Le gerarchie di tipi

1

Sottotipo

Il tipo A e' un sottotipo del tipo B se A puo' fare tutto quello che puo' fare B.

B viene anche detto supertipo di A

```
public interface Area {
    public double getArea();
}

public class Circle implements Area {
    private double r;
    private Point p;
    public Circle (double x0, double y0, double r0) {
       r = r0; p = new Point(x0,y0);
    }
    public double getArea ()  {
       return 3.14159 * r * r;
    }
    public double getRadius () { return r; }
}
```

Supertipi e sottotipi

- ✓ un supertipo
  - `class`
  - `interface`
- ✓ può avere più sottotipi
  - un sottotipo `extends` il supertipo (`class`)
    - un solo supertipo (ereditarietà singola)
  - un sottotipo `implements` il supertipo (`interface`)
    - più supertipi `interface`
- ✓ la gerarchia può avere un numero arbitrario di livelli
- ✓ come si può utilizzare la gerarchia?

3

Come si può utilizzare una gerarchia di tipi

- ✓ implementazioni multiple di un tipo
  - i sottotipi non aggiungono alcun comportamento nuovo
  - la classe che implementa il sottotipo implementa esattamente il comportamento definito dal supertipo
- ✓ il sottotipo estende il comportamento del suo supertipo
  - fornendo nuovi metodi
- ✓ dal punto di vista semantico, supertipo e sottotipo sono legati dal principio di sostituzione
  - che definisce esattamente il tipo di astrazione coinvolto nella definizione di gerarchie di tipo

4

Principio di sostituzione

- ✓ un oggetto del sottotipo può essere sostituito ad un oggetto del supertipo senza influire sul comportamento dei programmi che utilizzano il tipo
  - i sottotipi supportano il comportamento del supertipo
  - API prevedono Reader e BufferedReader come classi di sistema per la lettura da socket. BufferedReader e' un sottotipo di Reader. Pertanto, un programma scritto in termini del tipo `Reader` può lavorare correttamente su oggetti del tipo `BufferedReader`
- ✓ il sottotipo deve soddisfare le specifiche del supertipo
- ✓ astrazione via specifica per una famiglia di tipi
  - astraiamo diversi sottotipi a quello che hanno in comune

5

Riassunto di tutte le puntate precedenti

- ✓ tipo apparente, tipo effettivo, dispatching dei metodi
- ✓ definizione di gerarchie di tipi
  - specifica
  - implementazione
- ✓ gerarchie di tipi in Java
  - supertipo: classe concreta
  - supertipo: classe astratta
  - supertipo: interfaccia
- ✓ implementazioni multiple
- ✓ semantica dei sottotipi (il principio di sostituzione)

6

Tipo apparente e tipo effettivo

- ✓ supertipo τ, sottotipo σ
- ✓ Sia Obj un oggetto di tipo σ
  - Assegnamento. Sia X: τ, X = Obj e' corretto.
  - può essere passato come parametro ad un metodo che ha un parametro formale di tipo τ
  - può essere ritornato da un metodo che ha un risultato di tipo τ
- ✓ in tutti i casi, abbiamo un tipo apparente τ ed un tipo effettivo σ
- ✓ il compilatore ragiona solo in termini di tipo apparente
  - per fare il controllo dei tipi
  - per verificare la correttezza dei nomi (di variabili e di metodi)
  - per risolvere i nomi

7

Tipo apparente e tipo effettivo

- ✓ supponiamo che il tipo `Poly` abbia un metodo di istanza `degree` senza argomenti che restituisce un intero e due sottotipi `DensePoly` e `SparsePoly`
  - implementazioni multiple
  - in cui è ridefinito (overriding) il metodo `degree`

```
Poly p1 = new DensePoly(); // il polinomio zero
Poly p2 = new SparsePoly(3,2);// il polinomio 3 . x^2
```

