

Estensioni

# Estensioni pure

- Estensione è il termine Java per ereditarietà
  - Un'estensione C' di una classe C è un erede di C
- Un'estensione è detta **pura** se non modifica la **specificità** dei metodi ereditati
  - Un'estensione pura può quindi solo estendere la specificità, aggiungendo nuove operazioni
  - Può modificare (a parità di specificità) l'implementazione dei metodi ereditati

# Estensione pura MaxIntSet

- Estensione di IntSet, che aggiunge solo il metodo per estrarre il massimo elemento
  - La specifica di tutti gli altri metodi è inalterata
  - Il costruttore non è ereditato, quindi potrebbe cambiare

```
public class MaxIntSet extends IntSet {  
    // MaxIntSet è sottotipo di IntSet con aggiunta metodo max  
    //@ensures (*inizializza this come il MaxIntSet vuoto*)  
    public MaxIntSet(){...}  
  
    //@ ensures this.size()>0 &&\result == (\max int i; this.isIn(i);i);  
    //@ signals(EmptyException e) this.size() == 0;  
    public int /* pure */ max() throws EmptyException {...}
```

# MaxIntSet: Implementazione

```
public class MaxIntSet extends IntSet {  
    private int biggest; // Memorizza il max. Non definito per this=vuoto  
    public MaxIntSet(){ super(); }  
  
    public int max() throws EmptyException {  
        if (size() == 0) throw new EmptyException("MaxIntSet.max");  
        return biggest;  
    }  
    @ Override  
    public void insert(int x) {  
        if (size() == 0 || x > biggest) biggest = x;  
        super.insert(x); //chiama metodo insert di IntSet  
    }  
    @ Override  
    public void remove(int x) {  
        super.remove(x);  
        if (size() == 0 || x < biggest) return;  
        for (Iterator<Integer> i = this.iterator(); i.hasNext();) {  
            Integer z = i.next();  
            if (z > biggest) biggest=z;  
        }  
    }  
}
```

Cambiamo solo  
l'implementazione,  
mantenendo  
inalterata la  
specifica!

Cambiamo solo  
l'implementazione,  
mantenendo  
inalterata la  
specifica!

# AF e RI?

- AF di sottoclasse **di solito** è uguale a quella della superclasse
  - AF di MaxIntSet è identica a AF di IntSet:  
MaxIntSet utilizza ancora IntSet per memorizzare gli elementi
- RI della classe base è ereditato senza modifiche e si aggiunge un nuovo RI

// @ also

// @ private invariant this.size > 0 ==> this.isIn(this.biggest) &&

// @ (\forall int x; this.isIn(x); x <= this.biggest);

- Valgono le consuete regole di visibilità, RI può usare
  - Tutti gli attributi e metodi puri (anche private) definiti nella classe stessa
  - Attributi e metodi puri ereditati, purché pubblici o protected

# Estensioni non pure

- È modificata la specifica di uno o più metodi ereditati
- Esempio: SortedIntSet

```
public class SortedIntSet extends IntSet {  
    public SortedIntSet {super();}  
    @ Override  
    //@ ensures (*restituisce un iteratore su tutti e soli elementi di  
    //@ this, ciascuno una sola volta, in ordine crescente *);  
    public Iterator<Integer> iterator() {...}  
}
```

- Le estensioni non pure di classi concrete **andrebbero evitate** o quantomeno limitate

# Principio di sostituzione di Liskov

- Gli oggetti della sottoclasse devono **rispettare il contratto** della superclasse
  - Il contratto può essere esteso per coprire ulteriori casi, ma non cambiato
- Questo garantisce che moduli che usano oggetti di un tipo devono potere usare oggetti di un tipo derivato "senza accorgersi della differenza"

# Principio di Sostituzione (OK)

- L'estensione pura `MaxIntSet` soddisfa il contratto di `IntSet`
  - Ha tutti i metodi di `IntSet`, ciascuno con la stessa specifica di `IntSet`
  - Ha in più il metodo `max`, ma un utilizzatore di `IntSet` comunque non lo userà
- `IntSet` offre un iteratore che itera su tutti e soli gli elementi di `this`, ciascuno una sola volta, in ordine qualunque

```
public static int traccia(IntSet x) {  
    int somma=0;  
    for (Iterator<Integer> i = this.iterator(); i.hasNext();)   
        somma+=i.next();  
    return somma;  
}
```

