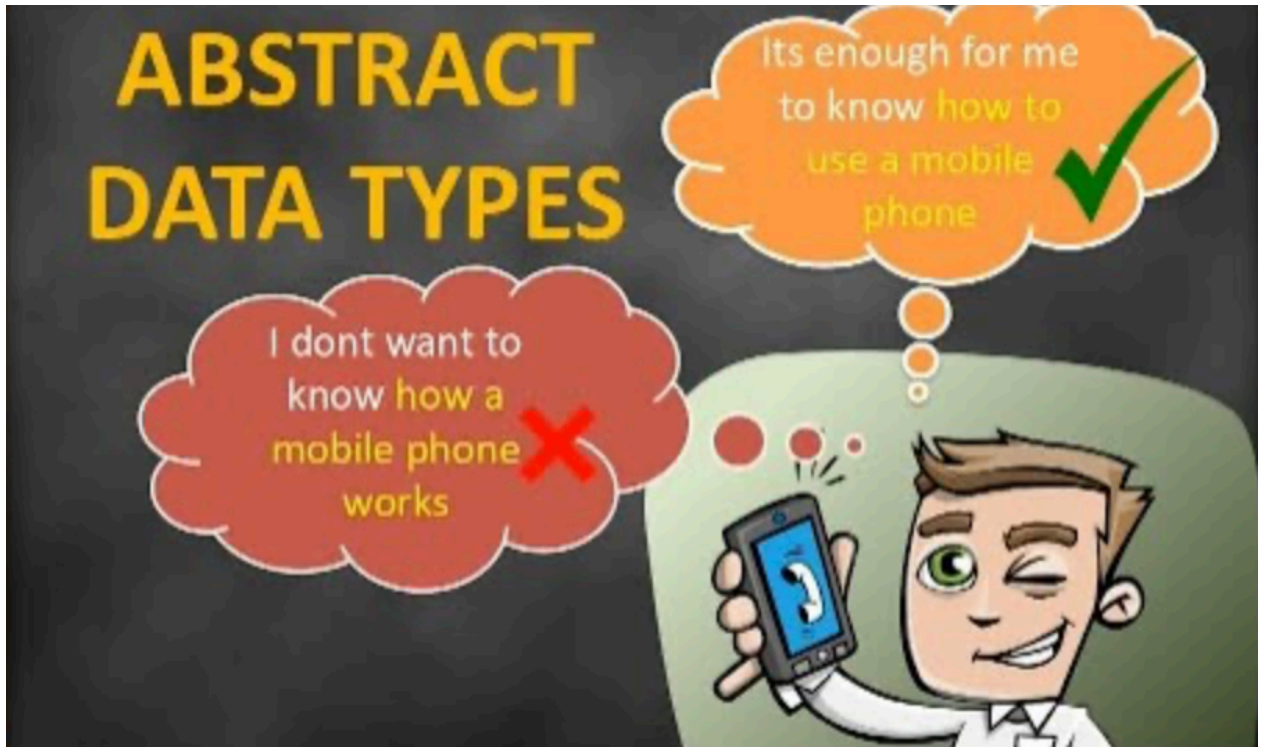
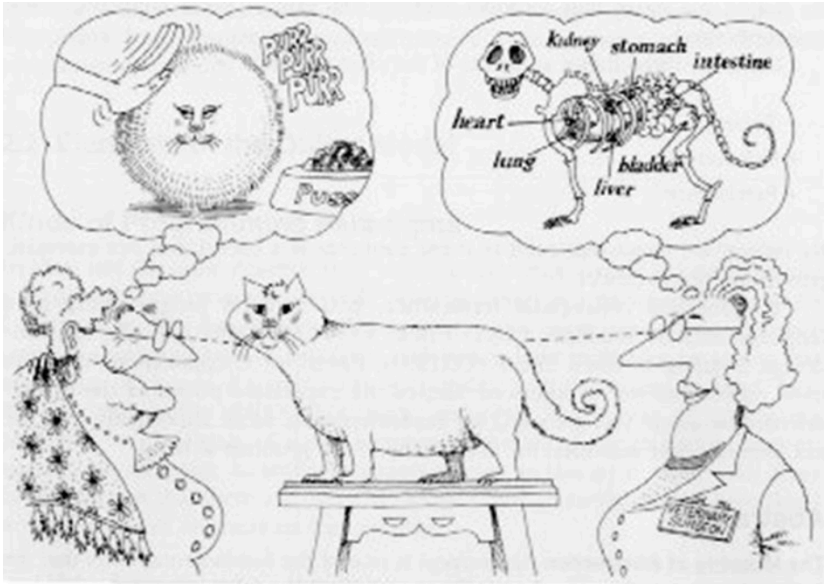
Department of Information Engineering

Tipi di dati astratti e strutture dati (capitolo 14, e ancora capitolo 9)



Tipi di dati astratti



- **Definizione -** una **struttura dati** (**data structure**) è un modo sistematico di organizzare i dati in un contenitore e di controllarne le modalità d'accesso
  - in Java si definisce una struttura dati con una **classe**
- **Definizione -** 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, insieme al tipo di informazioni necessarie per eseguire tali operazioni

Tipi di dati astratti e interfacce

- 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

Tipi di dati astratti

- Un ADT mette in generale a disposizione metodi per svolgere le seguenti azioni (a volte solo alcune)
  - **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
- Il package **java.util** della libreria standard contiene molte definizioni/realizzazioni di **ADT** come **interfacce** e **classi**
- Noi svilupperemo le nostre definizioni/realizzazioni, introducendo i più comuni ADT a partire dai più semplici

Un contenitore generico

```
public interface Container
{  boolean isEmpty();
   void makeEmpty();  }
```

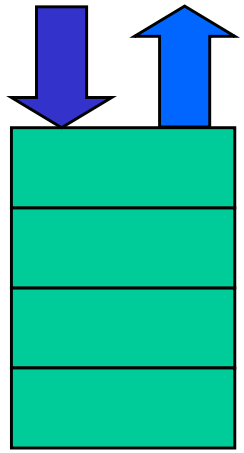
- **Container** specifica la firma di due soli metodi
  - **isEmpty** verifica che il contenitore sia vuoto
  - **makeEmpty** svuota il contenitore
- Scriveremo i nostri ADT come **interfacce di Java** che estendono **Container**

```
public interface TipoDatoAstratto extends Container
{ ... // firme metodi di inserimento/rimozione/ispezione
}     // (che sono diversi a seconda del tipo di dato astratto)
```

  - Le interfacce possono essere **estese** da altre interfacce
  - Un'interfaccia **eredita** i metodi della **super-interfaccia**
  - Una classe che realizza un'interfaccia estesa deve realizzare anche i metodi della sua super-interfaccia

Pila (stack)

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 che realizza questo comportamento

