object[] toArray();
}
```

- Le operazioni consentite sull'insieme sono
  - **inserimento** di un oggetto
    - fallisce silenziosamente se l'oggetto è già presente
  - **verifica della presenza** di un oggetto
  - **ispezione di tutti gli oggetti**, mediante un metodo che restituisce un array di riferimenti agli oggetti contenuti nell'insieme, senza alcun requisito di ordinamento di tale array (anche perché i dati non sono ordinabili...)
- Non esiste un'operazione di **rimozione**
  - si usa la sottrazione tra insiemi (vedi in seguito)

interface java.util.Set

Operazioni sugli insiemi

- Per due insiemi A e B, si definiscono le operazioni
  - **unione** A ∪ B
    - appartengono all'unione di due insiemi tutti e soli gli elementi che appartengono ad almeno uno dei due insiemi
  - **intersezione** A ∩ B
    - appartengono all'intersezione di due insiemi tutti e soli gli elementi che appartengono ad entrambi gli insiemi
  - **sottrazione** A - B (oppure anche A \ B)
    - appartengono all'insieme sottrazione A-B tutti e soli gli elementi che appartengono all'insieme A e non appartengono all'insieme B
    - non è necessario che B sia un sottoinsieme di A

Realizzazione con array non ordinato

- Quando si scrive una classe, è comodo scrivere prima le
- intestazioni di tutti i metodi, con un corpo "***vuoto***"

```
public class ArraySet implements Set
{
   public void makeEmpty() { }
   public boolean isEmpty() { return true; }
   public int size() { return 0;}
   punlic void add(Object x) {  }
   public boolean contains(Object x) {return false; }
   public Object[] toArray() {return null; }
}
```

- Invece di compilare tutto alla fine, si compila ogni volta che si scrive il corpo di un metodo
  - in questo modo, si evita di trovarsi nella situazione in cui il compilatore segnala molti errori

Realizzazione con array non ordinato

```
public class ArraySet implements Set
{  private static int SET_DIM = 1;
   private Object[] v;
   private int vSize;

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

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

   public boolean contains(Object x) // O(n)
   {  for (int i = 0; i < vSize; i++)
        if (v[i].equals(x))
          return true;

      return false;
   }
…//continua
```

Realizzazione con array non ordinato

```
public class ArraySet implements Set
{  ...
   public Object[] toArray()      // O(n)
   {  Object[] x = new Object[vSize];
      for (int i = 0; i < vSize; i++)
        x[i] = v[i]; // non si puo' fare meglio!
      return x; }    // su un oggetto generico

// O(n) (usa contains)
   public void add(Object x)
   {  if (contains(x)) return;

      if (vSize >= v.length)
         v = resize(v, 2 * v.length);

      v[vSize++] = x;
   }
   private Object[] resize(Object[] a, int length)
   {...}

…//continua
```

Operazioni su insiemi: unione

```
public static Set union(Set s1, Set s2)
{   Set x = new ArraySet();
    // inseriamo gli elementi del primo insieme
    Object[] v = s1.toArray();
    for (int i = 0; i < v.length; i++)
        x.add(v[i]);
    // inseriamo tutti gli elementi del
    // secondo insieme, sfruttando le
    // proprietà di add (niente duplicati...)
    v = s2.toArray();
    for (int i = 0; i < v.length; i++)
        x.add(v[i]);
    return x;
}
```

- Se **contains** è **O(n)** (e, quindi, lo è anche **add**()), questa operazione è **O(n^2)**

Insiemi: intersezione e sottrazione

```
public static Set intersection(Set s1, Set s2)
{   Set x = new ArraySet();
    Object[] v = s1.toArray();
    for (int i = 0; i < v.length; i++)
        if (s2.contains(v[i]))
            x.add(v[i]);
    return x;
}                                          // O(n²)
```

