

---

# **PROGRAMMAZIONE 2**

## **12. Gerarchie di tipi: implementazioni multiple e principio di sostituzione**

# Implementazioni multiple

---

- Il tipo superiore della gerarchia definisce una famiglia di tipi tale per cui
  - tutti i membri hanno esattamente gli stessi metodi e la stessa semantica che forniscono l'implementazione di tutti i metodi astratti, in accordo con le specifiche del super-tipo
  - gli oggetti dei sotto-tipi vengono visti dall'esterno come oggetti dell'unico super-tipo
  - dall'esterno si vedono solo i costruttori dei sotto-tipi

# IntList

---

- Il super-tipo è una classe astratta
- Usiamo i sotto-tipi per implementare i due casi della definizione ricorsiva
  - lista vuota
  - lista non vuota
- La classe astratta ha alcuni metodi non astratti
  - comuni alle due sotto-classi
  - definiti in termini dei metodi astratti
- La classe astratta non ha variabili di istanza e quindi nemmeno costruttori

# Super-tipo IntList

---

```
public abstract class IntList {
    // OVERVIEW: un IntList e' una lista modificabile
    // di Integers.
    // Elemento tipico [x1,...,xn]
    public abstract Integer first( ) throws EmptyException;
        // EFFECTS: se this e' vuoto solleva EmptyException,
        // altrimenti ritorna il primo elemento di this
    public abstract IntList rest( ) throws EmptyException;
        // EFFECTS: se this e' vuoto solleva EmptyException,
        // altrimenti ritorna la lista ottenuta da this
        // togliendo il primo elemento
    public abstract IntList addEl(Integer x);
        // EFFECTS: aggiunge x all'inizio di this
    public abstract int size( );
        // EFFECTS: ritorna il numero di elementi di this
    public abstract boolean repOk( );
    public String toString( ) {...}
    public boolean equals(IntList o) {...}
}
```

# Implementazione `EmptyIntList`

---

```
public class EmptyIntList extends IntList {
    public EmptyIntList( ) {...}
    public Integer first( ) throws EmptyException {
        throw new EmptyException("EmptyIntList.first");
    }
    public IntList rest( ) throws EmptyException {
        throw new EmptyException("EmptyIntList.rest");
    }
    public IntList addEl (Integer x) {
        return new FullIntList(x);
    }
    public int size( ) {...}
    public boolean repOk( ) {...}
}
```

# Implementazione FullIntList

---

```
public class FullIntList extends IntList {
    private int dim;
    private Integer val;
    private IntList next;
    public FullIntList(Integer x) {
        dim = 1; val = x;
        next = new EmptyIntList( );
    }
    public Integer first( ) { return val; }
    public IntList rest( ) { return next; }
    public IntList addEl(Integer x) {
        FullIntList n = new FullIntList(x);
        n.next = this; n.dim = this.dim + 1;
        return n;
    }
    public int size( ) {...}
    public boolean repOk( ) {...}
}
```

# Principio di sostituzione

---

- Un oggetto del sotto-tipo può essere sostituito da un oggetto del super-tipo senza influire sul comportamento dei programmi che utilizzano il tipo
  - i sotto-tipi supportano il comportamento del super-tipo
  - per esempio, un programma scritto in termini del tipo `IntList` può lavorare correttamente su oggetti del tipo `FullIntList`
- Il sotto-tipo deve soddisfare le specifiche del super-tipo
- Astrazione via specifica per una famiglia di tipi
  - astraiamo diversi sotto-tipi a quello che hanno in comune: la specifica del loro super-tipo

# Principio di sostituzione

---

- Devono essere supportate
  - la regola della segnatura
    - ✓ gli oggetti del sotto-tipo devono avere tutti i metodi del super-tipo
    - ✓ le signature dei metodi del sotto-tipo devono essere compatibili con le signature dei corrispondenti metodi del super-tipo
  - la regola dei metodi
    - ✓ le chiamate dei metodi del sotto-tipo devono comportarsi come le chiamate dei corrispondenti metodi del super-tipo
  - la regola delle proprietà
    - ✓ il sotto-tipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del super-tipo
- NB: le regole riguardano la semantica!

# Regola della segnatura

---

- Se una chiamata è type-correct per il super-tipo lo è anche per il sotto-tipo
  - garantita dal compilatore Java
    - che permette che i metodi del sotto-tipo sollevino meno eccezioni di quelli del super-tipo
    - da Java 5 un metodo della sotto-classe può sovrascrivere un metodo della super-classe con la stessa firma fornendo un return type più specifico
      - [docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.8.3](https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.8.3)
- le altre due regole non possono esser garantite dal compilatore Java...
  - dato che hanno a che fare con la specifica della semantica!

# Covariante vs. controvariante

---

- Una sotto-classe può riscrivere un metodo restituendo come risultato un valore di un sotto-tipo di quello previsto dal metodo della super-classe (covariant return type)
- Una nozione più liberale potrebbe avere argomenti contravarianti
  - un tipo è detto covariante se mantiene l'ordinamento dato dalla gerarchia dei tipi
  - controvariante se inverte l'ordinamento
  - invariante se non dipende dall'ordinamento

# Classi vs. Tipi

---

- Una classe definisce il comportamento degli oggetti (istanza)
  - il meccanismo di ereditarietà può modificare i comportamenti riscrivendo i metodi
- Un tipo definisce i comportamenti in termini del tipo dei metodi
- Sono concetti differenti e devono essere usati in modo coerente
  - Java/C# confondono le due nozioni: il nome di una classe è il tipo degli oggetti
  - nella pratica questa confusione è utile ma si deve tenere a mente che sono due nozioni differenti

