Realizzazione Mappa

Mappa

```
public interface Map extends Container{

   // inserisce un'associazione nella mappa
   // se esisteva gia' un'associazione con la stessa
   // chiave, sostituisco il valore vecchio e lo restituisco
   Object put(Object key, Object value);

   // restituisce l'elemento associato a key
   Object get(Object key);

   // restituisce l'elemento associato a key e
   // rimuove l'associazione
   Object remove(Object key);

   // restituisce un array con le chiavi contenute
   // nella mappa
   Object[] keys();

}
```

Mappa realizzata con array

- Una mappa può essere realizzata con un array riempito solo in parte
  - Ogni cella dell'array contiene un riferimento a una associazione chiave/valore
  - Un'associazione viene descritta da una classe specifica detta Pair (da definire)

Realizzazione della classe Pair

```
private class Pair // interna alla classe ArrayMap{

   private Object key;
   private Object value;

   public Pair(Object k, Object v){
      setKey(k);
      setValue(v);
   }

   public Object getKey()          {  return key;   }
   public Object getValue()        {  return value;}
   public void setKey(Object k)    {  key = k;      }
   public void setValue(Object v){  value = v;    }
}
```

Realizzazione della classe Pair se le chiavi sono Comparable

```
private class Pair // interna alla classe ArrayMap{

   private Object key;
   private Object value;

   public Pair(Comparable k, Object v){
      setKey(k);
      setValue(v);
   }

   public Object getKey()          {  return key;   }
   public Object getValue()        {  return value;}
   public void setKey(Object k)    {  key = k;      }
   public void setValue(Object v){  value = v;    }
}
```

Classi Interne

- La classe Pair può essere definita all'interno della classe ArrayMap

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

    private class Pair
    {... //costruttori, metodi, campi di
            //esemplare, variabili statiche
            //della classe interna Pair
    }
}
```

Classi interne

- Sintassi generale che ci interessa:

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

    <tipoaccesso> class ClasseInterna //<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 **ClasseEsterna** con la classe interna **ClasseInterna**, troviamo nella nostra cartella
  - Il file di bytecode **ClasseEsterna.class** (come al solito…)
  - Il file di bytecode **ClasseEsterna$ClasseInterna.class**

Classi interne: vantaggi e limiti

- 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 **ClasseInterna** è sempre associato a un oggetto di **ClasseEsterna**. Ovvero si possono creare oggetti di tipo **ClasseInterna** **solo dentro metodi non statici** di **ClasseEsterna**
  - La classe interna può essere resa **inaccessibile** al codice scritto in altre classi

Uso di classi interne
primo caso: dentro ClasseEsterna

```
public class ClasseEsterna{

      //campi di esemplare della classe esterna
      private double campoEsterno;

      //metodi della classe esterna
      public ClasseInterna metEsterno(){
            ClasseInterna obj = new ClasseInterna(); //LECITO
            obj.campointerno = 2;// LECITO: anche se il campo e`
            return obj;          // privato in ClasseInterna
      }

      //definizione di classe interna
      public class ClasseInterna{  //puo` anche essere private!

            //campi di esemplare della classe interna
            private int campointerno;

