



## LE GERARCHIE DI TIPI

1



## Gerarchie di tipi

- Le gerarchie di tipi permettono di definire **famiglie di tipi** costituite da un supertipo e dai suoi sottotipi
- Esempio: fornire implementazioni diverse di un supertipo (es: **Poly** con sottotipi **SparsePoly**, **DensePoly**)
- Un sottotipo può *estendere* il comportamento del supertipo (es: con nuovi metodi)

2

## Sottotipi e Principio di Sostituzione



- ☞ B è un sottotipo di A => ogni valore di tipo B può essere messo in qualunque contesto dove sia previsto un valore di tipo A
- ☞ Codice scritto usando il tipo A funziona correttamente con oggetti del sottotipo B
- ☞ Il **principio di sostituzione** fornisce l'**astrazione per specifica** per le famiglie di tipi, richiedendo che le istanze del sottotipo rispettino la specifica del supertipo
  - astraiano diversi sottotipi a quello che hanno in comune

3

## Sottotipo vs. Ereditarietà



- ☞ Concetto di *sottotipo* è una nozione semantica, definibile con il principio di sostituzione:
  - B è un sottotipo di A <-> un oggetto di B si può mascherare come un oggetto di A in tutti i possibili contesti
- ☞ Ereditarietà è una nozione di implementazione:
  - Permette di creare una nuova classe evidenziando solo le differenze (il codice nuovo)

4

## Gerarchia di tipi: specifica



- ✎ Specifica del tipo superiore della gerarchia
  - come quelle che già conosciamo
  - può essere parziale:
    - ✓ possono mancare i costruttori
    - ✓ specifica di operazioni può essere non deterministica
- ✎ Specifica di un sottotipo
  - è data relativamente a quella dei suoi supertipi
  - non si ripetono parti della specifica 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

5

## 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 (*protected*)
    - ✓ un “normale” utente del supertipo non li 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

6

## Gerarchie di tipi in Java



- Attraverso l'ereditarietà
  - una classe può essere sottoclasse di un'altra (la sua superclasse) e implementare zero 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

7

## Gerarchie di tipi in Java



- 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, es:

```
public abstract int size();
```

- Concrete** (forniscono un'implementazione piena del tipo)
- Le classi astratte e concrete possono contenere metodi finali
  - non possono essere reimplementati da sottoclassi

8

## Gerarchie di tipi in Java



- Le interfacce definiscono solo il tipo (specifica) e non implementano nulla
  - contengono solo (le specifiche di) metodi
    - ✓ pubblici → attributo **public** facoltativo
    - ✓ non statici
    - ✓ astratti → attributo **abstract** facoltativo

9

## Gerarchie di tipi in Java



- Una sottoclasse dichiara quale superclasse estende (e/o quali interfacce implementa)
  - ha tutti i metodi della superclasse con gli stessi nomi e segnature
  - può implementare i metodi astratti e reimplementare (**override**, **sovrascrivere**) i metodi normali (purché non **final**)
  - un metodo della sottoclasse deve avere **firma identica** a quello sovrascritto [**firma**: (*nome*, *tipo\_par\_1*, *tipo\_par\_2*, ...)], e tipo del risultato **covariante** (cioè **sottotipo** di quello sovrascritto) [da **Java 5**]
    - ✓ 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 dichiarate **private** (come dovrebbero essere)
- Ogni classe che non estenda esplicitamente un'altra classe estende implicitamente **Object**

10

## Gerarchie di tipi in Java



- ✎ La superclasse può lasciare parti della sua implementazione accessibili alle sottoclassi
  - dichiarando metodi e variabili **protected**
    - ✓ implementazioni delle sottoclassi più 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

11

## Esempio: 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 ha metodi astratti
  - fornisce un insieme di metodi che le sottoclassi possono ereditare, estendere o sovrascrivere

12

## 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 boolean subset (IntSet s)
// EFFECTS: se this è un sottoinsieme di s
// ritorna true, altrimenti false
}
```

13

## 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 this è un sottoinsieme di s 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; }
}
```

14

## Un sottotipo: `MaxIntSet`



- Si comporta come `IntSet`
  - ma ha un metodo nuovo `max`, che ritorna l'elemento massimo nell'insieme
  - la specifica di `MaxIntSet` definisce solo quello che c'è di nuovo
    - ✓ il costruttore e 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

