Metodologie di Programmazione

- tecniche per la programmazione orientata ad oggetti (in piccolo)
- esemplificate utilizzando il linguaggio Java
- testo di riferimento (fino al Cap. 10):
  - Barbara Liskov, Program Development in Java, Abstraction, Specification and Object-Oriented Design, Addison-Wesley 2001

1

Contenuti della parte introduttiva del corso

- implementazione di linguaggi ad alto livello
  - interpretazione, compilazione, implementazioni miste
- programmazione come decomposizione guidata da astrazioni
  - meccanismi di astrazione: parametrizzazione, specifica
  - tipi di astrazione: procedure, tipi di dato astratti, iterazione astratta, gerarchie di tipi
- cenni di semantica operazionale di Java
  - classi, oggetti, metodi, gerarchie
  - il modello di esecuzione

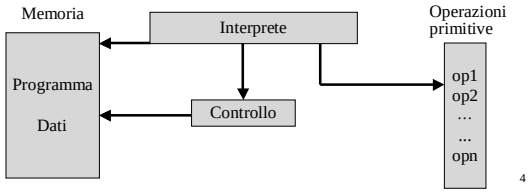
2

Macchine astratte, linguaggi, interpretazione, compilazione

3

Macchine astratta

- una collezione di strutture dati ed algoritmi in grado di memorizzare ed eseguire programmi
- componenti della macchina astratta
  - interprete
  - memoria (dati e programmi)
  - controllo
  - operazioni "primitive"



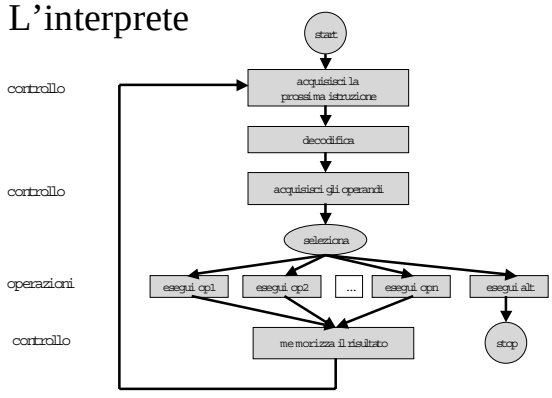
4

Il componente di controllo

- una collezione di strutture dati ed algoritmi per
  - acquisire la prossima istruzione
  - gestire le chiamate ed i ritorni dai sottoprogrammi
  - acquisire gli operandi e memorizzare i risultati delle operazioni
  - mantenere le associazioni fra nomi e valori denotati
  - gestire dinamicamente la memoria
  - …...

5

L'interprete



6

Il linguaggio macchina

- **M** macchina astratta
- L_M linguaggio macchina di **M**
  - è il linguaggio che ha come stringhe legali tutti i programmi interpretabili dall'interprete di **M**
- i programmi sono particolari dati su cui opera l'interprete

7

Macchine astratte: implementazione

- **M** macchina astratta
- i componenti di **M** sono realizzati mediante strutture dati ed algoritmi implementati nel linguaggio macchina di una **macchina ospite** M_O, già esistente (implementata)
- è importante la realizzazione dell'interprete di **M**
  - può coincidere con l'interprete di M_O
    - **M** è realizzata come **estensione** di M_O
    - altri componenti della macchina possono essere diversi
  - può essere diverso dall'interprete di M_O
    - **M** è realizzata su M_O in modo **interpretativo**
    - altri componenti della macchina possono essere uguali

8

Dal linguaggio alla macchina astratta

- **M** macchina astratta — L_M linguaggio macchina di **M**
- **L** linguaggio — M_L macchina astratta di **L**
- implementazione di **L** =
  realizzazione di M_L su una macchina ospite M_O
- se **L** è un linguaggio ad alto livello ed M_O è una macchina "fisica"
  - l'interprete di M_L è necessariamente diverso dall'interprete di M_O
    - M_L è realizzata su M_O in modo interpretativo
    - l'implementazione di **L** si chiama **interprete**
    - esiste una soluzione alternativa basata su tecniche di traduzione (**compilatore**?)