            //metodi della classe interna
            public void metInterno(){
            ClasseEsterna obj = new ClasseEsterna(); //LECITO
             double a = campoEsterno; //LECITO: anche se il campo
            }                         //  e` privato in ClasseEsterna
      }
}
```

CODICE

Uso di classi interne
secondo caso: in una classe diversa da ClasseEsterna

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

Uso di classi interne

secondo caso: in una classe diversa da ClasseEsterna

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

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

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

       //LECITO SOLO SE ClasseInterna e` public in ClasseEsterna
       ClasseEsterna.ClasseInterna i = e.metEsterno();

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

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

Prestazioni

- Analizziamo le prestazioni di una mappa realizzata con array e contenente n associazioni

- Assumiamo che le associazioni siano inserite senza un ordinamento particolare rispetto alle chiavi
  - Nella prima posizione libera dell'array
  - Utilizzabile qualsiasi sia il tipo della chiave

Mappa realizzata con array

- La ricerca ha prestazioni **O(n)**
  - Bisogna usare la ricerca sequenziale
- L'inserimento ha prestazioni **O(n)**
  - Prima devo fare una ricerca per chiave (lineare perche' non ordinato)
  - Se la chiave non c'è
    - Inserire la nuova associazione nell'ultima posizione dell'array (con eventuale ridimensionamento)
    - Aumentare la dimensione logica
  - **In assenza del vincolo di chiave unica,** l'inserimento sarebbe **O(1)** (in media, con analisi ammortizzata per eventuale ridimensionamento)
- La rimozione ha prestazioni **O(n)**
  - Prima devo fare una ricerca per chiave. Se la trovo, devo spostare nella posizione trovata l'ultimo elemento presente nell'array, e poi diminuire la dimensione logica dell'array

```
public class ArrayMap implements Map
{  // realizziamo con array non ordinato

   private Pair[] p;
   private int pSize;
   private static final int CAPACITY = 1;

   public ArrayMap()
   {  p = new Pair[CAPACITY]
      makeEmpty();
   }

   public void makeEmpty()
   { pSize = 0;
   }
   public boolean isEmpty()
   { return (pSize == 0);
   }
   public int size()
   {  return pSize;
   }
   . . .
}
```

```
public class ArrayMap implements Map
{  // realizziamo con array non ordinato
   . . .
   public Object[] keys()
   {
      Object[] keys = new Object[pSize];
      for(int i=0; i< pSize; i++)
         keys[i] = p[i].getKey();
      return keys;
   }

   public Object put(Object key, Object value)
   {
      if (key == null || value == null)
         throw new IllegalArgumentException();

      // elimino eventuale Pair esistente con stessa chiave
      Object old = remove(key);
      // se necessario ridimensiono
      if (pSize == p.length) p = resize(p, 2*pSize);
      // inserisco nuova coppia in fondo e inc. pSize
      p[pSize++] = new Pair(key, value);
      return old;
   }
   . . .
```

```
public class ArrayMap implements Map
{  // realizziamo con array non ordinato
   . . .
   public Object remove(Object key)
   {  for (int i = 0; i < pSize; i++)
      {  if (p[i].getKey().equals(key))
         {  Object obj = p[i].getValue();
            p[i] = p[pSize-1];
            pSize--;
            return obj;
         }
      }
      return null; // non c'e'
   }

   public Object get(Object key)
   {  for (int i = 0; i < pSize; i++)
         if (p[i].getKey().equals(key))
            return p[i].getValue();
      return null; // non c'e'
   }
}
```

Mappa con chiavi ordinabili

- Se si pone il vincolo aggiuntivo che le chiavi siano Comparable, è possibile definire un ADT "mappa ordinata", **SortedMap**
  - Aggiunge il solo metodo **sortedKeys**, che restituisce un array contenente le chiavi ordinate

```
public interface SortedMap extends Map
{
   Comparable[] sortedKeys();
}
```

- Le implementazioni di **sortedMap** devono però lanciare eccezioni ogni volta che viene fornita a **put** una chiave che non sia **Comparable**
  - Alternativa: definire **SortedMap** che non sia derivata da **Map**, ma sarebbe meno utile!

L'operatore *instance of*

- **l'operatore instance of** stabilisce se un oggetto è un'istanza del tipo specificato anche per le interfacce!

