



AA 2014-2015

## 12. Le gerarchie di tipi

1



### Sottotipo

- ☞ *B è un sottotipo di A: “every object that satisfies interface B also satisfies interface A”*
- ☞ Obiettivo metodologico: il codice scritto guardando la specifica di A opera correttamente anche se viene usata la specifica di B

2

## Sottotipi e principio di sostituzione



- ☞ B è un sottotipo di A: B può essere sostituito per A
  - una istanza del sottotipo soddisfa le proprietà del supertipo
  - una istanza del sottotipo può avere maggiori vincoli di quella del supertipo
- ☞ Questo non è sempre vero in Java

3

☞ Quella di sottotipo è una nozione semantica (specifica delle proprietà)

☞ B è un sottotipo di A se e solo se un oggetto di B si può mascherare come un oggetto di A in tutti i possibili contesti

☞ L'ereditarietà è una nozione di implementazione

- creare una nuova classe evidenziando solo le differenze (il codice nuovo)

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
  - per esempio, 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
  - astraiano diversi sottotipi a quello che hanno in comune

5

## Gerarchia di tipi: specifica



- Specifica del tipo superiore della gerarchia
  - come quelle che già conosciamo
  - l'unica differenza è che può essere parziale, e 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 definiscono nuovamente 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

6

## Gerarchia di tipi: implementazione



- ✎ Implementazione del supertipo
  - può non essere implementato affatto
  - può avere implementazioni parziali, ovvero alcuni metodi sono implementati e 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

7

## 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

8

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

9

## Gerarchie di tipi in Java



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

10

## Gerarchie di tipi in Java



- ✎ 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

11

## 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

12

## 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 è astratta
  - fornisce un insieme di metodi che le sottoclassi possono ereditare, estendere o sovrascrivere

13

## Specifiche 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 s è un sottoinsieme di this
        // ritorna true, altrimenti false
}
```

14

## 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;
    }
}
```

15

## 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

16



## 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 MaxIntSet definisce solo quello che c'è di nuovo
  - ✓ il costruttore
  - ✓ il metodo max
- tutto il resto della specifica viene ereditato da IntSet

17

## 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

18

## 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

19

## 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)

20

## 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 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

21

## 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) { ... }
```

- Problema: come possiamo riscrivere remove se non abbiamo accesso agli elementi del supertipo? Lo discutiamo dopo!!
- Invariante di rappresentazione per MaxIntSet
  - //  $I_{\text{MaxIntSet}}(c) = c.\text{size}() > 0 \implies$
  - //  $(c.\text{mass} \text{ appartiene a } AF_{\text{IntSet}}(c) \ \&\&$
  - // per tutti gli  $x$  in  $AF_{\text{IntSet}}(c)$ ,  $x \leq c.\text{mass}$

22

## 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  
//  $AF_{MaxIntSet}(c) = AF_{IntSet}(c)$ 
```

- 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

23

## Invariante di rappresentazione di sottoclassi di una classe concreta



- Invariante di rappresentazione per MaxIntSet

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

- 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

24

## repOk di sottoclassi di una classe concreta



- Invariante di rappresentazione per MaxIntSet

```
// I_MaxIntSet (c) = c.size() > 0 ==>  
// (c.mass appartiene a AF_IntSet(c) &&  
// per tutti gli x in AF_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

25

## 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 AF_IntSet(c) &&  
// per tutti gli x in AF_IntSet(c), x <= c.mass)
```

26

## Classi astratte come supertipi



- Implementazione parziale di un tipo
- Può avere variabili di istanza e uno o più 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 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

27

## 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

28

## 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 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

29

## 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
- 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 le 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

30

## 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( );
    public abstract Vector elements( );

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

```

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:
    //  $I(c) = c.els \neq \text{null} \ \&\& \ c.sz = c.els.size()$ 

    // 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

32



## SortedIntSet



```
public class SortedIntSet extends IntSet {
    private OrderedIntList els; // la rappresentazione
    // la funzione di astrazione:
    //  $\alpha(c) = \{c.els[0], \dots, c.els[c.sz-1]\}$ 
    // l'invariante di rappresentazione:
    //  $I(c) = c.els \neq null \ \&\& \ c.sz = c.els.size()$ 
    ...
    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
    }
}
```

33

## 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

34

## Ereditarietà multipla



- Una classe può estendere soltanto una classe
- Ma può implementare una o più 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

35



## Discussione finale

36

# Quadrati vs rettangoli



```
interface Rectangle {
    // effects: this_post.width = w, this_post.height = h
    void setSize(int w, int h);
}
interface Square extends Rectangle { ... }
```

Quale è la specifica ottimale per la specifica di `setSize` per `Square`?

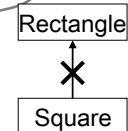
1. `// requires: w = h`  
`// effects: ... come sopra`  
`void setSize(int w, int h);`
2. `// effects: esattamente come sopra unico parametro`  
`void setSize(int edgeLength);`
3. `// effects: come sopra`  
`// throws BadSizeException if w != h`  
`void setSize(int w, int h) throws BadSizeException;`

# Discussione



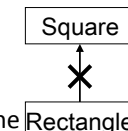
**Square** non è un sottotipo di **Rectangle**

- gli oggetti di **Rectangle** hanno variabili di istanza (base e altezza) indipendenti
- **Square** viola questa proprietà



**Rectangle** non è sottotipo di **Square**

- **Square** base e altezza sono identici
- **Rectangle** non vale questa proprietà



La nozione di sottotipo non è sempre quella suggerita dalla intuizione

Possibile soluzione

- aggiungere **Shape**
- trasformarli in oggetti immutabili