- ✓ `Poly` è il tipo apparente di `p1` e `p2`
- ✓ `DensePoly` e `SparsePoly` sono i loro tipi effettivi
- ✓ cosa fa il compilatore quando trova il seguente comando?

```
int d = p1.degree();
```

8

Tipo apparente e tipo effettivo

```
Poly p1 = new DensePoly();
...;
int d = p1.degree();
```

- ✓ il compilatore controlla che il metodo `degree` sia definito per il tipo apparente `Poly` di `p1`
  - guardando l'ambiente di metodi di istanza della classe `Poly`
- ✓ non può generare codice che trasferisce direttamente il controllo al codice del metodo
  - perché il metodo da invocare a tempo di esecuzione è determinato dal tipo effettivo di `p1` che non può essere determinato staticamente a tempo di compilazione
    - tra la dichiarazione ed il comando, il tipo effettivo di `p1` può essere stato modificato
- ✓ può solo generare codice che a run-time trova il metodo giusto e poi gli passa il controllo (dispatching)

9

Dynamic Dispatching

- ✓ a tempo di esecuzione gli oggetti continuano ad avere una parte dell'ambiente di metodi
  - un dispatch vector che contiene i puntatori al codice (compilato) dei metodi dell'oggetto
- ✓ il compilatore traduce il riferimento al nome in una posizione nel dispatch vector
  - la stessa per il dispatch vector del supertipo e dei suoi sottotipi
- ✓ e produce codice che ritrova l'indirizzo del codice del metodo da tale posizione e poi passa il controllo a quell'indirizzo

10

Definizione di una gerarchia di tipi: specifica

- ✓ specifica del tipo superiore della gerarchia
  - come quelle che già conosciamo
  - l'unica differenza è che può essere parziale
    - per esempio, possono mancare i costruttori
- ✓ specifica di un sottotipo
  - la specifica di un sottotipo è data relativamente a quella dei suoi supertipi
  - non si ridanno quelle parti delle specifiche del supertipo che non cambiano
  - vanno specificati
    - i costruttori del sottotipo
    - i metodi "nuovi" forniti dal sottotipo
    - i metodi del supertipo che il sottotipo ridefinisce
      - sono ammesse modifiche molto limitate

11

Definizione di una gerarchia di tipi: implementazione

- ✓ implementazione del supertipo
  - può non essere implementato affatto
  - può avere implementazioni parziali
    - alcuni metodi sono implementati, altri no
  - può fornire informazioni a potenziali sottotipi dando accesso a variabili o metodi di istanza
    - che un "normale" utente del supertipo non può vedere
- ✓ i sottotipi sono implementati come estensioni dell'implementazione del supertipo
  - la rep degli oggetti del sottotipo contiene anche le variabili di istanza definite nell'implementazione del supertipo
  - alcuni metodi possono essere ereditati
  - di altri il sottotipo può definire una nuova implementazione

12

Gerarchie di tipi in Java

- ✓ attraverso l'ereditarietà
  - una classe può essere sottoclasse di un'altra (la sua superclasse) e implementare 0 o più interfacce
- ✓ il supertipo (classe o interfaccia) fornisce in ogni caso la specifica del tipo
  - le interfacce fanno solo questo
  - le classi possono anche fornire parte dell'implementazione

13

Gerarchie di tipi in Java: supertipi 1

- ✓ i supertipi sono definiti da
  - classi
  - interfacce
- ✓ le classi possono essere
  - astratte
    - forniscono un'implementazione parziale del tipo
      - non hanno oggetti
      - il codice esterno non può chiamare i loro costruttori
      - possono avere metodi astratti la cui implementazione è lasciata a qualche sottoclasse
  - concrete
    - forniscono un'implementazione piena del tipo
- ✓ le classi astratte e concrete possono contenere metodi finali
  - non possono essere reimplementati da sottoclassi