- Poiché `MaxIntSet` ha ancora il metodo `iterator()` con la stessa specifica, il codice di `traccia(...)` non sarà “sorpreso” se gli viene passato un `MaxIntSet` al posto di un `IntSet`
  - Permette di utilizzare con tranquillità polimorfismo e binding dinamico...



# Principio di Sostituzione (KO)

- Se invece `iterator()` restituisse un iteratore che itera su tutti gli elementi dispari di `this`, ciascuno una sola volta, in ordine crescente
  - Il codice utilizzatore `traccia()` non calcolerebbe più la traccia dell'insieme!
  - Il metodo non è compatibile con quello di `IntSet()`: non restituisce gli elementi di valore pari
- Comportamento scorretto!
- Il motivo è che il contratto di `iterator()` nella classe estesa non mantiene la promessa fatta dal contratto di `iterator()` in `IntSet`

# Regole per la specifica di sottoclassi

- Come garantire che il contratto dell'estensione sia compatibile con quello della superclasse?
  - Principio di sostituzione impone vincolo sulla specifica del sottotipo: deve essere “compatibile” con specifica del supertipo
  - Si parla di **equivalenza comportamentale** (behavioral) tra esemplari della superclasse e della sottoclasse
- Basta mostrare le seguenti tre proprietà della specifica dei tipi per compatibilità
  - **Regola della segnatura**: un sottotipo deve avere tutti i metodi del supertipo, e signature dei metodi del sottotipo devono essere compatibili
  - **Regola dei metodi**: le chiamate ai metodi del sottotipo devono comportarsi come le chiamate ai metodi corrispondenti del supertipo
  - **Regola delle proprietà**: un sottotipo deve preservare tutti i public invariant degli oggetti del supertipo

# Ovvero...

- Regola della segnatura: garantisce che il contratto della superclasse sia ancora applicabile, ossia che la sintassi della sottoclasse sia compatibile con la sintassi della superclasse
  - Esempio: il metodo non cambia prototipo
- Regola dei metodi: verifica che il contratto dei singoli metodi ereditati sia compatibile con il contratto dei metodi originali
  - Esempio: una estensione pura non cambia specifica
- Regola delle proprietà: verifica che la specifica nel suo complesso sia compatibile con quella originale

# Regola della segnatura

- Garantisce type-safety, ossia che ogni chiamata corretta (senza errori di tipo) per il supertipo sia corretta anche per il sottotipo
  - Correttezza verificabile staticamente
    - Esempio: traccia(...) chiama il metodo iterator(): questo deve essere presente nell'estensione, con un prototipo utilizzabile da traccia(...)
- In Java (versioni <1.5), la regola delle segnatura diventa
  - Le signature dei metodi del sottotipo devono essere identiche alle corrispondenti del supertipo
    - ...però un metodo del sottotipo può avere meno eccezioni nella segnatura
  - In realtà la regola della segnatura in Java è inutilmente restrittiva
- In Java (da 1.5 in poi) consente di restringere il tipo del valore ritornato
  - Ossia consente la **Covarianza** del risultato

# Esempio

```
public class Line {...} // Una linea che collega dei punti
public class ColoredLine extends Line { // Una Line con un colore
    private Colore col;
    ...
}
public class Point {private double x; private double y;...} // Un punto
(immutabile)
public class GridPoint extends Point{ // Punto su una griglia
    public Line newLine(GridPoint p)// \result è una linea da this a p
}
public class ColoredGridPoint extends GridPoint {
    private Colore col;
    // \result è una ColoredLine colorata come this.col
    @Override
    public Line newLine(GridPoint p)
}
```

La (ri)definizione di  
newLine è corretta  
anche se a run-time  
restituisce una  
ColoredLine...