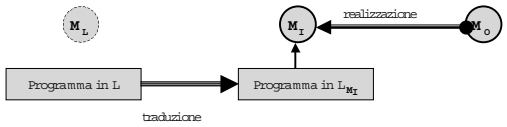
9

Implementare un linguaggio

- **L** linguaggio ad alto livello
- M_L macchina astratta di **L**
- M_O macchina ospite
- implementazione di **L** 1: **interprete** (puro)
  - M_L è realizzata su M_O in modo interpretativo
  - scarsa efficienza, soprattutto per colpa dell'interprete (ciclo di decodifica)
- implementazione di **L** 2: **compilatore** (puro)
  - i programmi di **L** sono tradotti in programmi funzionalmente equivalenti nel linguaggio macchina di M_O
  - i programmi tradotti sono eseguiti direttamente su M_O
    - M_L non viene realizzata
  - il problema è quello della dimensione del codice prodotto
- due casi limite che nella realtà non esistono quasi mai

10

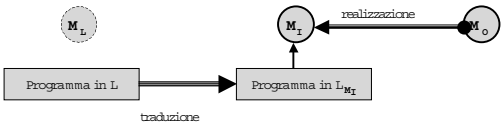
La macchina intermedia



- **L** linguaggio ad alto livello
- M_L macchina astratta di **L**
- M_I macchina intermedia
- L_{M_I} linguaggio intermedio
- M_O macchina ospite
- traduzione dei programmi da **L** al linguaggio intermedio L_{M_I} + realizzazione della macchina intermedia M_I su M_O

11

Interpretazione e traduzione pura



- M_L = M_I interpretazione pura
- M_O = M_I traduzione pura
  - possibile solo se la differenza fra M_O e M_L è molto limitata
    - L linguaggio assembler di M_O
  - in tutti gli altri casi, c'è sempre una macchina intermedia che estende eventualmente la macchina ospite in alcuni componenti

12

Il compilatore

- quando l'interprete della macchina intermedia M_I coincide con quello della macchina ospite M_O
- che differenza c'è tra M_I e M_O?
  - il **supporto a tempo di esecuzione (rts)**
    - collezione di strutture dati e sottoprogrammi che devono essere caricati su M_O (estensione) per permettere l'esecuzione del codice prodotto dal traduttore (compilatore)
  - $M_I = M_O$ + rts
- il linguaggio L_{M_I} è il linguaggio macchina di M_O esteso con chiamate al supporto a tempo di esecuzione

A che serve il supporto a tempo di esecuzione?

- un esempio da un linguaggio antico (FORTRAN)
  - praticamente una notazione "ad alto livello" per un linguaggio macchina
- in linea di principio, è possibile tradurre completamente un programma FORTRAN in un linguaggio macchina puro, senza chiamate al rts, ma ...
  - la traduzione di alcune primitive FORTRAN (per esempio, relative all'I/O) produrrebbe centinaia di istruzioni in linguaggio macchina
    - se le inserissimo nel codice compilato, la sua dimensione crescerebbe a dismisura
    - in alternativa, possiamo inserire nel codice una chiamata ad una routine (indipendente dal particolare programma)
    - tale routine deve essere caricata su M_O ed entra a far parte del rts
- nei veri linguaggi ad alto livello, questa situazione si presenta per quasi tutti i costrutti del linguaggio
  - meccanismi di controllo
  - non solo routines ma anche strutture dati

Il caso del compilatore C

- il supporto a tempo di esecuzione contiene
  - varie strutture dati
    - la pila dei records di attivazione
      - ambiente, memoria, sottoprogrammi, …
    - la memoria a heap
      - puntatori, ...
  - i sottoprogrammi che realizzano le operazioni necessarie su tali strutture dati
- il codice prodotto è scritto in linguaggio macchina esteso con chiamate al rts

Implementazioni miste

- quando l'interprete della macchina intermedia M_I non coincide con quello della macchina ospite M_O
- esiste un ciclo di interpretazione del linguaggio intermedio L_{M_I} realizzato su M_O
  - per ottenere un codice tradotto più compatto
  - per facilitare la portabilità su diverse macchine ospiti
    - si deve riimplementare l'interprete del linguaggio intermedio
    - non è necessario riimplementare il traduttore

Compilatore o implementazione mista?