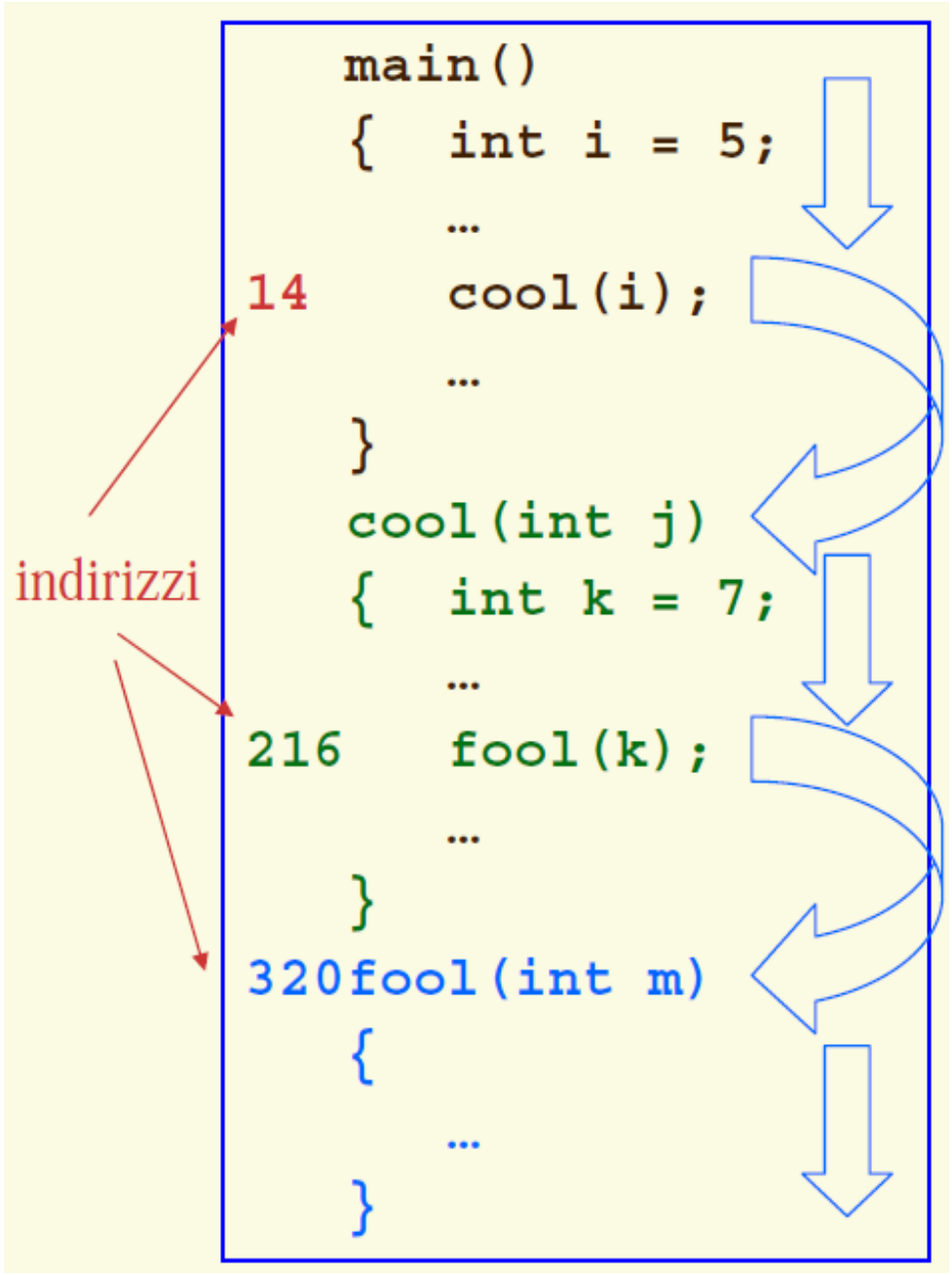
Utilizzo di pile

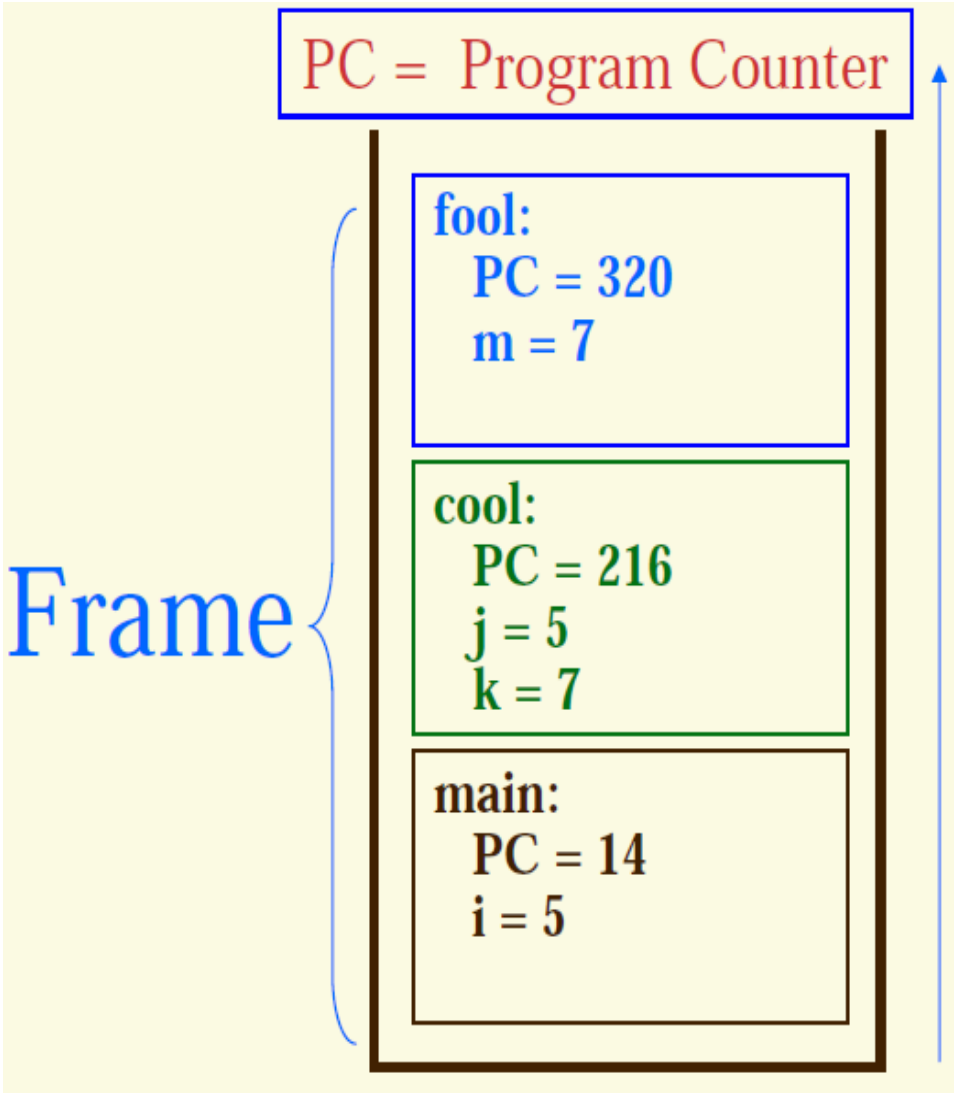
- I **browser** per internet memorizzano gli indirizzi dei siti visitati recentemente in una struttura di tipo pila. Quando l'utente visita un sito, l'indirizzo è inserito (**push**) nella pila. Il browser permette all'utente di saltare indietro (**pop**) al sito precedente tramite il pulsante "indietro"
- Gli **editor di testo** 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.
- La **JVM** usa una pila per memorizzare l'elenco dei metodi in attesa durante l'esecuzione in un dato istante

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
- La JVM mantiene quindi 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


main()
{ int i = 5;
…
14 cool(i);
…
}
cool(int j)
{ int k = 7;
…
216 fool(k);
…
}
320 fool(int m)
{
…
}
indirizzi


PC = Program Counter
Frame
fool:
PC = 320
m = 7
cool:
PC = 216
j = 5
k = 7
main:
PC = 14
i = 5

Pile nella Java Virtual Machine

- Il metodo ***fool*** è chiamato dal metodo ***cool*** che a sua volta è stato chiamato dal metodo ***main***.
- A ogni invocazione di metodo in run-time viene inserito (**push**) 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 (**pop**) ed eliminato
- Quando il metodo ***fool*** termina la propria esecuzione, il metodo ***cool*** continuerà la propria esecuzione a partire dall'istruzione di indirizzo 217, ottenuto incrementando il valore del PC contenuto nel proprio frame

Pila (stack)

```
public interface Stack extends Container
{   void push(Object obj);
    Object pop();
    Object top();
}
```

- Le operazioni (metodi) che caratterizzano una pila sono
  - **push:** inserisce un oggetto in cima alla pila
  - **pop:** elimina l'oggetto che si trova in cima alla pila
  - **top:** ispeziona l'oggetto in cima alla pila senza estrarlo
- Inoltre una pila (come ogni **Container**) ha i metodi
  - **isEmpty** per sapere se il contenitore è vuoto
  - **makeEmpty** per vuotare il contenitore
- **Molto importante:**
  - Definiremo tutti gli **ADT** in modo che possano genericamente contenere oggetti di tipo **Object**
  - Ciò consente di inserire nel contenitore oggetti di qualsiasi tipo (un riferimento di tipo **Object** può essere relativo a qualsiasi oggetto)

Utilizzo di pile

- Per evidenziare la potenza della definizione di tipi di dati astratti come interfacce, supponiamo che qualcuno abbia progettato le seguenti classi

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

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

- Senza sapere come siano realizzate **StackX** e **StackY**, possiamo usare esemplari di queste classi sfruttando il **comportamento astratto** definito in **Stack**

Utilizzo di pile: esempio

```
public class StackSwapper
{  public static void main(String[] args)
   {  Stack s = new StackX();
      s.push("Pippo");
      s.push("Pluto");
      s.push("Paperino");
      printAndClear(s);
      System.out.println();
      s.push("Pippo");
      s.push("Pluto");
      s.push("Paperino");
      printAndClear(swapAndClear(s));
   }
   private static Stack swapAndClear(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
```

Realizzazione della pila

Realizzazione della pila

- Per **realizzare una pila** è facile ed efficiente usare una struttura di tipo **array "riempito solo in parte"**
- In fase di realizzazione vanno affrontati **due problemi**
  - Cosa fare quando viene invocato il metodo (di inserimento) **push** nella situazione di **array pieno**
    - Una prima soluzione prevede il **lancio di un'eccezione**
    - Una seconda soluzione usa il **ridimensionamento dell'array**
  - Cosa fare quando vengono invocati i metodi **pop** (di rimozione) o **top** (di ispezione) quando la pila **è vuota**
  - Una possibile soluzione prevede il lancio di un'eccezione

Eccezioni nella pila

- Definiamo due **nuove eccezioni** (che sono classi) **EmptyStackException** e **FullStackException**,
  - Entrambe estendono **RuntimeException**, quindi chi usa la pila **non è obbligato a gestirle**

```
public interface Stack extends Container
{ ... } //codice come prima
class EmptyStackException extends RuntimeException {   }
class FullStackException extends RuntimeException  {   }
```

- **A cosa serve definire classi vuote??**
  - A definire un tipo di dato che ha le **stesse caratteristiche** della propria superclasse, ma un **nome diverso**
  - In realtà la classe non è vuota, perché contiene tutto ciò che eredita dalla sua superclasse
- Con le eccezioni si usa spesso questa tecnica
  - Il nome della classe eccezione specifica il tipo di errore

Pila senza ridimensionamento

```
public class FixedArrayStack implements Stack
{  //costruttore
   public FixedArrayStack()
   {  v = new Object[INITSIZE];
      // per rendere vuota la struttura invochiamo
      // il metodo makeEmpty: è sempre meglio evitare
      // di scrivere codice ripetuto
      makeEmpty();
   }

   // dato che Stack estende Container,
   // occorre realizzare anche i suoi metodi
   public void makeEmpty()
   {  vSize = 0; }

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

Pila senza ridimensionamento

```
  public void push(Object obj)                    // continua
   {  if (vSize == v.length)
         throw new FullStackException();
      v[vSize++] = obj;
   }
   public Object top()
   {  if (isEmpty())
         throw new EmptyStackException();
      return v[vSize - 1];
   }
   public Object pop()
   {  Object obj = top();//top fa controllo di pila vuota
      vSize--;
      return obj;
   }
  //campi di esemplare e variabili statiche
  protected Object[] v; //array riempito solo in parte
  protected int vSize;//ci è comodo usare var. protected
  public static final int INITSIZE = 100;
}
```

Pila con ridimensionamento

- Ora miglioriamo il nostro progetto
  - 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;
   }
   ... // tutto il resto è identico
}      // al codice di FixedArrayStack!
```

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

Pila con ridimensionamento

- Per evitare di riscrivere codice usiamo **l'ereditarietà**

```
public class GrowingArrayStack extends FixedArrayStack
{  public void push(Object obj)
   {  if (vSize == v.length)
         v = resize(2*vSize);
      v[vSize++] = obj;
   }
   protected Object[] resize(int newLength) //solita tecnica
    {   if (newLength < v.length)
            throw new IllegalArgumentException();
        Object[] newv = new Object[newLength];
        System.arraycopy(v, 0, newv, 0, v.length);
        return newv;
    }
}
```