14

Gerarchie di tipi in Java: supertipi 2

- ✓ le interfacce definiscono solo il tipo (specifica) e non implementano nulla
  - contengono solo (le specifiche di) metodi
    - pubblici
    - non statici
    - astratti

15

Gerarchie di tipi in Java: sottotipi 1

- ✓ una sottoclasse dichiara la superclasse che estende (e/o le interfacce che implementa)
  - ha tutti i metodi della superclasse con gli stessi nomi e segnature
  - può implementare i metodi astratti e reimplementare i metodi normali (purché non `final`)
  - qualunque metodo sovrascritto deve avere segnatura identica a quella della superclasse
    - ma i metodi della sottoclasse possono sollevare meno eccezioni
- ✓ la rappresentazione di un oggetto di una sottoclasse consiste delle variabili di istanza proprie e di quelle dichiarate per la superclasse
  - quelle della superclasse non possono essere accedute direttamente se sono (come dovrebbero essere) dichiarate `private`
- ✓ ogni classe che non estenda esplicitamente un'altra classe estende implicitamente `Object`

16

Gerarchie di tipi in Java: sottotipi 2

- ✓ la superclasse può lasciare parti della sua implementazione accessibili alle sottoclassi
  - dichiarando metodi e variabili `protected`
    - implementazioni delle sottoclassi piú efficienti
    - si perde l'astrazione completa, che dovrebbe consentire di reimplementare la superclasse senza influenzare l'implementazione delle sottoclassi
    - le entità `protected` sono visibili anche all'interno dell'eventuale package che contiene la superclasse
- ✓ meglio interagire con le superclassi attraverso le loro interfacce pubbliche

17

Un esempio di gerarchia con supertipo classe concreta

- ✓ in cima alla gerarchia c'è una variante di `IntSet`
  - la solita, con in più il metodo `subset`
  - la classe non è astratta
  - fornisce un insieme di metodi che le sottoclassi possono ereditare, estendere o sovrascrivere

18

Specifica del supertipo

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile di interi di
 // dimensione qualunque
 public IntSet ()
  // EFFECTS: inizializza this a vuoto
 public void insert (int x)
  // EFFECTS: aggiunge x a this
 public void remove (int x)
  // EFFECTS: toglie x da this
 public boolean isIn (int x)
  // EFFECTS: se x appartiene a this ritorna true, altrimenti false
 public int size ()
  // EFFECTS: ritorna la cardinalità di this
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 // this (come Integers) ciascuno una  sola volta, in ordine arbitrario
 // REQUIRES: this non deve essere modificato finché il generatore è in
 // uso
 public boolean subset (Intset s)
  // EFFECTS: se s è un sottoinsieme di this ritorna true, altrimenti
  // false
}
```


Implementazione del supertipo

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile di interi di
 // dimensione qualunque
 private Vector els; // la rappresentazione
 public IntSet () {els = new Vector();}
  // EFFECTS: inizializza this a vuoto
 private int getIndex (Integer x) {... }
  // EFFECTS: se x occorre in this ritorna la posizione in cui si
  // trova, altrimenti -1
 public boolean isIn (int x)
  // EFFECTS: se x appartiene a this ritorna true, altrimenti false
  {return getIndex(new Integer(x)) >= 0; }
 public boolean subset (Intset s)
  // EFFECTS: se s è un sottoinsieme di this ritorna true, altrimenti
  // false
  {if  (s == null) return false;
   for (int i = 0; i < els.size(); i++)
     if (!s.isIn(((Integer) els.get(i)).intValue()))
     return false;
   return true; }
}
```

✓ la rappresentazione è privata
  – i sottotipi non la possono accedere, ma c'è l'iteratore pubblico `elements`