```
public static Set subtract(Set s1, Set s2)
{   Set x = new ArraySet();
    Object[] v = s1.toArray();
    for (int i = 0; i < v.length; i++)
        if (!s2.contains(v[i]))
            x.add(v[i]);
    return x;
}                                          // O(n²)
```

Insieme con array non ordinato

- Riassumendo, realizzando un insieme con un array non ordinato
  - le prestazioni di tutte le primitive dell'insieme sono **O(n)**
  - le prestazioni di tutte le operazioni che agiscono su due insiemi sono **$O(n^2)$**
- Si può facilmente verificare che si ottengono le stesse prestazioni realizzando l'insieme con una catena (prestazioni determinate dalla ricerca)

Insieme di dati ordinati

- Cerchiamo di capire se si possono ottenere prestazioni migliori quando l'insieme contiene dati ordinabili
- Definiamo l'interfaccia "insieme ordinato"

```
public interface SortedSet extends Set
{  void add(Comparable obj);
   Comparable[] toSortedArray();
}
```

```
public interface Set extends Container
{  void add(Object obj);
   boolean contains(Object obj);
   Object[] toArray();
}
```

Insieme di dati ordinati

- Realizziamo l'interfaccia **SortedSet** usando un array ordinato
  - dovremo definire due metodi **add**(), uno dei quali *impedisce l'inserimento di dati non ordinabili*

```
public interface SortedSet extends Set
{  void add(Comparable obj);
   Comparable[] toSortedArray();
}
```

```
public interface Set extends Container
{  void add(Object obj);
   boolean contains(Object obj);
   Object[] toArray();
}
```

Insieme con array ordinato

```
public class ArraySortedSet implements SortedSet
{ private static int INITIAL_DIM = 1;
  private Comparable[] v;
  private int vSize = 0;

  public ArraySortedSet() // costruttore
  { makeEmpty(); }
  public void makeEmpty()
  { v = new Comparable[INITIAL_DIM];
    vSize = 0;
  }
  public boolean isEmpty() { return vSize == 0; }
  public int size() { return vSize; }

  public Comparable[] toSortedArray() // O(n)
  { Comparable[] x = new Comparable[vSize];
    System.arraycopy(v, 0, x, 0, vSize);
    return x;
  }
  public Object[] toArray()
  { return toSortedArray();
  }
```

Insieme con array ordinato

```
public boolean contains(Object x)  // O(log n)
{ Comparable target = (Comparable) x;
  int i = binSearch(v, 0, vSize - 1, target);
  return i != -1;
}

public void add(Object x) // non deve essere usato!
{ throw new IllegalArgumentException();
}

public void add(Comparable x)       // O(n)
{ if (contains(x))
     return;
  if (vSize >= v.length)
     v = resize(v, 2 * vSize);

  int j;  // inserimento in array ordinato!
  for (j=vSize; j>0 && x.compareTo(v[j - 1]) < 0; j--)
     v[j] = v[j - 1];
  v[j] = x;
  vSize++;
}
```

Operazioni su insiemi ordinati

- Gli algoritmi già visti per le operazioni sugli insiemi generici possono essere utilizzati senza alcuna modifica anche per l'insieme ordinato, realizzato con un array ordinato
  - infatti, un **SortedSet** è anche un **Set**
  - la complessità di tutti gli algoritmi (unione, intersezione e sottrazione) rimane $\mathbf{O(n^2)}$, perché il metodo **add** è rimasto **O(n)**
- Senza sfruttare le informazioni sull'ordinamento dell'insieme, non è possibile ottenere prestazioni migliori...

Operazioni su insiemi ordinati

- Usare l'incapsulamento a oltranza può essere sconveniente…
- In questo caso, sapendo che
  - l'array ottenuto con il metodo **toSortedArray** è ordinato
  - l'inserimento nell'insieme avviene nel metodo **add()** con l'algoritmo di ordinamento per inserzione in un array ordinato

  è possibile scrivere versioni più efficienti dei metodi già visti

Operazioni su insiemi: unione

- Per realizzare l'***unione***, osserviamo che il problema è molto simile alla ***fusione di due array ordinati***
  - come abbiamo visto in **MergeSort**, questo algoritmo di fusione è **O(n)**
- L'unica differenza consiste nella contemporanea eliminazione (cioè nel non inserimento…) di eventuali oggetti duplicati
  - un oggetto presente in entrambi gli insiemi dovrà essere presente una sola volta nell'insieme unione

Operazioni su insiemi: unione

- Effettuando la fusione dei due array ordinati secondo l'algoritmo visto in **MergeSort**, gli oggetti vengono via via inseriti nell'insieme unione che si va costruendo
- Se gli inserimenti avvengono con oggetti ***in ordine crescente***, l'ordinamento per inserzione in un array ordinato che viene usato dal metodo **add()** ha prestazioni **O(1)** per ogni inserimento!
  - il metodo **add()** ha quindi prestazioni **O(log n)**
    - perché invoca **contains**() che è **O(log n)**
  - se gli inserimenti non avvengono in ordine crescente, il metodo **add()** ha prestazioni medie di tipo **O(n)**

Insieme con array ordinato