- nel compilatore non c'è di mezzo un livello di interpretazione del linguaggio intermedio
  - sorgente di inefficienza
    - la decodifica di una istruzione nel linguaggio intermedio (e la sua trasformazione nelle azioni semantiche corrispondenti) viene effettuata ogni volta che si incontra l'istruzione
- se il linguaggio intermedio è progettato bene, il codice prodotto da una implementazione mista ha dimensioni inferiori a quelle del codice prodotto da un compilatore
- un'implementazione mista è più portabile di un compilatore
- il supporto a tempo di esecuzione di un compilatore si ritrova quasi uguale nelle strutture dati e routines utilizzate dall'interprete del linguaggio intermedio

L'implementazione di Java

- è un'implementazione mista
  - traduzione dei programmi da Java a byte-code, linguaggio macchina di una macchina intermedia chiamata Java Virtual Machine
  - i programmi byte-code sono interpretati
  - l'interprete della Java Virtual Machine opera su strutture dati (stack, heap) simili a quelle del rts del compilatore C
    - la differenza fondamentale è la presenza di una gestione automatica del recupero della memoria a heap (garbage collector)
  - su una tipica macchina ospite, è più semplice realizzare l'interprete di byte-code che l'interprete di Java
    - byte-code è più "vicino" al tipico linguaggio macchina

Tre famiglie di implementazioni

- interprete puro
  - $M_L = M_I$
  - interprete di **L** realizzato su M_O
  - alcune implementazioni (vecchie!) di linguaggi logici e funzionali
    - LISP, PROLOG
- compilatore
  - macchina intermedia M_I realizzata per estensione sulla macchina ospite M_O (rts, nessun interprete)
    - C, C++, PASCAL
- implementazione mista
  - traduzione dei programmi da **L** a L_{M_I}
  - i programmi L_{M_I} sono interpretati su M_O
    - Java
    - i "compilatori" per linguaggi funzionali e logici (LISP, PROLOG, ML)
    - alcune (vecchie!) implementazioni di Pascal (Pcode)

19

Implementazioni miste e interpreti puri

- la traduzione genera codice in un linguaggio più facile da interpretare su una tipica macchina ospite
- ma soprattutto può effettuare una volta per tutte (a tempo di traduzione, staticamente) analisi, verifiche e ottimizzazioni che migliorano
  - l'affidabilità dei programmi
  - l'efficienza dell'esecuzione
- varie proprietà interessate
  - inferenza e controllo dei tipi
  - controllo sull'uso dei nomi e loro risoluzione "statica"
  - ….

20

Analisi statica

- dipende dalla semantica del linguaggio
- certi linguaggi (LISP) non permettono praticamente nessun tipo di analisi statica
  - a causa della regola di scoping dinamico nella gestione dell'ambiente non locale
- altri linguaggi funzionali più moderni (ML) permettono di inferire e verificare molte proprietà (tipi, nomi, …) durante la traduzione, permettendo di
  - localizzare errori
  - eliminare controlli a tempo di esecuzione
    - type-checking dinamico nelle operazioni
  - semplificare certe operazioni a tempo di esecuzione
    - come trovare il valore denotato da un nome

21

Analisi statica in Java

- Java è fortemente tipato
  - il type checking può essere in gran parte effettuato dal traduttore e sparire quindi dal byte-code generato
- le relazioni di subtyping permettono che una entità abbia un tipo vero (actual type) diverso da quello apparente (apparent type)
  - tipo apparente noto a tempo di traduzione
  - tipo vero noto solo a tempo di esecuzione
  - è garantito che il tipo apparente sia un supertype di quello vero
- di conseguenza, alcune questioni legate ai tipi possono solo essere risolte a tempo di esecuzione
  - scelta del più specifico fra diversi metodi overloaded
  - casting (tentativo di forzare il tipo apparente ad un suo possibile sottotipo)
  - dispatching dei metodi (scelta del metodo secondo il tipo vero)
- controlli e simulazioni a tempo di esecuzione

22

Programmazione = decomposizione basata su astrazioni

23

Decomposizione in "moduli"

- necessaria quando si devono sviluppare programmi abbastanza grandi
  - decomporre il problema in sotto-problemi
  - i moduli che risolvono i sotto-problemi devono riuscire a cooperare nella soluzione del problema originale
- persone diverse possono/devono essere coinvolte
  - si deve poter lavorare in modo indipendente (ma coerente) nello sviluppo dei diversi moduli
  - deve essere possibile eseguire "facilmente" (da parte di persone diverse da quelle coinvolte nello sviluppo) modifiche e aggiornamenti (manutenzione)
    - a livello dei singoli moduli, senza influenzare il comportamento degli altri
- i programmi devono essere decomposti in moduli, in modo che sia facile capirne le interazioni