15

## Specifica del sottotipo



```
public class MaxIntSet extends IntSet {  
    // OVERVIEW: un MaxIntSet è un sottotipo di IntSet che  
    // 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 `MaxIntSet` definisce solo quello che c'è di nuovo
  - ✓ il costruttore
  - ✓ il metodo `max`
- tutto il resto della specifica viene ereditato da `IntSet`

16



## 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**

17

## 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

18

## 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**)

19

## 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); }
    ... }

```

✎ Usa il metodo **insert** del supertipo, anche se overridden

- attraverso il prefisso **super**

✎ Per un programma esterno che usi un oggetto di tipo **MaxIntSet** il metodo overridden **insert** del supertipo non è accessibile

20

## Implementazione del sottotipo 4



```
public class MaxIntSet extends IntSet {  
    // OVERVIEW: un MaxIntSet è un sottotipo di IntSet che lo estende con  
    // il metodo max  
    ...  
    public void remove (int x) {...}
```

• Problema: come possiamo riscrivere **remove** se non abbiamo accesso agli elementi del super tipo?  
Lo discutiamo dopo!!

• Invariante di rappresentazione per `MaxIntSet`

```
//  $I_{\text{MaxIntSet}}(c) = c.\text{size}() > 0 \implies$   
//  $(c.\text{mass} \text{ appartiene a } a_{\text{IntSet}}(c) \ \&\&$   
// per tutti gli  $x$  in  $a_{\text{IntSet}}(c)$ ,  $x \leq c.\text{mass}$ 
```

21

## 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  
//  $a_{\text{MaxIntSet}}(c) = a_{\text{IntSet}}(c)$ 
```

• 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

22

## Invariante di rappresentazione di sottoclassi di una classe concreta



- invariante di rappresentazione per `MaxIntSet`

```
// IMaxIntSet(c) = c.size() > 0 ==>  
// (c.mass appartiene a aIntSet(c) &&  
// per tutti gli x in aIntSet(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
- Usa la funzione di astrazione del supertipo

23

## repOk di sottoclassi di una classe concreta



- invariante di rappresentazione per `MaxIntSet`

```
// IMaxIntSet(c) = c.size() > 0 ==>  
// (c.mass appartiene a aIntSet(c) &&  
// per tutti gli x in aIntSet(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

24

## Cosa succede se il supertipo fa vedere la rappresentazione?



- Il problema della **remove**: si potrebbe risolvere 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 a_IntSet(c) &&  
// per tutti gli x in a_IntSet(c), x <= c.mass)
```

25

## Classi astratte come supertipi



```
public abstract class GeometricShape{...}
```

- Implementazione parziale di un tipo
- Non ha oggetti
- Può avere variabili di istanza e uno o più costruttori, ma i costruttori possono essere chiamati solo dalle sottoclassi
- Può contenere metodi astratti (senza implementazione)
- Può contenere metodi regolari (implementati)
  - questo evita di implementare più volte i metodi quando la classe abbia più 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

26

## Perché può convenire trasformare **IntSet** in una classe astratta

- ☞ Vogliamo definire (come sottotipo di **IntSet**) il tipo **SortedIntSet**
  - un nuovo metodo **subset** (overloaded) per ottenere una implementazione più efficiente quando l'argomento è di tipo **SortedIntSet**
- ☞ vediamo la specifica di **SortedIntSet**

27

## Specifica di SortedSet

```
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 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 più 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

28

## 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**, e **repOk**
- **isIn**, **subset** e **toString** sono implementati in termini del metodo astratto **elements** (non visto: restituisce un *iteratore*)
- 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, **IntSet** non ha invariante di **rep**
- Non c'è funzione di astrazione
  - tipico delle classi astratte, perché la vera implementazione è fatta nelle sottoclassi

29

## 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 boolean repOk ( );

    // metodi
    public boolean isIn (int x)
        // implementazione}
    public int size () {return sz; }
    // implementazioni di subset e toString
}
```

30

# SortedIntSet



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

    // costruttore
    public SortedIntSet () {els = new OrderedIntList();}
    // metodi
    public int max () throws EmptyException {
        if (sz == 0) throw new EmptyException("SortedIntSet.max");
        return els.max(); }

    // 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**

31

# SortedIntSet



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

```

32



## 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

33

## Ereditarietà multipla



- Una classe può estendere soltanto una classe
  - Ereditare metodi concreti o variabili da più classi può causare ambiguità
- Ma può implementare una o più interfacce
  - Non ci sono problemi per metodi astratti o costanti
- 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**

34