# Invarianza del risultato e dei parametri

```
public static void usaPunto(GridPoint punto1) {  
    Line lin= punto1.newLine(new GridPoint(...));  
}
```

```
main(...) {  
    usaPunto(new ColoredGridPoint(...));  
}
```

- La chiamata è lecita in Java perché ColoredGridPoint è sottotipo di GridPoint
- Ridefinizione di newLine ha la stessa signature dell'originale, ossia:

```
@Override // result è una ColoredLine colorata come this.col  
public Line newLine(GridPoint p)
```

- Allora usaPunto(new ColoredGridPoint(...)); invoca la nuova newLine, che riceve in ingresso un GridPoint e restituisce una ColoredLine
- Sappiamo che è corretto in Java

# Covarianza del risultato

```
public static void usaPunto(GridPoint punto1) {  
    Line lin= punto1.newLine(new GridPoint(...));  
}  
main(...) {  
    usaPunto(new ColoredGridPoint(...));  
}
```

- Se ridefiniamo newLine come

```
@Override // \result è una ColoredLine colorata come this  
public ColoredLine newLine(GridPoint p)
```

- Allora usaPunto(new ColoredGridPoint(...)) invocherebbe la nuova newLine, che riceve in ingresso un GridPoint e restituisce una ColoredLine
- ColoredLine è sottotipo di Line e quindi lin a run-time può riferire un oggetto di tipo ColoredLine

# Regola dei metodi

- Le chiamate ai metodi del sottotipo devono comportarsi come le chiamate ai metodi corrispondenti del supertipo
- **Non** può essere verificata dal compilatore
- Perché la chiamata a un metodo del sottotipo abbia lo stesso effetto, basta che la specifica sia la stessa
  - ...quindi tutto ok se estensioni pure
- Spesso è necessario cambiare la specifica
  - Esempio: `iterator()` in `SortedIntSet()` non restituisce gli elementi in un ordine qualsiasi ma in ordine crescente
  - In questo caso, è corretto farlo, perché la ensures di `IntSet.iterator()` era nondeterministica: qualunque ordine in cui si ritornano gli elementi andava bene; allora ritornarli in ordine crescente è accettabile (rafforziamo la postcondizione)
- Valgono in generale regole di
  - Precondizione più **debole**
  - Postcondizione più **forte**



# Forza e debolezza delle condizioni (1)

- Che significano?
  - **Forte**: più restrittivo, meno facile da rendere vero, verificato “in meno casi”
  - **Debole**: meno restrittivo, più facile da rendere vero, verificato “in più casi”
  - Esempio:  $x > 15$  è più forte di  $x > 7$ , detto altrimenti  $\{x \mid x > 15\} \subseteq \{x \mid x > 7\}$
- Come si formalizza in Logica Matematica la “forza” di una condizione?
  - Le condizioni (formule logiche) sono ordinate (anche se non totalmente): dalle più forti (vere in meno casi) alle più deboli (vere in più casi)
    - A un estremo **false**, la condizione più forte di tutte
    - All’altro estremo **true**, la più debole che ci sia
- L’operatore logico per indicare che A è più forte di B è l’implicazione  $A \implies B$
- L’implicazione logica è una specie di operatore di relazione che confronta il valore di verità delle formule
- L’implicazione corrisponde all’inclusione insiemistica dei “valori che rendono vere le formule”
  - ...dei “casi” in cui la formula è vera, o modelli
  - Es: se A è  $x > 15$ , B è  $x > 7$ , vale l’implicazione  $A \implies B$