Un sottotipo: `MaxIntSet`

- ✓ si comporta come `IntSet`
  - ma ha un metodo nuovo `max`
    - che ritorna l'elemento massimo nell'insieme
  - la specifica di Max`IntSet` definisce solo quello che c'è di nuovo
    - il costruttore
    - il metodo `max`
  - tutto il resto della specifica viene ereditato da `IntSet`
- ✓ perché non realizzare semplicemente un metodo `max` stand alone esterno alla classe `IntSet`?
  - facendo un sottotipo si riesce ad implementare `max` in modo più efficiente

21

Specifica del sottotipo

```
public class MaxIntSet extends IntSet {
 // OVERVIEW: un MaxIntSet è un sottotipo di IntSet che lo estende con
 // il metodo max
 public MaxIntSet ()
  // EFFECTS: inizializza this al MaxIntSet vuoto
 public int max () throws EmptyException
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  // ritorna l'elemento massimo in this
}
```

  - la specifica di Max`IntSet` definisce solo quello che c'è di nuovo
    - il costruttore
    - il metodo `max`
  - tutto il resto della specifica viene ereditato da `IntSet`

22

Implementazione di `MaxIntSet`

- ✓ per evitare di generare ogni volta tutti gli elementi dell'insieme, memorizziamo in una variabile di istanza di `MaxIntSet` il valore massimo corrente
  - oltre ad implementare `max`
  - dobbiamo riimplementare `insert` e `remove` per tenere aggiornato il valore massimo corrente
  - sono i soli metodi per cui c'è overriding
  - tutti gli altri vengono ereditati da `IntSet`

23

Implementazione del sottotipo 1

```
public class MaxIntSet {
 // OVERVIEW: un MaxIntSet è un sottotipo di IntSet che lo estende con
 // il metodo max
 private int mass; // l'elemento massimo, se this non è vuoto
 public MaxIntSet ()
  // EFFECTS: inizializza this al MaxIntSet vuoto
  { super( ); }
 ...  }
```

- ✓ chiamata esplicita del costruttore del supertipo
  - potrebbe in questo caso essere omessa
  - necessaria se il costruttore ha parametri
- ✓ nient'altro da fare
  - perché `mass` non ha valore quando `els` è vuoto

24

Implementazione del sottotipo 2

```
public class MaxIntSet extends IntSet {
 // OVERVIEW: un MaxIntSet è un sottotipo di IntSet che lo estende con
 // il metodo max
 private int mass; // l'elemento massimo, se this non è vuoto
 ...
 public int max () throws EmptyException
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  // ritorna l'elemento massimo in this
  {if (size( ) == 0) throw new
       EmptyException("MaxIntSet.max"); return mass;}
 ...  }
```

✓ usa un metodo ereditato dal supertipo (`size`)

Implementazione del sottotipo 3

```
public class MaxIntSet extends IntSet {
 // OVERVIEW: un MaxIntSet è un sottotipo di IntSet che lo estende con
 // il metodo max
 private int mass; // l'elemento massimo, se this non è vuoto
 ...
 public void insert (int x) {
   if (size() == 0 || x > mass) mass = x;
   super.insert(x); }
 ...  }
```

✓ ha bisogno di usare il metodo `insert` del supertipo, anche se overriden
  - attraverso il prefisso `super`
    - analogo a `this`

✓ per un programma esterno che usi un oggetto di tipo `MaxIntSet` il metodo overriden `insert` del supertipo non è accessibile in nessun modo

Implementazione del sottotipo 4

```
public class MaxIntSet extends IntSet {
 // OVERVIEW: un MaxIntSet è un sottotipo di IntSet che lo estende con
 // il metodo max
 private int mass; // l'elemento massimo, se this non è vuoto
 ...
 public void remove (int x) {
   super.remove(x);
   if (size() == 0 || x < mass) return;
   Iterator g = elements();
   mass = ((Integer) g.next()).intValue();
   while (g.hasNext() {
     int z = ((Integer) g.next( )).intValue();
     if (z > mass) mass = z; }
   return; }   }
```

✓ anche qui si usa il metodo overriden del supertipo
  – oltre ai metodi ereditati `size` e `elements`