# Regola dei metodi

---

- Si può ragionare sulle chiamate dei metodi usando la specifica del super-tipo anche se viene eseguito il codice del sotto-tipo
- Si è garantiti che va bene se i metodi del sotto-tipo hanno esattamente le stesse specifiche di quelli del super-tipo
- Come possono essere diverse?
  - se la specifica nel super-tipo è non-deterministica (comportamento sotto-specificato) il sotto-tipo può avere una specifica più forte che risolve (in parte) il non-determinismo

# Regola dei metodi

---

- In generale un sotto-tipo può indebolire le pre-condizioni e rafforzare le post-condizioni
- Per avere compatibilità tra specifiche del super-tipo e del sotto-tipo devono essere soddisfatte le regole
  - regola delle pre-condizione
    - ✓  $\text{pre}_{\text{super}} \implies \text{pre}_{\text{sub}}$
  - regola delle post-condizione
    - ✓  $\text{pre}_{\text{super}} \ \&\& \ \text{post}_{\text{sub}} \implies \text{post}_{\text{super}}$

# Regola dei metodi

---

- Ha senso indebolire la preconditione
  - $pre_{super} \implies pre_{sub}$perché il codice che utilizza il metodo è scritto per usare il super-tipo
  - ne verifica la pre-condizione
  - verifica anche la pre-condizione del metodo del sotto-tipo
- Esempio: un metodo in IntSet

```
public void addZero( )  
    // REQUIRES: this non e' vuoto  
    // EFFECTS: aggiunge 0 a this
```

potrebbe essere ridefinito in un sotto-tipo

```
public void addZero( )  
    // EFFECTS: aggiunge 0 a this
```

# Regola dei metodi

---

- Ha senso rafforzare la post-condizione
  - $\text{pre}_{\text{super}} \ \&\& \ \text{post}_{\text{sub}} \implies \text{post}_{\text{super}}$perché il codice che utilizza il metodo è scritto per usare il super-tipo
  - assume come effetti quelli specificati nel super-tipo
  - gli effetti del metodo del sotto-tipo includono comunque quelli del super-tipo (se la chiamata soddisfa la pre-condizione più forte)
- Esempio: un metodo in IntSet

```
public void addZero( )
    // REQUIRES: this non e' vuoto
    // EFFECTS: aggiunge 0 a this
```

potrebbe essere ridefinito in un sotto-tipo

```
public void addZero( )
    // EFFECTS: se this non e' vuoto aggiunge 0 a this
    // altrimenti aggiunge 1 a this
```

# Regola dei metodi: violazioni

---

- Consideriamo `insert` in `IntSet`

```
public class IntSet {  
    public void insert(int x)  
        // EFFECTS: aggiunge x a this
```

- Supponiamo di definire un sotto-tipo di `IntSet` con la seguente specifica di `insert`

```
public class StupidIntSet extends IntSet {  
    public void insert(int x)  
        // EFFECTS: aggiunge x a this se x è pari,  
        // altrimenti non fa nulla
```

# Regola delle proprietà

---

- Il ragionamento sulle proprietà degli oggetti basato sul super-tipo è ancora valido quando gli oggetti appartengono al sotto-tipo
- Sono proprietà degli oggetti (non proprietà dei metodi)
- Da dove vengono le proprietà degli oggetti?
  - dal modello del tipo di dato astratto
    - ✓ le proprietà degli insiemi matematici, etc.
    - ✓ le elenchiamo esplicitamente nell'overview del super-tipo
      - un tipo astratto può avere un numero infinito di proprietà
- Proprietà invarianti
  - un FatSet non è mai vuoto
- Proprietà di evoluzione
  - il numero di caratteri di una String non cambia

# Regola delle proprietà

---

- Per mostrare che un sotto-tipo soddisfa la regola delle proprietà dobbiamo mostrare che preserva le proprietà del super-tipo
- Per le proprietà invarianti
  - bisogna provare che creatori e produttori del sotto-tipo stabiliscono l'invariante (solita induzione sul tipo)
  - che tutti i metodi (anche quelli nuovi, inclusi i costruttori) del sotto-tipo preservano l'invariante
- Per le proprietà di evoluzione
  - bisogna mostrare che ogni metodo del sotto-tipo le preserva

# Regola delle proprietà: una proprietà invariante

---

- Il tipo `FatSet` è caratterizzato dalla proprietà che i suoi oggetti non sono mai vuoti

```
`// OVERVIEW: un FatSet e' un insieme modificabile di interi  
// la cui dimensione e' sempre almeno 1
```

- Assumiamo che `FatSet` non abbia un metodo `remove`, ma invece abbia un metodo `removeNonEmpty`

```
public void removeNonEmpty (int x)  
    // EFFECTS: se x e' in this e this contiene altri elementi  
    // rimuovi x da this
```

e abbia un costruttore che crea un insieme con almeno un elemento. Si può provare che gli oggetti `FatSet` hanno dimensione maggiore di 0?

# Regola delle proprietà: una proprietà invariante

---

- Consideriamo il sotto-tipo ThinSet che ha tutti i metodi di FatSet con identiche specifiche e in aggiunta il metodo

```
public void remove(int x)
    // EFFECTS: rimuove x da this
```

- ThinSet non è un sotto-tipo legale di FatSet
  - perché il suo extra metodo può svuotare l'oggetto, e
  - l'invariante del super-tipo non sarebbe conservato

# Una proprietà di evoluzione (non modificabilità)

---



- Il tipo SimpleSet ha i due soli metodi insert e isIn
  - gli oggetti di SimpleSet possono solo crescere in dimensione
  - IntSet non può essere un sotto-tipo di SimpleSet perché il metodo remove non conserva la proprietà