```
public static SortedSet union(SortedSet s1,SortedSet s2)
{ SortedSet x = new ArraySortedSet();
  Comparable[] v1 = s1.toSortedArray();
  Comparable[] v2 = s2.toSortedArray();

  int i = 0, j = 0;
  while (i < v1.length && j < v2.length)
    if (v1[i].compareTo(v2[j]) < 0)
      x.add(v1[i++]); // inserisco da s1
    else if (v1[i].compareTo(v2[j]) > 0)
      x.add(v2[j++]); // inserisco da s2
    else // se sono uguali avanzo entrambi gli indici
    {  x.add(v1[i++]);
       j++;
    }
  while (i < v1.length) // se non e' finito, lo finisco
    x.add(v1[i++]);
  while (j < v2.length) // se non e' finito, lo finisco
    x.add(v2[j++]);
  return x;
} // prestazioni O(n log n) anziche'quadratiche
```

Insieme con array ordinato

```
public static SortedSet intersection(SortedSet s1,
  SortedSet s2)
{
  SortedSet x = new ArraySortedSet();

  Comparable[] v1 = s1.toSortedArray();
  Comparable[] v2 = s2.toSortedArray();

  int j = 0;
  for (int i = 0; i < v1.length; i++)
  {
     while (j < v2.length && v1[i].compareTo(v2[j]) > 0)
       j++;

     if (j != v2.length && v1[i].compareTo(v2[j]) == 0)
       x.add(v1[i]);
  }

  return x;
} // prestazioni O(n log n) anziché quadratiche
```

Insieme con array ordinato

```
public static SortedSet subtract(SortedSet s1,
  SortedSet s2)
{ SortedSet x = new ArraySortedSet();
  Comparable[] v1 = s1.toSortedArray();
  Comparable[] v2 = s2.toSortedArray();

  int j = 0;
  int i;
  for (i = 0; i < v1.length; i++)
  {  while (j < v2.length && v1[i].compareTo(v2[j]) > 0)
       j++;
     if (j != v2.length && v1[i].compareTo(v2[j]) != 0)
       x.add(v1[i]);
  }
  while(i < v1.length)
     x.add(v1[i++]);