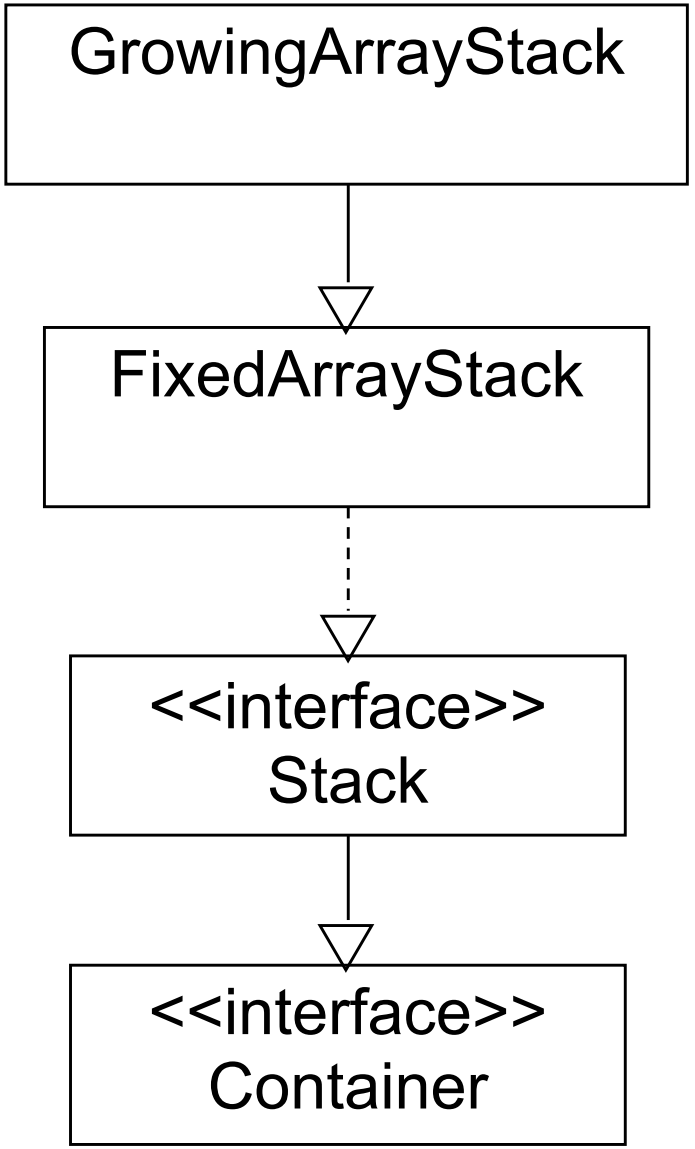
- Il metodo **push** sovrascritto fa accesso alle variabili di esemplare **v** e **vSize** definite nella superclasse
  - Questo è consentito dalla definizione **protected**

Campi di esemplare protected

- Il progettista della superclasse decide se rendere accessibile in modo **protected** lo stato della classe (o una sua parte…)
  - È una parziale **violazione dell'incapsulamento**, ma 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 e delle sue sottoclassi
- **Non** bisogna usare variabili di esemplare con accesso **protected**, se non in casi particolari
  - In questo caso la scelta di usare variabili di esemplare **protected** è motivata da finalità didattiche, e serve per rendere **più semplice** (ma **meno sicuro**) il codice.

Gerarchia di classi e interfacce

- Abbiamo realizzato la seguente **gerarchia** di classi e interfacce
  - L'interfaccia **Stack** estende l'interfaccia **Container**
  - La classe **FixedArrayStack** implementa l'interfaccia **Stack**
  - La classe **GrowingArrayStack** estende la classe **FixedArrayStack**



Prestazioni dei metodi di una pila e “analisi ammortizzata”

Prestazioni dei metodi di una pila

- Si noti che
  - **le prestazioni dipendono dalla definizione della struttura dati** e non dalla sua interfaccia…
  - per valutare le prestazioni è necessario conoscere il codice che realizza le operazioni!
- Il tempo di esecuzione di ogni operazione su una pila senza ridimensionamento (classe **FixedArrayStack**) è **costante**
  - Cioè non dipende dalla dimensione `n` della struttura dati stessa (non ci sono cicli…)
- Quindi ogni operazione su **FixedArrayStack** è `O(1)`

Prestazioni dei metodi di una pila

- Analizziamo i tempi di esecuzione nel caso di pila con ridimensionamento (**GrowingArrayStack**)
- L'unica differenza con **FixedArrayStack** è l'operazione **push**
  - "**Alcune volte**" **push** richiede un tempo `O(n)`
  - Perché all'interno del metodo **resize** è necessario copiare **tutti** gli **n** elementi nel nuovo array
  - **Osservazione**: poiché un array di lunghezza **n** viene ridimensionato in un array di lunghezza doppia, un eventuale nuovo ridimensionamento verrà effettuato dopo altre `n` operazioni di **push**

Analisi ammortizzata

- Cerchiamo di calcolare il **tempo di esecuzione medio** di `n` operazioni di push, 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 un ridimensiomento su **n** operazioni di **push**, si ottiene quindi ancora `O(1)`
- Questo metodo di stima del costo medio di una operazione si chiama **analisi ammortizzata** delle prestazioni asintotiche

Analisi ammortizzata

- **push** con ridimensionamento ha prestazioni `O(1)` per qualsiasi costante **moltiplicativa** usata per calcolare la nuova dimensione, anche diversa da 2
- Se invece si usa una **costante additiva *k***, cioè si ridimensiona l'array da **n** a **n+k**, allora su **n** operazioni di inserimento quelle "lente" sono **n/k**
  - **Esempio**: aumentando la dimensione dell'array di 20 elementi (ovvero **k = 20**), 5 inserimenti su 100 (o 50 su 1000) richiedono di invocare **resize**
- Con una costante additiva le prestazioni di **push** sono

$$T(n) = [(n-n/k)*O(1)+(n/k)*O(n)]/n$$
$$= [O(n) + n*O(n)]/n$$
$$= O(n)/n + O(n) = O(1) + O(n) = O(n)$$

Analisi ammortizzata

- Tecnica di **analisi delle prestazioni** di un algoritmo
- Si usa per mostrare che, in una sequenza di operazioni, il **costo medio** di una operazione è piccolo, anche se una singola operazione della sequenza è “costosa”
- È diversa dall'**analisi di caso medio** vista in precedenza
  - Analisi di caso medio: si basa su stime statistiche dell'input
  - Analisi ammortizzata: fornisce il **costo medio** di una singola operazione **nel caso peggiore**
- Ci sono varie tecniche di analisi ammortizzata
  - Analisi aggregata: si stima il tempo **T(n)** per una sequenza di **n** operazioni, nel caso peggiore. Allora il tempo medio per una operazione è **T(n)/n**
  - Noi applichiamo questa tecnica all'analisi dei tempi di esecuzione dei **metodi di inserimento** in strutture dati

Pile di oggetti, pile di numeri, classi involucro (cfr. Sezione 14.5)

Pile di oggetti e pile di numeri

- L'interfaccia **Stack** che abbiamo definito può gestire dati (cioè riferimenti a oggetti) di qualsiasi tipo
  - Non è però in grado di gestire dati dei tipi fondamentali di Java (**int**, **double**, **char**…), che non sono oggetti e non possono essere assegnati a riferimenti di tipo **Object**
- Per gestire una pila di int potremmo ridefinire tutto
- **Svantaggi**
  - occorre replicare codice, con poche modifiche
  - non esiste più un unico tipo di dati astratto **Stack**

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

Classi involucro

- Alternativa: "**trasformare**" un numero intero (o un altro tipo di dato fondamentale di Java) in un oggetto.
  - Questo è possibile, usando le **classi involucro** (**wrapper**)
- **Esempio**: per dati **int** esiste la classe involucro **Integer**
  - il costruttore accetta un parametro di tipo **int** e crea un oggetto **Integer** contenente il valore **int** passato come parametro, "**avvolto**" nella struttura di un oggetto
  - gli oggetti di tipo **Integer** sono immutabili
  - per conoscere il "valore" di un oggetto di tipo Integer si usa il metodo **intValue**, che restituisce valori di tipo int

```
Integer iObj = new Integer(2); //avvolge l'int 2 in un oggetto
Object obj = iObj //lecito, ovviamente
int x = iObj.intValue(); //x vale 2
```

Classi involucro

- Esistono classi involucro per **tutti i tipi di dati fondamentali**, con i nomi uguali al nome del tipo corrispondente ma iniziale maiuscola (eccezioni alla regola: **Integer** e **Character**)
  - Boolean, Byte, Character, Short, Integer, Long, Float, Double
  - Metodi: booleanValue( ), charValue( ), doubleValue( ), ecc.

```
//esempi di uso di classi involucro
double d1 = 3.5;  // tipo fondamentale
Double dObj1 = new Double(3.5);  //oggetto che incapsula double
Double dObj2 = 3.5; //auto-boxing: lecito a partire da java 5.0
double d2 = dObj2.doubleValue(); //restituisce valore double
double d3 = dObj1;  //auto-unboxing: lecito ... da java 5.0
```

Department of Information Engineering

Estrarre oggetti da strutture dati

Estrarre oggetti da strutture dati

- Le strutture dati generiche, definite in termini di **Object** sono molto comode perché possono contenere oggetti di qualsiasi tipo
- Quando si effettuano estrazioni o ispezioni di oggetti in esse contenuti viene restituito un riferimento di tipo **Object**, qualunque sia il tipo effettivo dell'oggetto
  - **Bisogna usare un cast** per ottenere un riferimento del tipo originario
  - Operazione potenzialmente pericolosa perché se il cast non è permesso viene lanciata **ClassCastException**

```
Stack st = new GrowingArrayStack();
...
Object obj = st.pop();
Character ch = (Character)obj; //possibile lancio di eccezione
```

Estrarre oggetti da strutture dati

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

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

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

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

Esercizio su utilizzo di pile: Calcolatrice (cfr. note di cronaca 14.6.2)

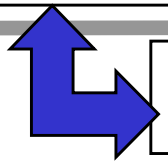
Esercizio: calcolatrice

- Vogliamo risolvere il seguente problema
  - calcolare il risultato di una espressione aritmetica (ricevuta come **String**) contenente somme, sottrazioni, moltiplicazioni e divisioni
- Se l'espressione usa la classica **notazione *infissa*** (in cui i due operandi di un'operazione si trovano ai due lati dell'operatore) l'ordine di esecuzione delle operazioni è determinato dalle **regole di precedenza** tra gli operatori e da eventuali parentesi
- Scrivere un programma per tale compito è piuttosto **complesso**, mentre è molto più facile calcolare espressioni che usano una **diversa notazione**

Notazione postfissa

- Usiamo una notazione ***postfissa*** detta ***notazione polacca inversa*** (**RPN**, ***Reverse Polish Notation***)
  - La notazione polacca (***prefissa***) fu introdotta dal matematico polacco Jan Lukasiewicz nel 1920
  - **Non sono ammesse parentesi** (e non sono necessarie)
  - Nella stringa che rappresenta l'espressione aritmetica, gli operandi sono scritti alla **sinistra** dell'operatore
  - Ogni volta che nella stringa si incontra un operatore, si esegue la corrispondente operazione sui due numeri che lo precedono
    - Possono essere numeri scritti nella stringa di partenza
    - Oppure possono essere numeri risultanti da un'operazione eseguita precedentemente

```
7 1 2 + 4 * 5 6 + - /
```



```
7/[(1+2)*4 - (5+6)]
```