# Effetto degli operatori logici

- Disgiunzione (OR,  $||$ ) indebolisce
  - Rispetto alla formula A, la formula  $A || B$  è vera in qualche caso in più: quelli in cui è vera B
    - Esempio  $(1 < x < 10)$  è più forte di  $(1 < x < 10) || (20 < x < 30)$
- Congiunzione (AND,  $\&\&$ ) rafforza
  - Rispetto alla formula A, la formula  $A \&\& B$  è vera in qualche caso in meno: quelli in cui è vera A ma non è vera B
    - Esempio:  $(1 < x < 10)$  è più debole di  $(1 < x < 10) \&\& \text{dispari}(x)$
- Implicazione (passare da B ad  $A \implies B$ ) indebolisce
  - Rispetto alla formula B, la formula  $A \implies B$  è vera in qualche caso in più: quelli in cui A è falsa
    - ...infatti  $A \implies B$  equivale a  $!A || B$
    - Esempio  $(1 < x < 10)$  è più forte di  $\text{dispari}(x) \implies (1 < x < 10)$
    - Soddisfatte rispettivamente da  $[2 .. 9]$  e da  $[2 .. 9] || \{x \mid \text{pari}(x)\}$

# Precondizione più debole

- Se la precondizione del metodo ridefinito è più debole di quella del metodo originale, allora tutti i casi in cui si chiamava il metodo originale si può chiamare anche il metodo ridefinito
- Se specifichiamo un metodo indebolendo la precondizione, il chiamante la verifica a fortiori
- **Regola della precondizione:**  $\text{pre\_super} \Rightarrow \text{pre\_sub}$

# Se la preconditione fosse più forte?

```
public class Stack<T> {  
    // Una pila di elementi di tipo T  
    //..  
    ...  
    //@requires true;  
    //@ensures (* inserisce v in cima a this *);  
    public void push(T v)  
    ...  
}
```

```
public class BoundedStack<T> {  
    // Una pila di elementi di tipo T  
    // di dimensione max pari a cento  
    ...  
    //@requires this.size() <=100;  
    //@ensures (* inserisce v in cima alla pila *);  
    public void push(T v)  
    ...  
}
```

- Se BoundedStack estendesse Stack, allora codice che usa Stack potrebbe non funzionare: ad esempio se provassimo ad aggiungere 200 elementi
  - La procedura funzionerebbe con uno Stack, ma non con un BoundedStack, violando il principio di sostituzione.
- Ovviamente Java consente di definire BoundedStack come erede di Stack, ma la sua implementazione violerà la specifica

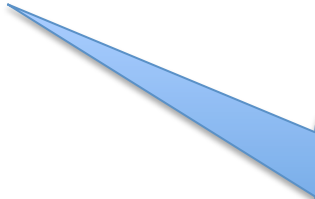
# Postcondizione più forti

- Se rafforziamo postcondizione, allora la postcondizione attesa dal chiamante sarà comunque verificata
- Poiché al termine dell'esecuzione del metodo, `post_sub` deve valere, anche `post_super` vale, e l'utilizzatore del supertipo non è sorpreso
- **Regola della postcondizione:** `post_sub ==> post_super`

# Esempio dal calcolo numerico

```
public class Num {  
    //@ requires r > 0  
    //@ ensures abs(\result * \result - r) < 0.1  
    public static float sqrt(float r) {...}  
}
```

```
public class NumPreciso {  
    //@ requires r > 0  
    //@ ensures abs(\result * \result - r) < 0.01  
    public static float sqrt(float r) {...}  
}
```



NumPreciso può essere  
definito come erede di  
Num, la sua post-  
condizione rafforza  
quella originale

# Violazione della regola dei metodi

```
public class CharBuffer {  
  // Un buffer di caratteri.  
  ...  
  //ensures this.isIn(x) &&...  
  public void insert(char x) {...}  
  ...  
}
```

```
public class LowerCaseCharBuffer {  
  // Un buffer di caratteri minuscoli  
  ...  
  // ensures minuscolo(x) ==> this.isIn(x) &&...  
  // se x e' minuscolo inserisce x in this, se no  
  // non assicura nulla  
  public void insert(char x) {...}  
  ...  
}
```

- Se LowerCaseCharBuffer venisse definito come sottoclasse di CharBuffer, il principio di sostituzione verrebbe **violato**
- La regola della segnatura è verificata, ma la regola dei metodi **no**: la post di insert non inserisce caratteri minuscoli: è più debole