    return x;
 } // prestazioni O(n log n) anziché quadratiche
```

Lezione XXXVI
Gi 6-Dic-2007

Riepilogo Java

F. Bombi modificato da A. Luchetta

Esercitazione: Compito 17.12.2002

- Si veda il diario delle lezioni.

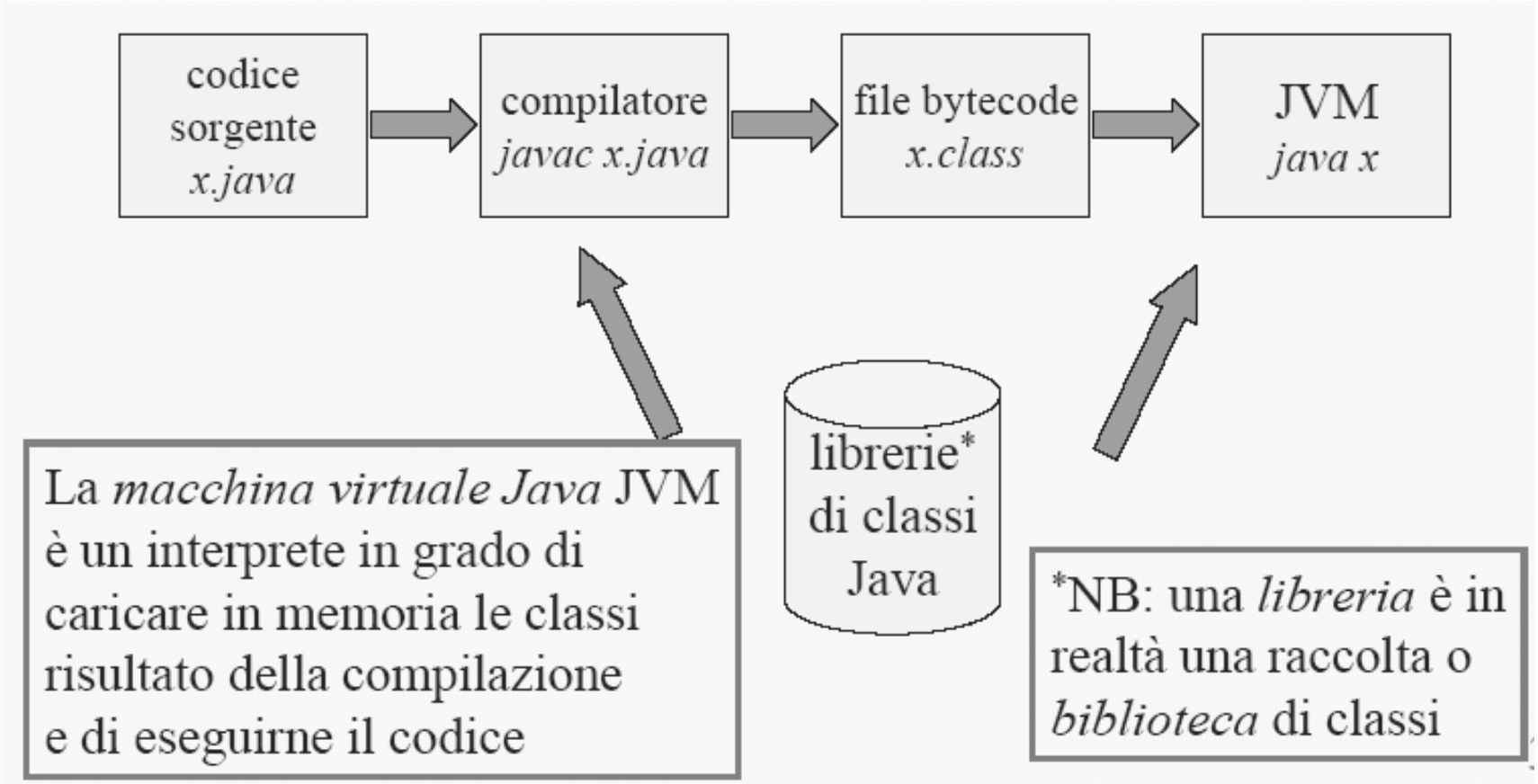
Java - introduzione

- Java è un linguaggio di impiego generale, basato su ***classi*** e ***orientato agli oggetti***
- Java è un linguaggio ***fortemente tipizzato*** in modo che sia possibile individuare molti errori al momento della compilazione
- Java è un linguaggio di ***alto livello*** che nasconde i dettagli della macchina che esegue un programma
- La gestione dinamica della memoria è affidata ad un ***garbage collector*** di elevata efficienza

La macchina virtuale Java

- Un programma scritto in linguaggio Java viene di solito tradotto, all'atto della compilazione, in un insieme di istruzioni definite dalla macchina virtuale Java (JVM)
- Il codice binario della JVM, detto ***bytecode***, viene poi eseguito per mezzo di un ***interprete*** che ha anche la funzione di caricare in memoria le classi delle librerie standard utilizzate dal programma
- Noi abbiamo studiato solo la realizzazione di applicazioni cioè di programmi da avviare con un comando del tipo:
  - ***`java nomeClasse parametri`***

Compilazione ed esecuzione di un programma in Java



Unità compilabili e pacchetti

- Un programma Java è composto da un insieme di ***pacchetti*** (***package***) organizzati gerarchicamente
- Ogni pacchetto è a sua volta costituito da una o più ***unità compilabili*** cioè file individuati da un nome con l'estensione .java
- Più unità compilabili contenute in una cartella costituiscono, in assenza di una dichiarazione esplicita, un ***pacchetto senza nome***
- Un'unità di compilazione può importare in modo esplicito classi da altri pacchetti, il pacchetto **`java.lang`** viene sempre importato implicitamente

Grammatiche

- Java è un linguaggio formale definito da un grammatica non contestuale (o quasi)
- La ***grammatica*** può essere divisa in due parti che descrivono rispettivamente le ***regole lessicali*** e la ***sintassi*** che governano la costruzione di unità compilabili sintatticamente corrette
- Dal punto di vista lessicale un programma Java è composto da ***righe*** (stringhe di caratteri terminate da un fineriga) a loro volta composte da unità elementari dette ***token*** (in italiano lessemi)
- I token devono essere separati da spazi e/o da commenti quando non ci sia un separatore o un operatore che marchi la fine del token

I codici Unicode e ASCII

- I programmi Java sono scritti usando l'insieme di caratteri Unicode
- Unicode utilizza, per rappresentare un carattere, un numero intero senza segno di 16 bit e definisce circa 64.000 caratteri in molti alfabeti diversi
- I primi 128 codici Unicode coincidono con l'insieme di caratteri Basic Latin noto anche come ASCII (American Standard Code for Information Interchange)
- Parole chiave, caratteri di interpunzione e operatori sono definiti con i soli caratteri ASCII
- Noi abbiamo utilizzato solo i caratteri ASCII anche per definire nomie quindi possiamo dimenticarci di Unicode

Token

- I token (le unità lessicali con cui si costruisce una unità compilabile) possono essere un:
  - ***Identificatore*** (identifier): una stringa di caratteri alfanumerici con il primo carattere alfabetico (anche '_' e' ammesso come primo carattere)