Notazione postfissa

- Esiste un semplice algoritmo che usa una **pila** per valutare un'espressione in notazione postfissa
- **Finché** l'espressione non è terminata
  - **leggi** da sinistra il primo simbolo dell'espressione non letto
  - **se** è un valore numerico, inseriscilo sulla pila
  - **altrimenti** (è un operatore…)
    - **estrai** dalla pila l'operando destro
    - **estrai** dalla pila l'operando sinistro
    - **esegui** l'operazione
    - **inserisci** il risultato sulla pila
- **Se** (al termine) la pila contiene più di un valore, l'espressione contiene un **errore**
- L'unico valore presente sulla pila è il **risultato**

Una classe RPNTester

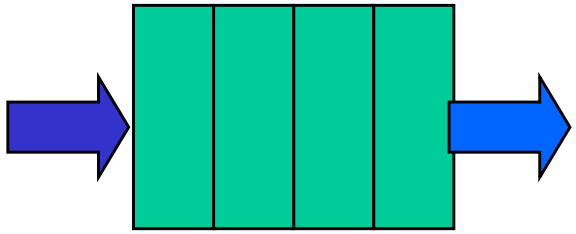
```
import java.util.Scanner;

public class RPNTester
{   public static void main(String[] args)
    {
        System.out.println("Inserisci operazione. Una stringa su una riga,");
        System.out.println("solo numeri e operatori +-*/ separati da spazi");

        Scanner in = new Scanner(System.in);
        String rpnString = in.nextLine();
        try
        {   System.out.println("Risultato: " + evaluateRPN(rpnString));
        }
        catch(NumberFormatException e)
        { System.out.println("Uso di simboli non permessi!");
        }
        catch(EmptyStackException e)
        { System.out.println("Troppi operatori nell'espressione");
        }
        catch(IllegalStateException e)
        { System.out.println("Troppi numeri nell'espressione");
        }
    }

                                                          //continua
```

Il metodo evaluateRPN

```
    private static double evaluateRPN(String rpnString)            //continua
    throws EmptyStackException, NumberFormatException, IllegalStateException
    {   Stack st = new GrowingArrayStack();
        Scanner linescan = new Scanner(rpnString);
        while (linescan.hasNext())
        {   String s = linescan.next();
            if (isOperator(s))  // il token s e` un operatore
            {   Double r = evalOp(s, (Double)st.pop(),(Double)st.pop());
                st.push(r);
            }
            else   // il token s non e` un operatore
                st.push(Double.parseDouble(s)); //dovrebbe essere un numero
        }
        double result = (Double)st.pop();
        if (!st.isEmpty()) throw new IllegalStateException();
        return result;
    }
    // verifica se il il token s e` un operatore
    private static boolean isOperator(String s)
    {return s.equals("+")|| s.equals("-")|| s.equals("*")|| s.equals("/");}
    // calcola il risultato dell'operazione "left op right"
    private static double evalOp(String op, double right, double left)
    {   if (op.equals("+")) return left + right;
        if (op.equals("-")) return left - right;
        if (op.equals("*")) return left * right;
        return left / right;
    }
}
```

Coda (*queue*)

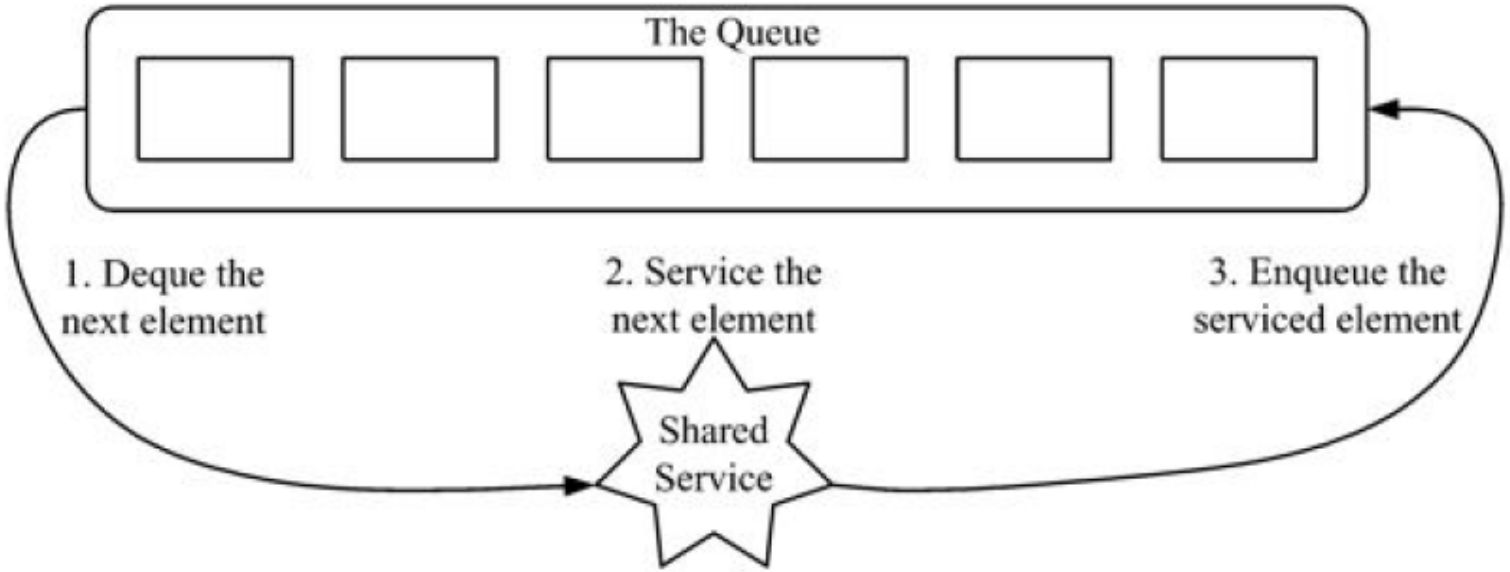
Coda (queue)



- 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 suggerisce l'analogia con una coda di persone
- L'unico oggetto che può essere ispezionato è il primo oggetto della coda
- Esistono molti possibili utilizzi di una coda
  - Simulazione del funzionamento di uno sportello bancario
    - Clienti inseriti in coda per rispettare priorità di servizio
  - File da stampare vengono inseriti in una **coda di stampa**
    - La stampante estrae dalla coda e stampa un file alla volta

Coda circolare

- Spesso si utilizza una coda secondo una modalità **circolare**: gli elementi vengono **estratti** dalla prima posizione, "**serviti**", e **reinseriti** in ultima posizione
- Esempio: lo **scheduler** di un sistema operativo assegna le risorse della **CPU** a molti processi attivi in parallelo
  - Politica **round robin**: la CPU esegue una porzione del processo che ha atteso più a lungo di essere "servito", che poi viene reinserito in ultima posizione



Coda (queue)

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

- Le operazioni (metodi) che caratterizzano una coda sono
  - **enqueue:** inserisce un oggetto nella coda
  - **dequeue:** elimina dalla coda l'oggetto inserito per primo
  - **getFront:** esamina il primo oggetto, senza estrarlo
- Si notino le similitudini con i metodi di una pila
  - **enqueue** corrisponde a **push**
  - **dequeue** corrisponde a **pop**
  - **getFront** corrisponde a **top**
- Come ogni ADT di tipo "contenitore", la coda ha i metodi
  - **isEmpty** per sapere se il contenitore è vuoto
  - **makeEmpty** per vuotare il contenitore

Coda (queue)

- Per **realizzare una pila** è facile ed efficiente usare una struttura di tipo **array "riempito solo in parte"**
- In fase di realizzazione vanno affrontati **due problemi**
  - Come gestire le condizioni di coda piena e coda vuota
    - Definiamo **EmptyQueueException** e **FullQueueException**

```
public interface Queue extends Container
{ ... } //codice come prima
class EmptyQueueException extends RuntimeException { }
class FullQueueException extends RuntimeException { }
```

  - Come realizzare inserimento ed estrazione di elementi ai **due diversi estremi** dell'array
    - Nella pila si inseriscono e si estraggono elementi allo stesso estremo dell'array (l'estremo "destro")
    - Nella coda decidiamo di **inserire a destra** ed **estrarre a sinistra**

La classe SlowFixedArrayQueue

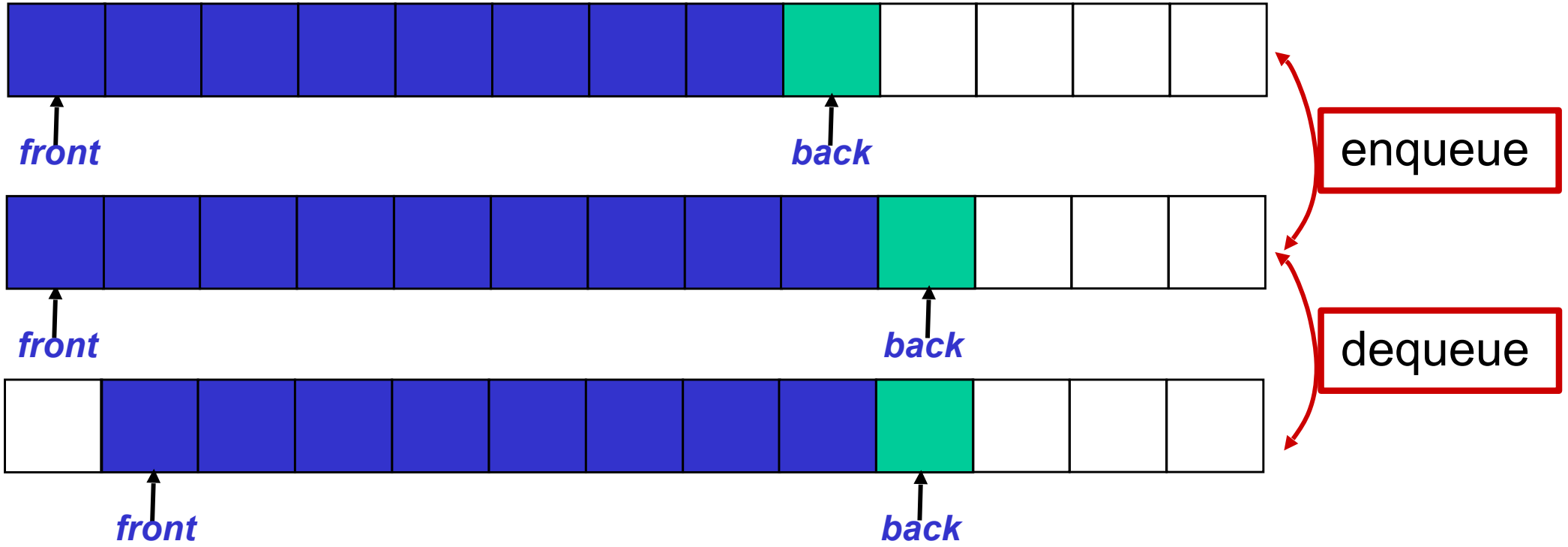
```
public class SlowFixedArrayQueue implements Queue
{  public SlowFixedArrayQueue()
   {  v = new Object[INITSIZE];
      makeEmpty();
   }
   public void makeEmpty()
   {  vSize = 0;
   }
   public boolean isEmpty()
   {  return (vSize == 0);
   }
   public void enqueue(Object obj)
   {  if (vSize == v.length) throw new FullQueueException();
      v[vSize++] = obj;
   }
   public Object getFront()
   {  if (isEmpty()) throw new EmptyQueueException();
      return v[0];
   }
   public Object dequeue()
   {  Object obj = getFront();
      vSize--;
      for (int i = 0; i < vSize; i++) v[i] = v[i+1];
      return obj;
   }
   //campi di esemplare e variabili statiche
   private Object[] v;
   private int vSize;
   public static final int INITSIZE = 100;
}
```