24

Decomposizione e astrazione

- caratteristiche
  - i sotto-problemi devono avere lo stesso livello di dettaglio
  - ogni sotto-problema può essere risolto in modo indipendente
  - una combinazione delle soluzioni ai sotto-problemi risolve il problema originale
- la decomposizione può essere effettuata in modo produttivo ricorrendo all'**astrazione**
  - cambiamento del livello di dettaglio, nella descrizione di un problema, limitandosi a "considerare" solo alcune delle sue caratteristiche
  - si passa ad un problema più semplice
    - su questo si effettua la decomposizione in sotto-problemi
- il passo astrazione-decomposizione si può ripetere più volte finché non si arriva a sottoproblemi per cui si conosce una soluzione

25

Astrazione

- processo con cui ci si dimentica di una parte dell'informazione
  - effetto
    - cose che sono diverse diventano uguali
  - perché?
    - perché si spera di semplificare l'analisi, separando gli attributi che si ritengono rilevanti da quelli che si ritiene possano essere trascurati
    - la rilevanza dipende dal contesto
- a noi interessano i meccanismi di astrazione legati alla programmazione
- lo strumento fondamentale è l'utilizzazione di **linguaggi ad alto livello**
  - enorme semplificazione per il programmatore
    - usando direttamente i costrutti del linguaggio ad alto livello
    - invece che una delle numerosissime sequenze di istruzioni in linguaggio macchina "equivalenti"

26

I linguaggi non bastano

```
//ricerca all'insù
   found = false;
   for (int i = 0; i < a.length; i++)
        if (a[i] == e) {
              z = i; found = true;}
//ricerca all'ingiù
   found = false;
   for (int i = a.length - 1; i >= 0; i--)
        if (a[i] == e) {
              z = i; found = true;}
```

- sono diversi
  - possono dare risultati diversi
  - potrebbero essere stati scritti con l'idea di risolvere lo stesso problema
    - verificare se l'elemento è presente nell'array e restituire una posizione in cui è contenuto

27

Migliori astrazioni nel linguaggio?

- il linguaggio potrebbe avere delle potenti operazioni sull'array del tipo isIn e indexOf

```
//ricerca indipendente dall'ordine
   found = a.isIn(e);
   if found z = a.indexOf(e);
```

- l'astrazione è scelta dal progettista del linguaggio
  - quali e quante?
  - quanto complicato diventa il linguaggio?
- meglio progettare linguaggi dotati di **meccanismi che permettano di definire le astrazioni che servono**

28

Il più comune tipo di astrazione

- l'astrazione procedurale
  - presente in tutti i linguaggi di programmazione
- la separazione tra "definizione" e "chiamata" rende disponibili nel linguaggio i due meccanismi fondamentali di astrazione
  - **l'astrazione attraverso parametrizzazione**
    - si astrae dall'identità di alcuni dati, rimpiazzandoli con parametri
    - si generalizza un modulo per poterlo usare in situazioni diverse
  - **l'astrazione attraverso specifica**
    - si astrae dai dettagli dell'implementazione del modulo, per limitarsi a considerare il comportamento che interessa a chi utilizza il modulo (ciò che fa, non come lo fa)
    - si rende ogni modulo indipendente dalle implementazioni dei moduli che usa

29

Astrazione via parametrizzazione

- l'introduzione dei parametri permette di descrivere un insieme (anche infinito) di computazioni diverse con un singolo programma che le astrae tutte

`x * x + y * y`
- descrive una computazione

λx,y:int. (x * x + y * y)
- descrive tutte le computazioni che si possono ottenere chiamando la procedura, cioè applicando la funzione ad una opportuna n-upla di valori

λx,y:int. (x * x + y * y) (w,z)
- ha la stessa semantica dell'espressione `w * w + z * z`

30

Astrazione via specifica

- la procedura si presta a meccanismi di astrazione più potenti della parametrizzazione
- possiamo astrarre dalla specifica computazione descritta nel corpo della procedura, associando ad ogni procedura una specifica
  - semantica intesa della procedura
- e derivando la semantica della chiamata dalla specifica invece che dal corpo della procedura
- non è di solito supportata dal linguaggio di programmazione
  - se non in parte (vedi specifiche di tipo)
- si realizza con specifiche semi-formali
  - sintatticamente, commenti