# JML ed estensioni

- Come si ridefinisce in JML la specifica di un metodo?
  - Una sottoclasse eredita pre e postcondizioni dei metodi pubblici e protetti della superclasse e i suoi invarianti pubblici
- Sintatticamente: tutto come al solito, basta aggiungere `//@also`  
`//@also`  
`//@ensures .....`  
`//@requires .....`
- Clausole aggiuntive interpretate in modo da rispettare la regola dei metodi
  - La nuova postcondizione si applica solo nel caso in cui valga la nuova preconditione



# Semantica in JML

- La parte `//@requires` della classe erede va in OR (`||`, disgiunzione) con quella della classe padre: `requires` risulta **indebolita**
- La parte `//@ensures` della classe erede è messa in AND con quella della classe padre: `ensures` risulta **rafforzata**
  - `Requires Pre_super || Pre_sub`
  - `Ensures (\old(Pre_super) ==> Post_super)) && (\old(Pre_sub) ==> Post_sub)`

# Esempio (1)

- Contratto di un metodo della classe Baratto: “se mi dai almeno dieci patate allora ti do almeno due fragole”

```
int fragole(int patate)
//@requires patate >= 10
//@ensures \result >= 2
```

- Esempio di ridefinizione del metodo in una estensione di Baratto: “tutto come prima, ma inoltre se mi dai almeno cinque patate ti do almeno una fragola”

```
//@also
//@requires patate >= 5
//@ensures \result >= 1
```

- La ridefinizione ha la seguente specifica

```
//@requires patate >= 10 || patate >= 5
//@ensures (patate >= 10 ==> \result >= 2) && (patate >= 5 ==> \result >= 1)
```

- Quindi... “se mi dai almeno dieci patate ti do almeno due fragole, altrimenti se mi dai almeno cinque patate ti do almeno una fragola”

# Esempio (2)

- Se volessi ridefinire fragole indebolendo la postcondizione, cioè dicendo che voglio restituire meno fragole?

```
//@also  
//@requires patate >= 10  
//@ensures \result >= 1
```

- La ridefinizione ha la seguente specifica

```
//@requires patate>=10 || patate >=10  
//@ensures (patate>=10 ==> \result >= 2) && (patate>=10 ==> \result >=1)
```

- Cioè

```
//@requires patate>=10  
//@ensures (patate>=10 ==> \result >= 2)
```

- Infatti **non è possibile** indebolire la postcondizione in JML

# Esempio (3)

- Se volessi ridefinire fragole rafforzando la postcondizione, cioè dicendo che voglio restituire più fragole?

```
//@also  
//@requires patate >=10  
//@ensures \result>=4
```

- La ridefinizione ha la seguente specifica

```
//@requires patate>=10 || patate >=10  
//@ensures (patate>=10 ==> \result >= 2) && (patate>=10 ==> \result >=4)
```

- cioè

```
//@requires patate>=10  
//@ensures (patate>=10 ==> \result >= 4)
```

- Quindi per rafforzare la postcondizione, a parità di preconditione, basta ripetere la preconditione originale e aggiungere la nuova postcondizione più forte

# Regola completa della postcondizione

- La regola della “postcondizione più forte” vale quindi solo dove è verificata la preconditione originale
  - Il codice che usa il metodo della superclasse non chiama il metodo nei casi aggiunti dalla preconditione più debole nella versione della sottoclasse
- La regola completa della postcondizione è sempre verificata in JML, per il modo in cui sono unite le specifiche del metodo nella superclasse e nella sottoclasse

# Esempio di regola completa

- Esempio: metodo addZero() di IntSet aggiunge 0 solo negli insiemi non vuoti  
//@ requires this.size() > 0  
//@ ensures this.isIn(0) &&...  
public void addZero()
- Definiamo sottotipo di IntSet in cui addZero() è ridefinito in modo che se l'insieme è vuoto viene aggiunto 1  
//@ also  
//@ requires this.size() = 0  
//@ ensures this.isIn(1)  
public void addZero()
- La clausola ensures è diversa solo per quei valori (this = vuoto) che non erano legali per addZero() della superclasse: il codice che usa addZero non è sorpreso dalla modifica (non chiamerà mai addZero con this == vuoto)
- In JML, la postcondizione corrisponde a:  
$$-(\text{old}(\text{size}) > 0) \implies \text{isIn}(0) \text{ ) } \&\& (\text{old}(\text{size}) = 0 \implies \text{isIn}(1))$$