Migliorare il metodo dequeue

- Questa realizzazione, 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 "ricompattare" l'array
  - Ciò avviene perché inserimenti e rimozioni nella coda avvengono alle **due diverse estremità** dell'array, mentre nella pila avvengono alla stessa estremità
- Per migliorare l'efficienza servono **due indici**
  - un indice punta al **primo oggetto** della coda e l'altro indice punta alla **prima posizione libera** nella coda
  - Aggiornando opportunamente gli indici si realizza una coda con un "**array riempito solo nella parte centrale**", in cui **tutte le operazioni sono** `O(1)`

Array riempito nella parte centrale

- Si usano **due indici** anziché uno soltanto
  - Indice **front**: punta al primo elemento nella coda
  - Indice **back**: punta al primo posto libero dopo l'ultimo elemento nella coda
  - Il numero di elementi è **(back – front)**, in particolare quando **front == back** l'array è vuoto.



La classe FixedArrayQueue

```
public class FixedArrayQueue implements Queue
{  public FixedArrayQueue()
   {  v = new Object[INITSIZE];
      makeEmpty();
   }
   public void makeEmpty()
   {  front = back = 0;
   }
   public boolean isEmpty()
   {  return (back == front);
   }
   public void enqueue(Object obj)
   {  if (back == v.length) throw new FullQueueException();
      v[back++] = obj;
   }
   public Object getFront()
   {  if (isEmpty()) throw new EmptyQueueException();
      return v[front];
   }
   public Object dequeue()
   {  Object obj = getFront();
      front++; //attenzione all'effetto di questo incremento
      return obj;
   }
   //campi di esemplare e variabili statiche
   protected Object[] v;
   protected int front, back;
   public static final int INITSIZE = 100;
}
```

Coda ridimensionabile

- Per rendere la coda **ridimensionabile**, usiamo la stessa strategia vista per la pila
  - Estendiamo la classe **FixedArrayQueue** e sovrascriviamo il solo metodo **enqueue**
- Tutte le operazioni continuano ad avere la massima efficienza: sono `O(1)`

```
public class GrowingArrayQueue extends FixedArrayQueue
{  public void enqueue(Object obj)
   {  if (back == v.length)
         v = resize(2*v.length);
      v[back++] = obj;
   }
   protected Object[] resize(int newLength)
   { ... }  // stesso codice gia` scritto in GrowingArrayStack
}
```

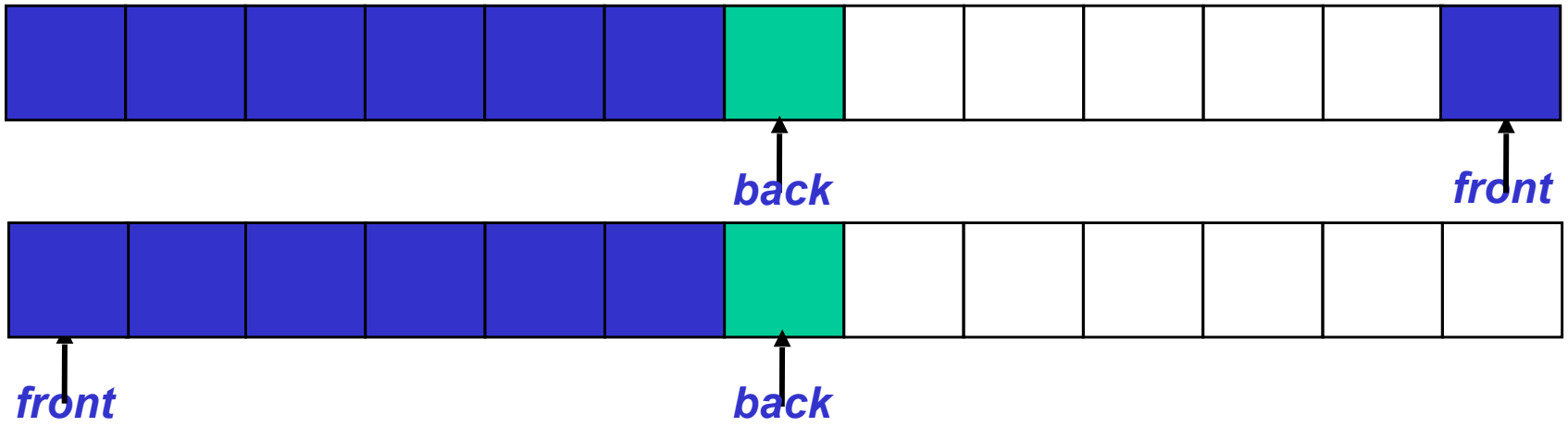
Coda con array circolare

- La realizzazione vista ha ancora un **punto debole**
  - Se l'array è lungo **n,** effettuare **n** operazioni **enqueue** e **n** operazioni **dequeue** (anche non consecutive) produce **front=back=n,** ovvero la coda è **vuota**
  - Ma al successivo **enqueue** **l'array sarà pieno** (perché **back=n**): lo spazio di memoria **non viene riutilizzato**
- Questo problema può essere risolto usando una struttura detta "**array circolare**"
  - I due indici possono, una volta giunti alla fine dell'array, **ritornare all'inizio** se si sono liberate delle posizioni
  - L'array circolare è pieno quando la coda contiene un numero di oggetti **uguale** a **n–1** (e non **n**).
    - Si "spreca" quindi un elemento dell'array: ciò è necessario per distinguere la condizione di coda vuota (**front==back**) dalla condizione di coda piena
  - le prestazioni temporali rimangono identiche

Array circolare

- Incremento dell'indice ***back*** da **n-1** a **0**

- Incremento dell'indice ***front*** da **n-1** a **0**

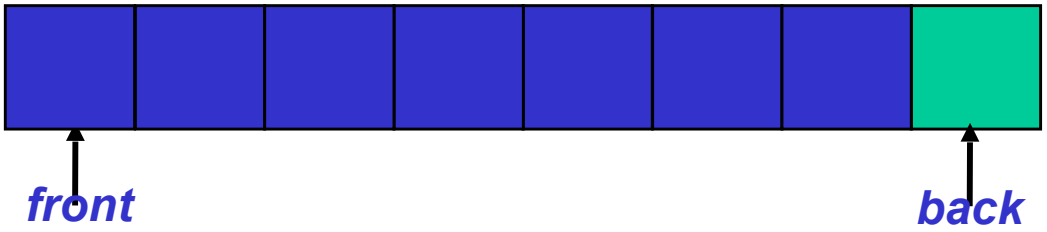


La classe FixedCircularArrayQueue

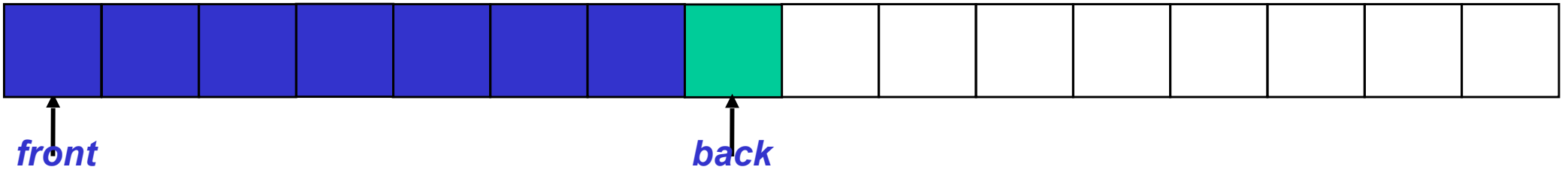
```
public class FixedCircularArrayQueue extends FixedArrayQueue
{
 // il metodo increment fa avanzare un indice di una
 // posizione, tornando all'inizio se si supera la fine.
 // Attenzione: non aggiorna direttamente i campi front,back
  protected int increment(int index)
  {  return (index + 1) % v.length;
  }
  public void enqueue(Object obj) //SOVRASCRITTO!
  {  if (increment(back) == front)//condizione di coda piena
        throw new FullQueueException();
     v[back] = obj;
     back = increment(back);
  }
  public Object dequeue() //SOVRASCRITTO!
  {  Object obj = getFront();
     front = increment(front);
     return obj;
  }
  // non serve sovrascrivere getFront perche` non modifica
  // le variabili back e front
}
```

Ridimensionare un array circolare

- Vogliamo estendere **FixedCircularArrayQueue** in maniera tale che l'array contenente i dati possa essere ridimensionato quando la coda è piena
  - Effettuiamo un **resize** come su un array ordinario
- Se **front = 0** e **back = n-1** la condizione di array pieno equivale alla condizione **increment(back) == front**

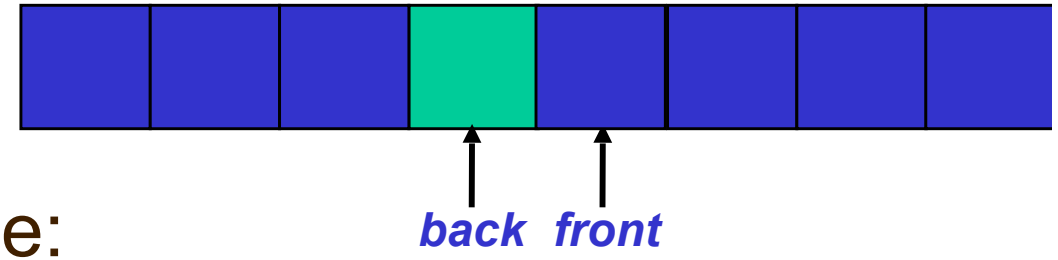


- L'operazione di **resize** ha l'effetto desiderato:

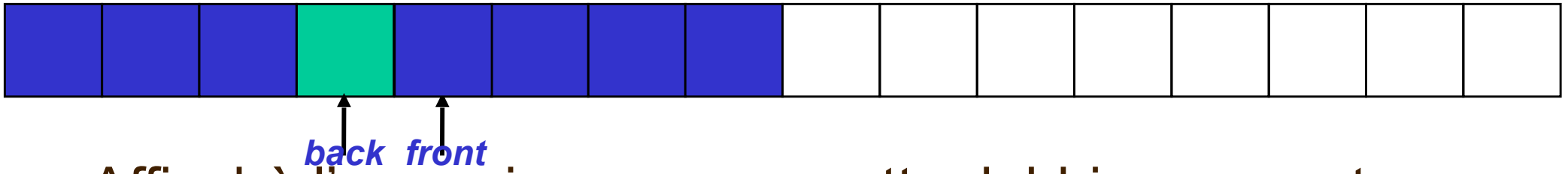


Ridimensionare un array circolare

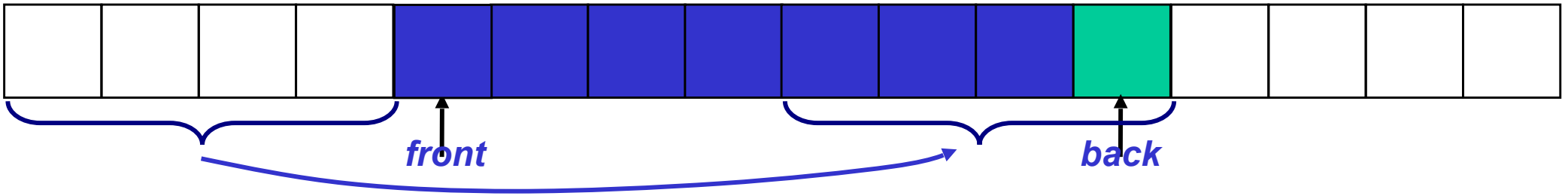
- In generale però la zona utile della coda è **attorno** alla sua fine (ovvero **back < front**): c'è un problema in più
  - La condizione di array pieno equivale sempre a **increment(back) == front**



- Raddoppiamo la dimensione:



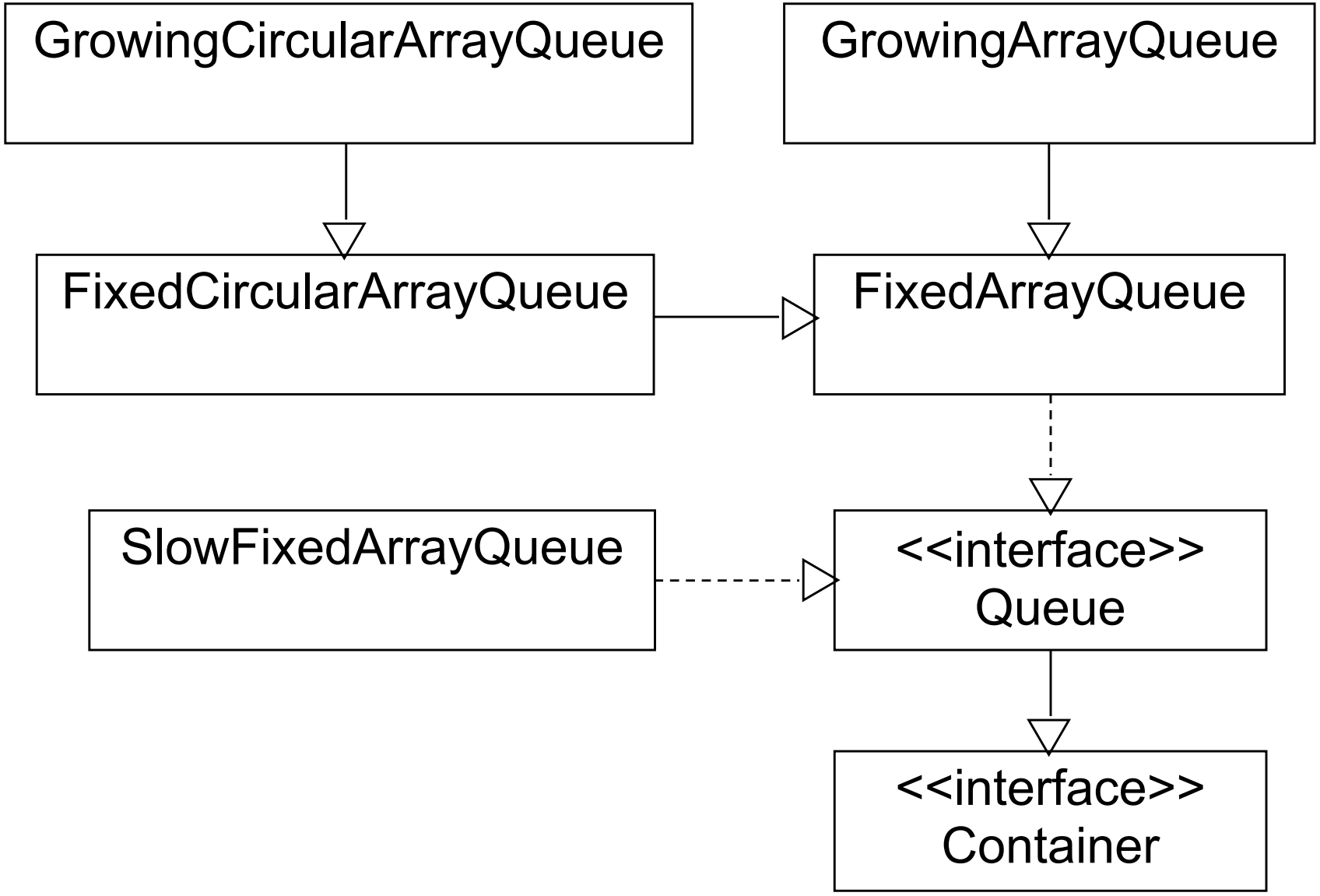
- Affinchè l'array rimanga compatto dobbiamo spostare nella seconda metà dell'array la prima parte della coda:



La classe GrowingCircularArrayQueue

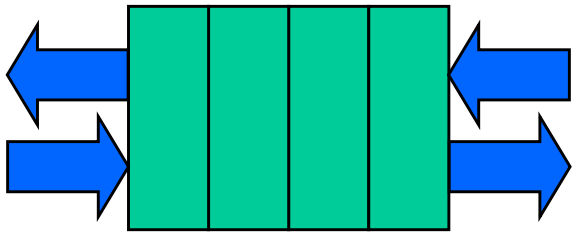
```
public class GrowingCircularArrayQueue
      extends FixedCircularArrayQueue
{ public void enqueue(Object obj)
  { if (increment(back) == front) //back e front distano 1
    {  v = resize(2*v.length);
       //se si ridimensiona v e la zona utile della coda e`
       //attorno alla sua fine (cioe` back < front) la seconda
       //meta` del nuovo array rimane vuota e provoca un
       //malfunzionamento della coda, che si risolve spostando
       //la parte di coda che si trova all'inizio dell'array
       if (back < front)
       {  System.arraycopy(v, 0, v, v.length/2, back);
          back += v.length/2;
       }
    }
    v[back] = obj;
    back = increment(back);
  }

  protected Object[] resize(int newLength)
  { ... } // solito codice
}
```

Gerarchia di classi e interfacce



L'ADT “Coda doppia” (cenni)

Coda doppia



- In una **coda doppia** (**deque**) gli oggetti possono essere inseriti ed estratti **ai due estremi** di una disposizione lineare, cioè all'**inizio** e alla **fine**.
  - Inoltre è consentita l'ispezione dei due oggetti presenti alle due estremità
- Si parla di **double-ended queue**, ovvero di “coda con due estremità terminali”
  - Tradizionalmente la definizione viene abbreviata con la parola **deque** (dove le prime due lettere sono le iniziali di “double-ended”), pronunciata come **deck** (per evitare confusione con il metodo dequeue).

Coda doppia (deque)

```
public interface Deque extends Container
{  void addFirst(Object obj);  // inserimento ai due capi
   void addLast(Object obj);

   Object removeFirst();       // rimozione ai due capi
   Object removeLast();

   Object getFirst();          // ispezione ai due capi
   Object getLast();

   int size();                 // dimensione della deque
}

// solite eccezioni per contenitore pieno/vuoto
class EmptyDequeException extends RuntimeException { }
class FullDequeException extends RuntimeException { }
```