  - ***Parola chiave*** (keyword): stringhe di caratteri il cui uso è riservato e predefinito dal linguaggio Java
  - ***Costante*** (literal): un valore costante
  - ***Separatore***: caratteri di interpunzione
  - ***Operatore***: simboli che indicano un'operazione fra variabili e/o costanti

Le parole chiave

- **`abstract assert boolean break byte case catch char class` *`const`* `continue default do double else extends final` *`finally`* `float for` *`goto`* `if implements import instanceof int interface long` *`native`* `new package private protected public return short static super switch` *`synchronized`* `this throw throws` *`transient`* `try void` *`volatile`* `while`**

- Alcune parole chiave non sono mai usate, ***di altre noi abbiamo fatto mai uso***, è comunque necessario saperne l'esistenza per non cercare di utilizzare una parola chiave come identificatore(vedi Horstmann)

I paradigmi di programmazione

I paradigmi di programmazione

- Un programma deve essere comprensibile sia al compilatore (e alla macchina) sia all'uomo

- È importante utilizzare ***paradigmi*** (modelli) di programmazione che aiutino a scrivere programmi ***corretti*** e ***facili da mantenere e modificare***

- Nel tempo si sono sviluppati molti paradigmi diversi nel tentativo di rendere più facile la produzione di programmi corretti e mantenibili

La decomposizione funzionale

- Per decomposizione funzionalesi intende la tecnica con la quale si risolve un problema attraverso la ***composizione di sottoprogrammi o funzioni***
- Ad esempio dovendo leggere ed elaborare dei dati si scrive
  - ***un sottoprogramma*** che legge i dati,
  - ***uno*** che gli elabora
  - ***un terzo*** che stampa i risultati
- Il primo linguaggio che ha messo a disposizione strumenti per facilitare la decomposizione funzionale è stato il ***Fortran IV***(fine anni ’50)

La programmazione strutturata

- Un programma si dice strutturato se è realizzato dalla composizione delle due sole strutture (o loro derivazione) **`if- then - else,`** `while-do`
- I linguaggi di programmazione moderni (sviluppati dopo l'invenzione del ***Pascal***, inizio anni '70) sono per loro natura strutturati e quindi obbligano ad utilizzare naturalmente la programmazione strutturata
- L'idea è così connaturata con i linguaggi moderni quali il C/C++ e Java che non è quasi più il caso di parlare di programmazione strutturata

"Go To Statement Considered Harmful"
E. W. Dijkstra, Comm. ACM, 11, 1968, 147

Modularizzazione

- Per affrontare la costruzione di un grande progetto software è necessario disporre di strumenti che consentano di costruire il software sotto forma di componenti indipendenti detti talvolta ***moduli***
- Pascal (in origine) non consentiva nessuna forma di modularizzazione, un programma doveva essere sempre pensato come monolitico in quanto doveva contenere al suo interno tutte le procedure e funzioni necessarie
- Il linguaggio C non dispone intrinsecamente di strumenti per la modularizzazione ma non la impedisce, è quindi possibile realizzare in C software modulare utilizzando le funzioni di ***macro elaborazione*** fornite dal linguaggio e strumenti esterni quali ***make*** per automatizzare le operazioni di espansione delle macro, compilazione e collegamento

Modularizzazione(segue)

- I linguaggi Ada e Modula 2 sono stati progettati in modo da facilitare e rendere controllabile la modularizzazione.