# Esempio con tipi primitivi

- `int` (Integer) è sottotipo di `long` (Long)? O viceversa?
- La somma di due `int` a 32 bit in caso di overflow lancia eccezione, mentre la somma degli stessi numeri rappresentati come `long` a 64 bit calcola il risultato
- Non possono essere uno sottotipo dell'altro perché le rispettive postcondizioni della somma non si implicano

# Riassumendo

- Regola dei metodi può essere enunciata come: **require no more, promise no less**
- JML impedisce di specificare astrazioni sui dati che violino la regola dei metodi
  - Ci sono comunque altri metodi di specifica che lo consentono
  - In JML, al massimo si può costruire una specifica non soddisfacibile (pre e/o post condizioni sono false per tutti i valori)
- Tuttavia, una classe in Java può essere implementata in modo scorretto e violare la regola



# Quando si eliminano eccezioni!

IntSet con insert che non ignora i duplicati, ma lancia un'eccezione:

```
public class ExIntSet {  
    //@ensures !\old(isIn(x)) && isIn(x)&&...  
    //@ signals (DuplicateException e) \old(isIn(x));  
    public void insert (int x) throws DuplicateException  
        ...  
}
```

Una sottoclasse che elimina eccezione:

```
public class IntSet extends ExIntSet {...  
    // NON LANCIARE ECCEZIONE  
    public void insert (int x)  
        ...  
}
```

- Regola della segnatura: OK, ma la regola dei metodi: KO!
- Utilizzatore di IntSet potrebbe usare l'eccezione Duplicate anche per stabilire se l'elemento era già presente e sarebbe sorpreso
- Quindi si può eliminare un'eccezione dalla segnatura solo se non è effettivamente usata o se il suo lancio è opzionale
  - Ad esempio se era prevista non per trattare violazioni di precondizioni sui dati, ma per il verificarsi di altre condizioni che non possono più verificarsi
  - Notare la differenza con le precondizioni, che possono invece essere indebolite

# Ancora su eccezioni

- Nell'es. BoundedStack non può essere definita come erede di Stack
- La Stack può essere definita come erede di BoundedStack?
- Stack.push
  - `//@ ensures (* inserisce v in cima *)`;
- BoundedStack.push
  - `//@ requires this.size() <=100`;
  - `//@ ensures (* inserisce v in cima *)`;
- Sì, la regola dei metodi è soddisfatta
- Se però contratto di BoundedStack.push fosse:
  - `//@ signals (OverflowException e) this.size() >100`;
  - `//@ ensures \old(size())<100 ==> (* inserisce v in cima *)`;
- Allora sarebbe violata la regola della postcondizione

# Regola delle proprietà

- “Sottotipo deve conservare le proprietà del supertipo”
  - Sono proprietà generali incluse nel contratto della classe, e deducibili dal contratto dei metodi
  - Non compaiono però esplicitamente nei metodi
  - Proprietà definite tipicamente nella OVERVIEW o come public invariant del supertipo
- Occorre mostrare che tutti i metodi nuovi o ridefiniti, inclusi i costruttori, del sottotipo conservano le proprietà invarianti e le proprietà evolutive del supertipo, osservabili con i metodi pubblici observer della sopraclasse
  - Si usa solo specifica dei metodi, non implementazione
  - Fra un’osservazione e l’altra, è possibile che vengano chiamati anche i metodi nuovi dell’estensione
- Proprietà invariante: proprietà degli stati astratti osservabili dalla sopraclasse
  - Esempio: `size()` di un `IntSet` è sempre  $\geq 0$
- Proprietà evolutiva: relazione fra uno stato astratto osservabile e lo stato astratto osservabile successivo
  - Esempio di mutabilità; il grado di un `Poly` non cambia