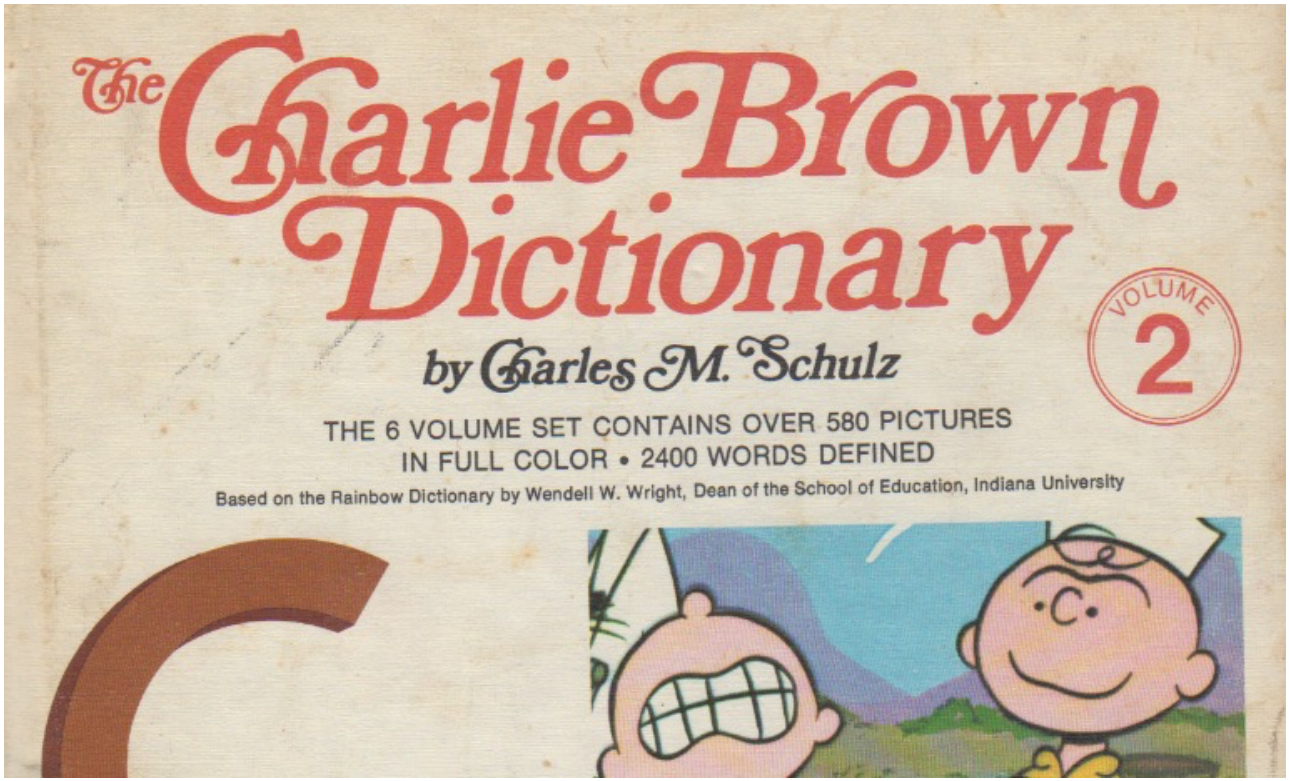
- Può essere realizzata con array, in particolare **array circolare ridimensionabile**
  - Tutti i metodi hanno prestazioni **O(1)** (in senso ammortizzato per i metodi di inserimento con ridimensionamento dell'array)

Pile e code tramite deque

- Una coda doppia può essere utilizzata per realizzare pile e code

Metodi Pila	***Metodi coda doppia***
push(Object obj)	addFirst(Object obj)
pop()	removeFirst()
top()	getFirst()
Metodi Coda	***Metodi coda doppia***
enqueue(Object obj)	addLast(Object obj)
dequeue()	removeFirst()
getFront()	getFirst()

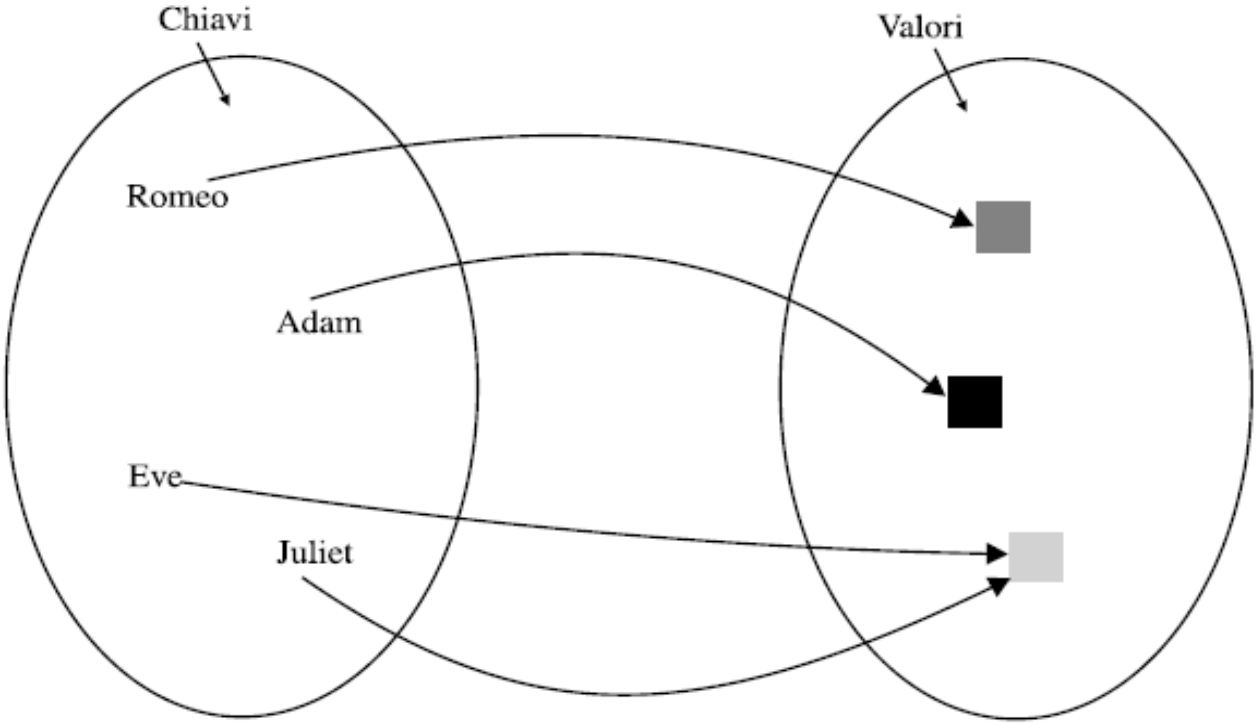
Mappe e Dizionari (capitolo 14 - ma la trattazione è abbastanza diversa)



Department of Information Engineering

Mappa: definizione

- Una **mappa** è un **ADT** con le seguenti proprietà
  - Contiene dati (non in sequenza) che sono coppie di tipo **chiave/valore**
  - Non può contenere coppie con identica chiave: ogni chiave deve essere **unica** nell'insieme dei dati memorizzati
  - Consente di inserire nuove coppie chiave/valore
  - Consente di effettuare ricerca e rimozione di valori **usando la chiave come identificatore**



Dizionario: definizione

- L'**ADT dizionario** ha molte similitudini con l'**ADT mappa**
  - Valgono tutte le proprietà dell'ADT mappa, tranne una
  - **Non** si richiede che le chiavi siano uniche nel dizionario
- C'è analogia con un dizionario di uso comune, in cui
  - le **chiavi** sono le singole **parole**
  - I **valori** sono le **definizioni** delle parole nel dizionario
  - le chiavi (parole) possono essere **associate a più valori** (definizioni) e quindi comparire più volte nel dizionario
  - la **ricerca** di un valore avviene tramite la sua chiave
- Si distinguono **dizionari ordinati** e **dizionari non-ordinati**
  - A seconda che sull'insieme delle chiavi sia o no definita una relazione totale di ordinamento, cioè (in Java) che le chiavi appartengano a una classe che implementa **Comparable**
- La nostra trattazione è limitata al caso **chiave unica** (cioè mappe), ordinati e non ordinati

L'interfaccia Dictionary

```
public interface Dictionary extends Container
{
   // l'inserimento va sempre a buon fine; se la chiave non
   // esiste la coppia viene aggiunta al dizionario. Se esiste,
   // il valore a essa associato viene sovrascritto dal nuovo
   // valore; se key è null si lancia IllegalArgumentException
   void insert(Comparable key, Object value);

   // la rimozione della chiave rimuove anche il corrispondente
   // valore dal dizionario. Se la chiave non esiste si lancia
   // DictionaryItemNotFoundException
   void remove(Comparable key);

   // la ricerca per chiave restituisce (solo) il valore a essa
   // associato nel dizionario. Se la chiave non esiste si
   // lancia DictionaryItemNotFoundException
   Object find(Comparable key);
}

//Eccezione che segnala il mancato ritrovamento di una chiave
class DictionaryItemNotFoundException extends RuntimeException
{  }
```

Implementare Dictionary con array

- Un **dizionario** può essere realizzato usando la struttura dati **array**
  - Ogni cella dell'array contiene un riferimento a una **coppia** chiave/valore
  - La coppia chiave/valore sarà un oggetto di tipo **Pair** (da definire)
  - Generalmente si usa un array riempito solo in parte
- A seconda degli ambiti applicativi ci sono due strategie possibili
  - mantenere le chiavi **ordinate** nell'array
  - mantenere le chiavi **non ordinate** nell'array
- A seconda della strategia scelta, **cambiano le prestazioni** dei metodi del dizionario

Dizionario con array ordinato

- Se le **n** chiavi vengono conservate **ordinate** nell'array
- La **ricerca** ha prestazioni **O(log n)**
  - Perché si può usare la **ricerca per bisezione**
- La **rimozione** ha prestazioni **O(n)**
  - Perché bisogna effettuare una ricerca, e poi spostare **mediamente n/2** elementi per mantenere l'ordinamento
- L'**inserimento** ha prestazioni **O(n)**
  - Perché si può usare **insertionsort in un array ordinato**
  - Usando un diverso algoritmo occorre riordinare l'intero array, con prestazioni almeno **O(n log n)**

Dizionario con array non ordinato

- Se le **n** chiavi vengono conservate **non ordinate**
- La **ricerca** ha prestazioni **O(n)**
  - Bisogna usare la **ricerca lineare**
- La **rimozione** ha prestazioni **O(n)**
  - Bisogna **effettuare una ricerca (lineare)**, e poi spostare nella posizione trovata l'ultimo elemento dell'array (l'ordinamento non interessa)
- L'**inserimento** ha prestazioni **O(n)**
  - Bisogna **rimuovere** (sovrascrivere) un elemento con la stessa chiave, se c'è, e poi inserire il nuovo elemento nella ultima posizione dell'array (l'ordinamento non interessa)
  - [Se non si richiede che le chiavi siano uniche nel dizionario, la rimozione non è necessaria e l'inserimento è **O(1)**]

Prestazioni di un dizionario

Dizionario	array ordinato	array non ordinato	
ricerca	O(lg n)	O(n)	
inserimento	O(n)	O(n)	[O(1) per chiavi non uniche]
rimozione	O(n)	O(n)	

- La scelta di una particolare realizzazione dipende dall'utilizzo tipico del dizionario **nell'applicazione**
  - Se si fanno **frequenti inserimenti** e **sporadiche ricerche e rimozioni** la scelta migliore è **l'array non ordinato**
  - Se il dizionario viene **costruito una volta per tutte**, poi viene usato **soltanto per fare ricerche** la scelta migliore è **l'array ordinato**

Realizzazione di un dizionario

La classe Pair

- Un dizionario contiene elementi formati da **coppie**
  - Chiave – Valore
- Per realizzare un dizionario tramite array, dobbiamo allora realizzare una classe **Pair**, che definisce i generici elementi di un dizionario
  - L'array contenente gli elementi del dizionario sarà un array di tipo **Pair[ ]**
- Oggetti di tipo **Pair** devono avere
  - Due **campi di esemplare**, **key** (di tipo **Comparable** perché trattiamo dizionari ordinati) e **value** (di qualsiasi tipo, ovvero di tipo **Object**)
  - Metodi di accesso e modificatori per questi campi di esemplare