```
variabileOggetto instanceof NomeInterfaccia
```

- Restituisce un valore booleano
- Dipende dal tipo di oggetto in esecuzione, non dal tipo dichiarato del riferimento

Mappa con chiavi ordinabili

- I metodi che restituivano un riferimento *alla chiave* di tipo Object o Object[] restituiranno Comparable o Comparable[].
- I metodi get e remove useranno la ricerca binaria

- Provare ad implementare la classe SortedArraySortedMap che implementa SortedMap

SortedArraySortedMap

```
public class SortedArraySortedMap implements SortedMap
{
    private Pair[] p; // sempre ordinato per chiave
    private int pSize;

    public SortedArraySortedMap()
    {   p = new Pair[1]; // qualsiasi dimensione > 0
        pSize = 0;
    }

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

    public Comparable[] sortedKeys()
    {   Comparable[] keys = new Comparable[pSize];
        for (int i = 0; i < pSize; i++)
            keys[i] = (Comparable) p[i].getKey();
        return keys;
    }

    public Object[] keys()   // non è richiesto che siano     {
return sortedKeys(); } // "disordinate"... …
    . . .
```

SortedArraySortedMap

```
public class SortedArraySortedMap implements SortedMap
{ ...
    public Object put(Object key, Object value)
    {   if (key==null || value==null || !(key instanceof Comparable))
            throw new IllegalArgumentException();

        int pos = binSearch(key); // ricerca binaria
        if (pos >= 0)
        {   Object old = p[pos].getValue();
            p[pos] = new Pair(key, value); // nuova associazione
            return old;    // non si incrementa pSize
        }
        // ridimensionamento dinamico dell'array
        if (pSize == p.length)
            p = resize(p, 2*p.length);

        // inserimento nell'array ordinato
        int i = pSize - 1;
        while (i>=0 && ((Comparable)key).compareTo(p[i].getKey())< 0)
        {   p[i+1] = p[i];
            i--;
        }
        p[i+1] = new Pair(key, value); // inserisco nell'array
        psize++; // si incrementa pSize
        return null;
    }
    . . .
```

```
public class SortedArraySortedMap implements SortedMap
{ ...
    // i metodi get e remove usano la ricerca binaria
    public Object get(Object key)   // il controllo su Comparable lo fa
    {   int pos = binSearch(key);   // binSearch
        if (pos == -1)
            return null; // non c'è
        return p[pos].getValue();
    }

    public Object remove(Object key)
    {   int pos = binSearch(key);
        if (pos == -1)
            return null; // non c'è
        Object obj = p[pos].getValue();
        for (int i = pos; i < pSize-1; i++)
            p[i] = p[i+1]; // mantengo l'ordinamento
         pSize--;
        return obj;
    }

    private int binSearch(Object key)
    {   // NON statico così non si devono passare
        // p e pSize come parametri
        if (!(key instanceof Comparable))
            return -1;
        return binarySearch(p, 0, pSize - 1, key)
    }

    int binarySearch(. . .){. . .} // DA COMPLETARE
}
```

SortedMap in array ordinato

- Se le n coppie vengono conservate ordinate per chiave nell'array
  - La ricerca ha prestazioni O(log n)
    - Posso usare la ricerca binaria
  - L'inserimento ha prestazioni O(n)
    - Si usa l'ordinamento per inserimento in un array ordinato
      - Altrimenti, se si inserisce in fondo (e quindi non necessariamente in ordine) occorrerebbe poi ordinare l'intero array con prestazioni almeno O(n logn)
    - La rimozione ha prestazioni O(n)
      - Si fa una ricerca O(log n), poi si spostano in media n/2 elementi per mantenere l'ordinamento

SortedMap in array non ordinato

- Non è necessario realizzare una **SortedMap** con un array ordinato, si può usare la stessa implementazione con array non ordinato (**ArrayMap**) vista per **Map**
  - Chiamiamola **UnsortedArraySortedMap**
  - Le prestazioni dei metodi rimangono uguali a quelle di **ArrayMap**
  - Il metodo **sortedKeys** deve restituire un array ordinato, quindi deve effettuare l'ordinamento
- Cosa conviene fare?

Prestazioni di una sortedMap

SortedMap	In array ordinato	In array non ordinato
ricerca	O(log n)	O(n)
Inserimento	O(n)	O(n)
rimozione	O(n)	O(n)

Conviene **decisamente** realizzare una **SortedMap**
con un array **ordinato**

ADT Dizionario (Multimappa)

Dizionario

- L'**ADT dizionario** ha molte similitudini con l'**ADT mappa**
  - E' un contenitore di associazioni chiave/valore
  - Valgono tutte le proprietà dell'ADT mappa, tranne una
    - **Non** si richiede che le chiavi siano uniche nel dizionario
  - le chiavi possono essere **associate a più valori** e quindi comparire più volte nel dizionario

Dizionario

- 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 ad una classe che implementa **Comparable**
  - Nella realizzazione del dizionario non ordinato, per ricercare e confrontare si usa il metodo equals()
  - Nella realizzazione del dizionario ordinato, per ricercare e confrontare si usa il metodo compareTo()

Dizionario