Funzione di astrazione di sottoclassi di una classe concreta

✓ definita in termini di quella del supertipo
  – nome della classe come indice per distinguerle

✓ funzione di astrazione per `MaxIntSet`

```
// la funzione di astrazione è
// α_MaxIntSet(c) = α_IntSet(c)
```

✓ la funzione di astrazione è la stessa di `IntSet` perché produce lo stesso insieme di elementi dalla stessa rappresentazione (`els`)
  – il valore della variabile `mass` non ha influenza sull'astrazione

Invariante di rappresentazione di sottoclassi di una classe concreta

- ✓ invariante di rappresentazione per `MaxIntSet`

  // $I_{MaxIntSet}$ (c) = c.size() > 0 ==>
  // (c.mass appartiene a α_{IntSet}(c) &&
  // per tutti gli x in α_{IntSet}(c), x <= c. mass)

- ✓ l'invariante non include (e non utilizza in questo caso) l'invariante di `IntSet` perché tocca all'implementazione di `IntSet` preservarlo
  - le operazioni di `MaxIntSet` non possono interferire perché operano sulla rep del supertipo solo attraverso i suoi metodi pubblici
  - ma se implementiamo `repOk`, ...
- ✓ usa la funzione di astrazione del supertipo

29

`repOk` di sottoclassi di una classe concreta

- ✓ invariante di rappresentazione per `MaxIntSet`

  // $I_{MaxIntSet}$ (c) = c.size() > 0 ==>
  // (c.mass appartiene a α_{IntSet}(c) &&
  // per tutti gli x in α_{IntSet}(c), x <= c. mass)

- ✓ l'implementazione di `repOk` deve verificare l'invariante della superclasse perché la correttezza di questo è necessaria per la correttezza dell'invariante della sottoclasse

30

`repOk` di `MaxIntSet`

✓ invariante di rappresentazione per `MaxIntSet`

// $I_{MaxIntSet}$ (c) = c.size() > 0 ==>
// (c.mass appartiene a α_{IntSet}(c) &&
// per tutti gli x in α_{IntSet}(c), x <= c. mass)

```
public class MaxIntSet extends IntSet {
 // OVERVIEW: un MaxIntSet è un sottotipo di IntSet che lo estende con
 // il metodo max
 private int mass; // l'elemento massimo, se this non è vuoto
 ...
 public boolean repOk () {
   if (!super.rep0k ()) return false;
   if (size() == 0) return true;
   boolean found = false;
   Iterator g = elements();
   while (g.hasNext()) {
     int z = ((Integer) g.next()).intValue();
     if (z > mass) return false;
     if (z == mass) found = true; }
   return found; } }
```


Cosa succede se il supertipo fa vedere la rappresentazione?

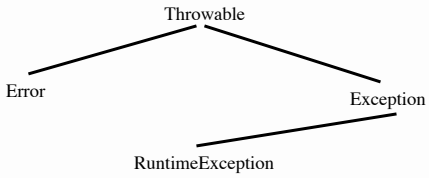
✓ l'efficienza di `remove` potrebbe essere migliorata
- questa versione richiede di visitare `els` due volte
  - per rimuovere l'elemento (attraverso la `remove` della superclasse)
  - per aggiornare il nuovo `mass` (utilizzando l'iteratore)

facendo vedere alla sottoclasse la rappresentazione della superclasse
- dichiarando `els protected` nell'implementazione di `IntSet`

✓ in questo caso, l'invariante di rappresentazione di `MaxIntSet` deve includere quello di `IntSet`
- perché l'implementazione di `MaxIntSet` potrebbe violarlo

// $I_{MaxIntSet}$ (c) = I_{IntSet} (c) && c.size() > 0 ==>
// (c.mass appartiene a α_{IntSet}(c) &&
// per tutti gli x in α_{IntSet}(c), x <= c. mass)