# Violazione proprietà invariante

- Tipo FatSet: come IntSet ma non è mai vuoto

```
//@ public invariant this.size() >= 1; // Sempre almeno un elemento
//@ ensures (*costruisce insieme {i}*)
public FatSet(int i)
//@ ensures \old(this.size())>=2) ==> !this.isIn(x); // Elimina x da this
public void removeNonEmpty(int x)
```

```
public class IntSet {
    //@ensures !this.isIn(x);
    public void remove (int x) {...}
}
```

- IntSet non può essere definita come sottoclasse di FatSet perché ne viola la proprietà  
—...anche solo aggiungendo costruttore IntSet() per l'insieme vuoto

# Violazione proprietà evolutiva

- Tipo SimpleSet come IntSet ma senza remove
- IntSet non può essere definito come estensione pura di SimpleSet aggiungendo remove, perché ne viola proprietà
  - Utilizzatori di SimpleSet sarebbero sorpresi non ritrovando più gli elementi che avevano inserito in precedenza in un SimpleSet

# Altro esempio

- Sottoclasse di Stack che aggiunge un metodo mirror()

```
public class MirrorStack extends Stack {  
    //@ ensures (*this diventa il riflesso di \old(this) *);  
    public void mirror() {...}  
}
```

- Pur essendo un'estensione pura di Stack, viola la proprietà evolutiva delle Pile (ossia la politica LIFO)
- Codice utilizzatore sarebbe sorpreso di non trovare i dati in ordine LIFO
  - Quindi anche un'estensione pura può violare il principio di sostituzione

# Ereditarietà e collezioni

```
class Automobile {...}  
class AutomobileElettrica extends Automobile {...}  
class ParcheggioAutomobili {  
    // @ ensures (*inserisce auto a in parcheggio*)  
    public void aggiungi(Automobile a) {  
        ....  
    }  
}
```

- La classe ParcheggioElettriche rappresenta parcheggi speciali dedicati esclusivamente alle AutomobiliElettriche
  - ParcheggioElettriche può essere sottotipo di ParcheggioAutomobili?

# ... quindi

```
Automobile a=new Automobile();  
ParcheggioAutomobili p=new ParcheggioElettriche(100);  
p.aggiungi(a);
```

- Inserisce Automobile non elettrica in ParcheggioElettriche (violata regola delle proprietà)
  - Se per impedirlo si ridefinisse ParcheggioElettriche.aggiungi(Automobile) in modo da inserire solo Auto elettriche, si violerebbe invece la postcondizione di Parcheggio.aggiungi(Automobile)
- Un ParcheggioElettriche non può essere definito come sottotipo di ParcheggioAutomobili
  - Questo perché ParcheggioAutomobili è un tipo mutabile
  - Se fosse immutabile, non ci sarebbe metodo aggiungi(): un ParcheggioElettriche potrebbe essere definito come estensione di ParcheggioAutomobili (il costruttore non è ereditato...)
- E' lo stesso problema già visto con i generici: Collection<String> non è sottotipo di Collection<Object>



# Riassumendo: Principio di Sostituzione

- Regola della segnatura: un sottotipo deve avere tutti i metodi del supertipo, e signature dei metodi del sottotipo devono essere compatibili
  - Garantito dal compilatore Java
- Regola dei metodi: le chiamate ai metodi del sottotipo devono comportarsi come le chiamate ai metodi corrispondenti del supertipo
  - Garantito da JML ma non da Java
- Regola delle proprietà: sottotipo deve preservare tutti i public invariant e le proprietà evolutive degli oggetti del supertipo
  - Garantito solo in parte da JML

# Consigli per usare ereditarietà

- Un approccio **errato**
  - Ereditarietà eguale a riuso diretto del codice
  - Esempio: classe Persona eredita da classe Data perché ha una data di nascita
- Approccio **corretto**: principio di sostituzione
  - Un oggetto della classe derivata deve potere essere sostituito ovunque ci sia un oggetto della classe base
  - Persona non può essere usato dove c'è una Data!
  - Ereditarietà quindi non è un meccanismo di condivisione/riuso del codice: la sottoclasse deve **estendere la semantica** della superclasse