31

Un esempio

```
float sqrt (float coef) {
  // REQUIRES: coef > 0
  // EFFECTS: ritorna una approssimazione
  // della radice quadrata di coef
  float ans = coef / 2.0; int i = 1;
  while (i < 7) {
      ans = ans-((ans*ans-coef)/(2.0*ans));
      i = i+1; }
  return ans; }
```

- precondizione (asserzione requires)
  - deve essere verificata quando si chiama la procedura
- postcondizione (asserzione effects)
  - tutto ciò che possiamo assumere valere quando la chiamata di procedura termina, se al momento della chiamata era verificata la precondizione

32

Il punto di vista di chi usa la procedura

```
float sqrt (float coef) {
  // REQUIRES: coef > 0
  // EFFECTS: ritorna una approssimazione
  // della radice quadrata di coef
  ... }
```

- gli utenti della procedura non si devono preoccupare di capire cosa la procedura fa, astraendo le computazioni descritte dal corpo
  - cosa che può essere molto complessa
- gli utenti della procedura non possono osservare le computazioni descritte dal corpo e dedurre da questo proprietà diverse da quelle specificate dalle asserzioni
  - astraendo dal corpo (implementazione), si "dimentica" informazione evidentemente considerata non rilevante

33

Tipi di astrazione

- parametrizzazione e specifica permettono di definire vari tipi di astrazione
  - astrazione procedurale
    - si aggiungono nuove operazioni a quelle della macchina astratta del linguaggio di programmazione
  - astrazione di dati
    - si aggiungono nuovi tipi di dato a quelli della macchina astratta del linguaggio di programmazione
  - iterazione astratta
    - permette di iterare su elementi di una collezione, senza sapere come questi vengono ottenuti
  - gerarchie di tipo
    - permette di astrarre da specifici tipi di dato a famiglie di tipi correlati

34

Astrazione procedurale

- fornita da tutti i linguaggi ad alto livello
- aggiunge nuove operazioni a quelle della macchina astratta del linguaggio di programmazione
  - per esempio, `sqrt` sui `float`
- la specifica descrive le proprietà della nuova operazione

35

Astrazione sui dati

- fornita da tutti i linguaggi ad alto livello moderni
- aggiunge nuovi tipi di dato e relative operazioni a quelli della macchina astratta del linguaggio
  - tipo `MultiInsieme` con le operazioni `vuoto`, `inserisci`, `rimuovi`, `numeroDi` e `dimensione`
  - la rappresentazione dei valori di tipo `MultiInsieme` e le operazioni sono realizzate nel linguaggio
  - l'utente non deve interessarsi dell'implementazione, ma fare solo riferimento alle proprietà presenti nella specifica
  - le operazioni sono astrazioni definite da asserzioni come

    `dimensione(inserisci(s,e)) = dimensione(s)+1`

    `numeroDi(vuoto(),e) = 0`
- la specifica descrive le relazioni fra le varie operazioni
  - per questo, è cosa diversa da un insieme di astrazioni procedurali

36

Iterazione astratta

- non è fornita da nessun linguaggio di uso comune
  - può essere simulata (per esempio, in Java)
- permette di iterare su elementi di una collezione, senza sapere come questi vengono ottenuti
- evita di dire cose troppo dettagliate sul flusso di controllo all'interno di un ciclo
  - per esempio, potremmo iterare su tutti gli elementi di un `MultiInsieme` senza imporre nessun vincolo sull'ordine con cui vengono elaborati
- astrae (nasconde) il flusso di controllo nei cicli

Gerarchie di tipo

- fornite da alcuni linguaggi ad alto livello moderni
  - per esempio, Java
- permettono di astrarre gruppi di astrazioni di dati (tipi) a famiglie di tipi
- i tipi di una famiglia condividono alcune operazioni
  - definite nel supertype, di cui tutti i tipi della famiglia sono subtypes
- una famiglia di tipi astrae i dettagli che rendono diversi tra loro i vari tipi della famiglia
- in molti casi, il programmatore può ignorare le differenze