Ricostruiamo i tipi eccezione



- ✓ `Throwable` è una classe concreta (primitiva), la cui implementazione fornisce metodi che accedono una variabile istanza di tipo stringa
- ✓ `Exception` è una sottoclasse di `Throwable`
  - che non aggiunge nuove variabili istanza
- ✓ una nuova eccezione può avere informazione aggiuntiva nell'oggetto e nuovi metodi

33

Una eccezione non banale

```
public class MiaEccezione extends Exception {
 // OVERVIEW: gli oggetti di tipo MiaEccezione contengono un intero in
 // aggiunta alla stringa
 private int val; // l'elemento massimo, se this non è vuoto
 public MiaEccezione (String s, int x) {
   super (s); val = x; }
 public MiaEccezione (int x) {
   super (); val = x; }
 public int valoreDi () {
   return val;}}
```

34

Classi astratte come supertipi

- ✓ implementazione parziale di un tipo
- ✓ può avere variabili di istanza e uno o piú costruttori
- ✓ non ha oggetti
- ✓ i costruttori possono essere chiamati solo dalle sottoclassi per inizializzare la parte di rappresentazione della superclasse
- ✓ può contenere metodi astratti (senza implementazione)
- ✓ può contenere metodi regolari (implementati)
  - questo evita di implementare piú volte i metodi quando la classe abbia piú sottoclassi e permette di dimostrare più facilmente la correttezza
  - l'implementazione può utilizzare i metodi astratti
    - la parte generica dell'implementazione è fornita dalla superclasse
    - le sottoclassi forniscono i dettagli

35

Perché può convenire trasformare `IntSet` in una classe astratta

- ✓ vogliamo definire (come sottotipo di `IntSet`) il tipo `SortedIntSet`
  - il generatore `elements` fornisce accesso agli elementi in modo ordinato
  - un nuovo metodo `subset` (overloaded) per ottenere una implementazione più efficiente quando l'argomento è di tipo `SortedIntSet`
- ✓ vediamo la specifica di `SortedIntSet`

36

Specifica del sottotipo

```
public class SortedIntSet extends IntSet {
 // OVERVIEW: un SortedIntSet è un sottotipo di IntSet che lo estende
 // con i metodi max e subset e in cui gli elementi sono
 // accessibili in modo ordinato
 public SortedIntSet ()
  // EFFECTS: inizializza this all'insieme vuoto
 public int max () throws EmptyException
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  // ritorna l'elemento massimo in this
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 // this (come Integers) ciascuno una  sola volta, in ordine crescente
 // REQUIRES: this non deve essere modificato finché il generatore è in
 // uso
 public boolean subset (Intset s)
  // EFFECTS: se s è un sottoinsieme di this ritorna true, altrimenti
  // false }
```

- ✓ la rappresentazione degli oggetti di tipo `SortedIntSet` potrebbe utilizzare una lista ordinata
  - non serve piú a nulla la variabile di istanza ereditata da `IntSet`
  - il vettore `els` andrebbe eliminato da `IntSet`
  - senza `els`, `IntSet` non può avere oggetti e quindi deve essere astratta

37

`IntSet` come classe astratta

- ✓ specifiche uguali a quelle già viste
- ✓ dato che la parte importante della rappresentazione (gli elementi dell'insieme) non è definita qui, sono astratti i metodi `insert`, `remove`, `elements` e `repOk`
- ✓ `isIn`, `subset` e `toString` sono implementati in termini del metodo astratto `elements`
- ✓ `size` potrebbe essere implementata in termini di `elements`
  - inefficiente
- ✓ teniamo traccia nella superclasse della dimensione con una variabile intera `sz`
  - che è ragionevole sia visibile dalle sottoclassi (`protected`)
  - la superclasse non può nemmeno garantire proprietà di `sz`
    - il metodo `repOk` è astratto
- ✓ non c'è funzione di rappresentazione
  - tipico delle classi astratte, perché la vera implementazione è fatta nelle sottoclassi