- Hanno però avuto uno sviluppo limitato, il primo solo nell'ambiente delle commesse militari e spaziali, il secondo solo in un limitato ambiente accademico
- Alcune estensione del linguaggio C quali C++ e in particolar modo il linguaggio orientato agli oggetti Eifell sono stati pensati in modo da facilitare la modularizzazione

Programmazione orientata agli oggetti

- I linguaggio orientati agli oggetti mettono a disposizione una ricco repertorio di strumenti per la realizzazione di software in forma modulare
  - package
  - incapsulamento (o information hiding)
  - polimorfismo
  - ereditarietà

Packages-> Pacchetti

- Un programma in Java è organizzato come un insieme di pacchetti
- Ogni pacchetto ha un suo insieme di nomi per i ***tipi*** (classi e interfacce)
- Un *tipo* dichiarato in un pacchetto è accessibile al di fuori del pacchetto in cui è stato dichiarato solo se ha l'attributo ***`public`***
- I pacchetti sono organizzati in forma gerarchica come pacchetti e sottopacchetti
- I pacchetti possono essere memorizzati come file o in un database
- L'organizzazione in pacchetti facilita la modularizzazione isolando le scelte dei nomi di un pacchetto da quelle di ogni altro pacchetto

Incapsulamento

- Per incapsulamento o information hiding si intende la caratteristica di un linguaggio che consente di nascondere all'utente di un pacchetto (o anche in particolare di una sola classe) i dettagli con cui le funzionalità sono realizzate
- Java consente di progettare pacchetti e classi in modo da nascondere in modo completo i dettagli realizzativi all'utente

Polimorfismo

- Il polimorfismo è la proprietà di un linguaggio orientato ad oggetti per cui la decisione di quale metodo viene invocato tramite un riferimento viene stabilito al momento dell'esecuzione del programma in funzione del valore effettivamente assegnato al riferimento (in sintesi si parla di ***late binding***)
- Java è intrinsecamente polimorfo
- Una forma elementare di polimorfismo è anche offerta dal ***sovraccarico*** (***overloading***) ***del nome*** di un metodo.
- Notare che il sovraccarico non richiede il late binding in quanto il compilatore può decidere quale metodo invocare dal confronto della forma della chiamata con l'intestazione del metodo (nome e elenco del tipo degli argomenti)

Ereditarietà

- L'ereditarietà consente di costruire una nuova classe che estende le funzionalità di un'altra classe senza avere accesso al codice sorgente della classe che si vuole estendere
- Ereditarietà e polimorfismo sono funzionalità da utilizzare in modo coordinato
- La programmazione orientata ad oggetti si caratterizza dalla possibilità di costruire l'estensione di una classe senza disporre del codice sorgente della classe da estendere combinata con la realizzazione del polimorfismo mediante late binding

Ultime Informazioni sull'esame

Modalità d'esame

- L'esame si articola in tre prove;
  - questionario a risposte multiple
    - (circa 50 domande, 55 minuti)
  - esercizio di programmazione in laboratorio
    - (circa 2 ore)
  - prova orale
    - (circa 20 minuti)

- Nella valutazione finale in trentesimi le tre parti hanno peso pressoché uguale.

- Il candidato può sostenere l'esame in ciascun appello.
  - tutte e tre le prove ***devono*** essere sostenute nello stesso appello.

Questionario a risposte multiple

I appello: Gi 13-Dic-2007 in ADT
II appello: Ma 8-Genn-2008 in ADT

- Organizzato in più turni