La classe Pair

```
public class Pair
{   public Pair(Comparable key, Object value)
    {   setKey(key);
        setValue(value);
    }

    //metodi pubblici
    public String toString()
    {  return key + " " + value; }
    public Comparable getKey()
    {  return key; }
    public Object getValue()
    {  return value; }
    public void setKey(Comparable key)
    {  this.key = key; }
    public void setValue(Object value)
    {  this.value = value; }

    //campi di esemplare
    private Comparable key;
    private Object value;

}
```

Classi interne
(cfr. sezione 10.5)

Classi interne

- Osserviamo che la classe **Pair**, usata dal dizionario, **non viene mai usata** al di fuori del dizionario stesso
  - I metodi dell'interfaccia Dictionary non restituiscono mai riferimenti a **Pair**
  - E non ricevono mai parametri espliciti di tipo **Pair**
- Per il principio dell'**incapsulamento** sarebbe preferibile che questa classe e i suoi dettagli non fossero visibili all'esterno della catena
  - in questo modo una modifica della struttura interna del dizionario e/o della classe **Pair** non avrebbe ripercussioni sul codice scritto da chi usa il dizionario

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 esaminare la seguente sintassi (che è lecita in Java)

```
public class ClEsterna
{   ... //costruttori, metodi, campi di
        //esemplare, variabili statiche
        //della classe esterna

    <tipoaccesso> class ClInterna //<tipoaccesso> puo`
    {                             //anche essere private!
        ... //costruttori, metodi, campi di
            //esemplare, variabili statiche
            //della classe interna
    }
}
```

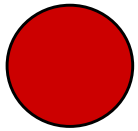
Compilare classi interne

- Quando compiliamo classi contenenti classi interne
  - Il compilatore traduce una classe interna in un **normale file di bytecode**
  - … **Diverso** dal file di bytecode della classe esterna
  - Il file di bytecode della classe interna ha un nome dal formato particolare
- Per esempio, se compiliamo **ClEsterna** con la classe interna **ClInterna**, troviamo nella nostra cartella
  - Il file di bytecode **ClEsterna.class** (come al solito...)
  - Il file di bytecode **ClEsterna$ClInterna.class**

Classi interne

- Solitamente si definisce una classe come interna se essa descrive un tipo logicamente correlato a quello della classe esterna
- **Vantaggio**: le due classi, interna ed esterna, condividono una **"relazione di fiducia"**
  - Ciascuna delle due classi ha accesso a **tutti** i metodi, campi di esemplare e statici dell'altra, **anche se private**
- **Limitazioni**:
  - un oggetto di **ClInterna** è sempre associato a un oggetto di **ClEsterna**. Ovvero si possono creare oggetti di tipo **ClInterna** **solo dentro metodi non statici** di **ClEsterna**
  - La classe interna può essere resa **inaccessibile** al codice scritto in altre classi

Uso di classi interne



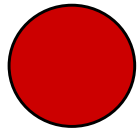
- Primo caso: dentro **ClEsterna**

```
public class ClEsterna
{   //metodi della classe esterna
    public ClInterna metEsterno()
    {   ClInterna obj = new ClInterna(); //LECITO
        obj.campointerno = 2;//LECITO: anche se il campo e`
        return obj;           //       privato in ClInterna
    }
    //campi di esemplare della classe esterna
    private double campoEsterno;

    //definizione di classe interna
    public class ClInterna  //puo` anche essere private!
    {   //metodi della classe interna
        public void metInterno()
        {   ClEsterna obj = new ClEsterna(); //LECITO
            double a = campoEsterno; //LECITO: anche se il campo
        }                            //  e` privato in ClEsterna
        //campi di esemplare della classe interna
        private int campointerno;
    }
}
```

Uso di classi interne

- Secondo caso: in una classe diversa da **ClEsterna**.
  - Il nome della classe interna **va sempre qualificato** rispetto al nome della classe esterna
    - Non si può usare la sintassi **ClInterna**
    - Bisogna usare la sintassi **ClEsterna.ClInterna**
  - Non è **mai** possibile **creare oggetti** di tipo **ClInterna**
  - Se **ClInterna** è public, allora è possibile definire **variabili oggetto** di tipo **ClInterna**
  - Se **ClInterna** è private, allora **è "inaccessibile"** da codice che non sia scritto in **ClEsterna**
    - In questo modo si protegge il codice da ogni possibile violazione dell'incapsulamento

Uso di classi interne



- Secondo caso: in una classe diversa da **ClEsterna**

```
public class  ClInterneTester
{   public static void main(String[] args)
    {  ClEsterna e = new ClEsterna();  //tutto ok

       //MAI LECITO: non si possono creare ogg. di ClInterna qui
       ClEsterna.ClInterna obj = new ClEsterna.ClInterna();

       //MAI LECITO: il tipo deve essere "ClEsterna.ClInterna"
       ClInterna i = e.metEsterno();

       //LECITO SOLO SE ClInterna e` public in ClEsterna
       ClEsterna.ClInterna i = e.metEsterno();

       //LECITO SOLO SE sia ClInterna che il metodo sono public
       (e.metEsterno()).metInterno();
       //LECITO SOLO SE sia ClInterna che il campo sono public
       (e.metEsterno()).campointerno = 1;

       //SEMPRE LECITO (ma inutile se ClInterna e` private)
       e.metEsterno();
    }
}
```

La classe interna Pair

```
public class ArrayDictionary implements Dictionary
{ ...
    protected class Pair //classe interna ad ArrayDictionary
    {   public Pair(Comparable key, Object value)
        {   setKey(key);
            setValue(value); }
        //metodi pubblici
        public String toString()
        {  return key + " " + value; }
        public Comparable getKey()
        {  return key; }
        public Object getValue()
        {  return value; }
        public void setKey(Comparable key)
        {  this.key = key; }
        public void setValue(Object value)
        {  this.value = value; }
        //campi di esemplare
        private Comparable key;
        private Object value;
    }
  ...
}
```

La classe ArrayDictionary

```
public class ArrayDictionary implements Dictionary
{   public ArrayDictionary()
    {   v = new Pair[INITSIZE];  // ... sempre uguale
        MakeEmpty();  }
    public boolean isEmpty()
    {   return vSize == 0;  }   // ... sempre uguale
    public void makeEmpty()
    {   vSize = 0;  }           // ... sempre uguale
    public String toString()
    {   String s = "";
        for (int i = 0; i < vSize; i++)
            s = s + v[i] + "\n";
        return s;
    }
    public void insert(Comparable key, Object value)
    {   if (key == null) throw new IllegalArgumentException();
        try
        { remove(key);} //elimina elemento se gia` presente
        catch (DictionaryItemNotFoundException e)
        {} //... ovvero sovrascrive elemento se gia` presente
        if (vSize == v.length)  v = resize(2*vSize);
        v[vSize++] = new Pair(key, value);
    }                                                    //
continua
```

La classe ArrayDictionary

```
                                                        //continua
protected Pair[] resize(int newLength) //metodo ausiliario
{ ... } //solito codice
public void remove(Comparable key)
{   v[linearSearch(key)] = v[--vSize];
}
public Object find(Comparable key)
{   return v[linearSearch(key)].getValue();
}
private int linearSearch(Comparable key) //metodo ausiliario
{   for (int i = 0; i < vSize; i++)
        if (v[i].getKey().compareTo(key) == 0)
        //oppure if (v[i].getKey().equals(key)), se il
        //metodo equals e` stato realizzato correttamente
            return i;
    throw new DictionaryItemNotFoundException();
}

//campi di esemplare
protected Pair[] v;
protected int vSize;
protected final static int INITSIZE = 10;
protected class Pair
{ ... } // codice della classe Pair
```

Dizionario con array ordinato

- Avendo usato un array non ordinato, i metodi **remove** e **find** effettuano una **ricerca lineare** sulle chiavi
- **Esercizio**: realizzare un dizionario con un array ordinato
  - Il metodo **insert** deve mantenere ordinato l'array a ogni inserimento (usando insertionSort...)
  - I metodi **remove** e **find** possono usare la **ricerca binaria** per trovare una chiave
  - Il metodo **remove** deve ricompattare l'array dopo la rimozione, mantenendolo ordinato

```
public class SortedArrayDictionary extends ArrayDictionary
{    // realizzazione con array non ordinato. Eredita campi di
     // esemplare e variabili statiche, la classe Pair, i metodi
     // isEmpty, makeEmpty, resize. Deve sovrascrivere i metodi
     // insert, remove, find
}
```

Collaudo di un dizionario

```
import java.util.Scanner;
import java.io.*;

public class SimpleDictionaryTester
{   public static void main(String[] args) throws IOException
    {   //creazione dizionario: leggo dati da file e assumo che
        //il file abbia righe nel formato <numero int> <stringa>
        Scanner infile =new Scanner(new FileReader("file.txt"));
        Dictionary dict = new ArrayDictionary();
        // ... oppure   = new SortedArrayDictionary();
        while (infile.hasNextLine())
        {   Scanner linescan = new Scanner(infile.nextLine());
            int key = Integer.parseInt(linescan.next());
            String value = linescan.next();
            dict.insert(key,value); //inserisco chiave e valore
        }
        infile.close();

        //ricerca/rimozione dati nel dizionario
        Scanner in = new Scanner(System.in);
        boolean done = false;                          // continua
```

Collaudo di un dizionario

```
        while (!done)                                          // continua
        {   System.out.println("**** Stampa dizionario ****");
            System.out.println(dict +"\nF=find,R=remove,Q=quit");
            String cmd = in.nextLine();
            if (cmd.equalsIgnoreCase("Q"))
            {    done = true;  }
            else if (cmd.equalsIgnoreCase("F"))
            {   System.out.println("Chiave da trovare?");
                int key = Integer.parseInt(in.nextLine());
                try{ //cerca key chiave e restituisce il valore
                    String value = (String)dict.find(key);
                    System.out.println("Valore: " + value); }
                catch(DictionaryItemNotFoundException e)
                {   System.out.println("Chiave non trovata");}
            }
            else if (cmd.equalsIgnoreCase("R"))
            {   System.out.println("Chiave da rimuovere?");
                int key = Integer.parseInt(in.nextLine());
                try{//rimuove la coppia identificata da key
                    dict.remove(key);
                    System.out.println("Chiave rimossa");  }
                catch(DictionaryItemNotFoundException e)
                {   System.out.println("Chiave non trovata");}
}   }   }   }
```