```
public interface Dictionary extends Container
{
 /**
    Inserisce nel dizionario un'associazione avente
    chiave uguale a key e valore uguale a value.
    @param key la chiave specificata
    @param value il valore specificato
    @throws IllegalArgumentException se key o value sono null
  */
    void insert(Object key, Object value);

 /**
    Elimina dal dizionario un'associazione di chiave specificata
    restituendone il valore associato oppure null se non e'
    presente nel dizionario.
    @param key la chiave specificata
    @return un array con i valori associati alla chiave specificata,
    se presente, o un array vuoto altrimenti
 */
    Object remove(Object key);

}
```

Dizionario

```
public interface Dictionary extends Container
{
    . . .
  /**
    Se il dizionario contiene un'associazione avente
    chiave uguale a key, restituisce true,
    altrimenti restituisce false
    @param key la chiave specificata
    @return uno dei valori associati se la chiave specificata e'
        presente, altrimenti null
    */
    Object find(Object key);

    /**
    Se il dizionario contiene una o più associazioni aventi chiave
    uguale a key, restituisce i valori, altrimenti restituisce un
    array vuoto
    @param key la chiave specificata
    @return un array con i valori associati alla chiave specificata,
    se presente, o un array vuoto se non presente
    */
    Object[] findAll(Object key);

    . . .
}
```

Dizionario

```
public interface Dictionary extends Container
{
    . . .
  /**
    @return un array contenente le chiavi del
    dizionario, eventualmente ripetute. Restituisce un
    array vuoto (0 elementi) se il dizionario e' vuoto
    */
    Object[] keys();

    /**
    Se ci sono associazioni di chiave uguale a key, ne
    restituisce i valori, altrimenti restituisce un
    array vuoto.
    @param key la chiave specificata
    @return valori associati alla chiave specificata,
    se presente, o un array vuoto altrimenti
    */
    Object[] removeAll(Object key);

}
```

Prestazioni Dizionario con array con chiavi ordinate

- 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)**
    - se si usa l'ordinamento per **inserzione in un array ordinato**
    - con altre strategie, occorre invece ordinare l'intero array, con prestazioni, nel migliore dei casi, **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

Prestazioni Dizionario con array con chiavi NON ordinate

- Se le **n** chiavi vengono conservate non ordinate nell'array
  - la **ricerca** ha prestazioni **O(n)**
    - Si deve usare la **ricerca lineare**
  - l'**inserimento** ha prestazioni **O(1)**
    - è sufficiente inserire il nuovo elemento nell'ultima posizione dell'array
  - 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 ci interessa

Dizionario

	In array ordinato	In array non ordinato
ricerca	O(log n)	O(n)
inserimento	O(n)	O(1)
rimozione	O(n)	O(n)

- Con l'array ordinato
  - Ricerca efficiente
  - Inserimento non efficiente
- Con l'array non ordinato
  - Ricerca non efficiente
  - Inserimento efficiente

Prestazioni di un Dizionario

- La scelta di una realizzazione piuttosto di un'altra dipende dall'utilizzo tipico del dizionario nell'applicazione
  - se nel dizionario si fanno frequenti inserimenti e sporadiche ricerche e rimozioni
    - la scelta più opportuna è l'array non ordinato
  - se il dizionario viene costruito una volta per tutte, poi viene usato per fare soltanto ricerche
    - la scelta più opportuna è l'array ordinato