- Verificare i turni nella bacheca elettronica il giorno precedente alla prova
- Consultare le bacheche elettroniche del DEI
- (**avvisi urgenti**) e la home page del corso

numero domande: ~50

tempo: 50 - 60 min

svolgimento a terminale

Questionario a risposte multiple

- **Punteggio grezzo**
  - risposta corretta: 1 punto
  - risposta errata: 0 punti
  - risposta non data: ***1/n*** punti dove n è il numero di alternative della domanda
    - meglio una risposta non data che una errata!

- **Voto in trentesimi**
- punteggio grezzo riportato in trentesimi con arrotondamento e qualche fattore correttivo
  - **punti 28.50 / 50 ~ 57% → 18 /30**
  - **punti 48.50 / 50 ~ 97% → 30 /30**
- Per essere ammessi alla prova di programmazione è necessario conseguire una votazione non inferiore a 18/30 nel questionario a risposte multiple

Prova di programmazione

I appello: Ve 14-Dic-2007 in ADT
II appello: Me 9-Genn-2008 in ADT

- Organizzato in più turni
- Verificare i turni nella bacheca elettronica il giorno precedente (avvisi urgenti per gli studenti!!!)

Tempo concesso per la prova: ~ 2 ore

svolgimento a terminale

La prova pratica viene, di norma, valutata solo se la soluzione proposta dal candidato compila senza errori.

Programmazione

- L'obiettivo principale della prova di programmazione è quello di realizzare un programma
  - che rispetti le specifiche (senza fare cose in più…)
  - che funzioni
- Come obiettivi secondari
  - che sia efficiente
  - che rispetti regole stilistiche
  - che gestisca coerentemente gli errori
  - che contenga commenti

Programmazione

- Consigli:
  - concentrarsi prima sulla realizzazione di un ***prototipo funzionante nel rispetto delle specifiche***
    - di solito, la cosa migliore è seguire le specifiche passo per passo
    - un programma funzionante che non rispetti le specifiche non serve a niente...
  - verificare accuratamente il corretto funzionamento del programma realizzato, soprattutto in ***eventuali situazioni limite***
    - per velocizzare il collaudo, si consiglia di preparare alcuni file di input da usare con la redirezione, prevedendo l'output corretto

Programmazione

- Consigli:
  - dopo aver realizzato un prototipo funzionante, ***fare una copia del file*** in modo da poter rapidamente tornare alla situazione corretta
  - modificare il programma in vista di eventuali obiettivi secondari
    - ***dopo ogni modifica, effettuare nuovamente il collaudo***
    - se non si riesce a far funzionare la modifica, ripristinare la situazione corretta copiando il file

Programmazione

Il candidato deve produrre un elaborato che:

1 rispetti le specifiche

2 non generi errori in compilazione

Obiettivi minimi

che, *possibilmente*,

3 *non generi errori in esecuzione*

4 esegua correttamente, secondo le specifiche

In subordine, se rimane tempo, che

5 sia ottimo dal punto di vista della complessita' temporale dei metodi realizzati

6 sia propriamente commentato

Oggetto della prova

Il candidato deve generalmente realizzare un ADT:

- Pila
- Coda
- Lista
- Dizionario
- Tabella
- Hash table

Prova Orale

Circa 20 min

Domade su tutto il programma svolto

Il corso è finito

Università di Padova

Facoltà di Ingegneria

Laurea triennale
Corsi della classe 9
Ingegneria dell'Informazione

Fondamenti di Informatica 1

a.a. 2007 - 08

Canale 89

Grazie a
Marcello Dalpasso

Molte delle trasparenze del corso sono sue e da me modificate

Gli errori sono tutti miei pero'...