38

Implementazione di `IntSet` come classe astratta

```
public abstract class IntSet {
 protected int sz; // la dimensione
// costruttore
 public IntSet () {sz = 0 ;}
// metodi astratti
 public abstract void insert (int x);
 public abstract void remove (int x);
 public abstract Iterator elements ( );
 public abstract boolean repOk ( );
// metodi
 public boolean isIn (int x)
 {Iterator g = elements ();
  Integer z = new Integer(x);
  while (g.hasNext())
   if (g.next().equals(z)) return true;
  return false; }
 public int size () {return sz; }
// implementazioni di subset e toString
}
```

39

Implementazione della sottoclasse SortedIntSet 1

```
public class SortedIntSet extends IntSet {
 private OrderedIntList els; // la rappresentazione
 // la funzione di astrazione:
 // α(c) = {c.els[1] , . . . , c.els[c.sz]}
 // l'invariante di rappresentazione:
 // I(c) = c.els != null && c.sz = c.els.size()

 // costruttore
 public SortedIntSet () {els = new OrderedIntList() ;}
// metodi
 public int max () throws EmptyException {
  if (sz == 0) throw new EmptyException("SortedIntSet.max");
  return els.max( ); }
 public Iterator elements ( ) {return els.elements(); }
// implementations of insert, remove, e repOk
 ...}
```

- ✓ la funzione di astrazione va da liste ordinate a insiemi
  - gli elementi della lista si accedono con la notazione []
- ✓ l'invariante di rappresentazione pone vincoli su tutte e due le variabili di istanza (anche quella ereditata)
  - `els` è assunto ordinato (perché così è in `OrderedIntList`)
- ✓ si assume che esistano per `OrderedIntList` anche le operazioni `size` e `max`

40

Implementazione della sottoclasse SortedIntSet 2

```
public class SortedIntSet extends IntSet {
 private OrderedIntList els; // la rappresentazione
 // la funzione di astrazione:
 // α(c) = {c.els[1] , . . . , c.els[c.sz]}
 // l'invariante di rappresentazione:
 // I(c) = c.els != null && c.sz = c.els.size()
.....
public boolean subset (IntSet s) {
 try { return subset((SortedIntSet) s); }
 catch (ClassCastException e) { return super.subset(s); }
public boolean subset (SortedIntSet s)
 // qui si approfitta del fatto che smallToBig di OrderedIntList
 // ritorna gli elementi in ordine crescente
}
```

41

Gerarchie di classi astratte

- ✓ anche le sottoclassi possono essere astratte
- ✓ possono continuare ad elencare come astratti alcuni dei metodi astratti della superclasse
- ✓ possono introdurre nuovi metodi astratti

42

Interfacce

- ✓ contengono solo metodi non statici, pubblici (non è necessario specificarlo)
- ✓ tutti i metodi sono astratti
- ✓ è implementata da una classe che abbia la clausola `implements` nell'intestazione
- ✓ un esempio che conosciamo: `Iterator`

```
public interface Iterator {
   public boolean hasNext ( );
// EFFECTS: restituisce true se ci sono altri elementi
// altrimenti false
   public Object next throws NoSuchElementException;
// MODIFIES: this
// EFFECTS: se ci sono altri elementi da generare dà il
// successivo e modifica lo stato di this, altrimenti
// solleva NoSuchElementException (unchecked)
    }
```


Ereditarietà multipla

- ✓ una classe può estendere soltanto una classe
- ✓ ma può implementare una o piú interfacce
- ✓ si riesce così a realizzare una forma di ereditarietà multipla
  - nel senso di supertipi multipli
  - anche se non c'è niente di implementato che si eredita dalle interfacce

```
public class SortedIntSet extends IntSet
               implements SortedCollection { .. }
```

- ✓ `SortedIntSet` è sottotipo sia di `IntSet` che di `SortedCollection`