Astrazione e programmazione orientata ad oggetti

- il tipo di astrazione più importante per guidare la decomposizione è l'astrazione sui dati
  - gli iteratori astratti e le gerarchie di tipo sono comunque basati su tipi di dati astratti
- l'astrazione sui dati è il meccanismo fondamentale della **programmazione orientata ad oggetti**
  - anche se esistono altre tecniche per realizzare tipi di dato astratti
    - per esempio, all'interno del paradigma di programmazione funzionale

Semantica operazionale

- modello di esecuzione
  - importanti soprattutto le strutture che compongono lo stato
- simile alle strutture a run-time della JVM, che esegue il byte-code prodotto dal compilatore
  - con alcune semplificazioni legate alle ottimizzazioni effettuate dal compilatore

Cosa aggiungiamo

- le gerarchie di classi
  - trattiamo l'ereditarietà ed l'overriding
- l'attributo "static" per variabili e metodi
  - esistono variabili e metodi propri della classe
- i costruttori
  - metodi che vengono invocati al momento della creazione di una istanza di classe

Semantica statica

- nella formalizzazione trascuriamo tutti gli aspetti legati alla semantica statica
  - in particolare, quelli che darebbero origine a messaggi di errore durante la compilazione
  - tipi, visibilità dei nomi, vincoli sull'overriding
- le proprietà statiche importanti verranno descritte in modo informale
- semantica semplificata solo per programmi che supererebbero con successo l'analisi statica

Lo stato

- pila di attivazioni σ
  - per la valutazione dei metodi
  - simile alla pila dei record di attivazione nei linguaggi tradizionali
- heap ζ
  - contiene gli oggetti (istanze di classi)
- ambiente delle classi ρ
  - contiene le classi dichiarate prima dell'inizio dell'esecuzione

43

Ambiente delle classi ρ

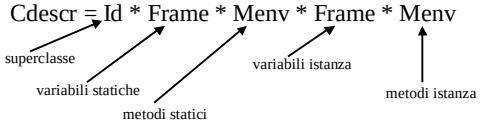
- ρ è una funzione da identificatori di classe a descrizioni di classe
  - ρ : Cenv
  - Cenv = Id -> Cdescr
- cos'è una descrizione di classe?
- vediamo prima la sintassi (semplificata) che usiamo per le dichiarazioni di classe

44

Dichiarazione di classe: sintassi

```
Class_decl := class Id extends Id {
   Static_var_decl_list
   Static_meth_decl_list
   Inst_var_decl_list
   Inst_meth_decl_list
   Costruttore  }
```

Cdescr = Id * Frame * Menv * Frame * Menv



il costruttore (sequenza di assegnamenti) viene aggiunto ai metodi istanza

45

Il frame

- è una tabella (estendibile e mutabile) che mantiene associazioni fra
  - identificatori (di variabili)
  - valori
    - interi, booleani
    - locazioni (puntatori ad oggetti)
- un frame φ:Frame viene creato vuoto (newframe())
- l'operazione bind(φ, i, v) estende φ inserendo l'associazione tra i e v
- l'operazione update(φ, i, v) modifica in φ l'associazione per i (che deve esistere)
- l'operazione copy(φ) costruisce una copia di φ
- l'operazione defined(φ, i) dice se φ contiene un'associazione per i
- per ottenere il valore di una variabile, si applica il frame all'Id

46

Ambiente di metodi μ

- μ è una funzione da identificatori di metodo a descrizioni di metodo
  - μ : Menv
  - Menv = Id -> Mdescr
- cos'è una descrizione di metodo?
- vediamo prima la sintassi (semplificata) che usiamo per le dichiarazioni di metodo

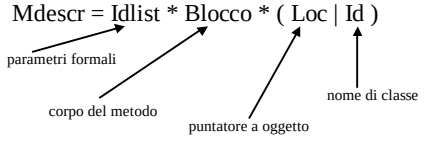
47

Dichiarazione di metodo: sintassi

```
Method_decl := Id (Idlist) Blocco
```

- ignoriamo i tipi
- quando il metodo verrà invocato, dovrà sapere la classe o l'oggetto a cui appartiene

Mdescr = Idlist * Blocco * (Loc | Id)



48

Operazioni sugli ambienti di metodi e di class

- cemptyenv() costruisce un ambiente di classi "vuoto"
- cbind((ρ:Cenv), (i:Id), (c:Cdescr)) estende ρ associando ad i il valore c
- cdefined(ρ, i) dice se ρ è definita per i
- memptyenv() costruisce un ambiente di metodi "vuoto"
- mbind((μ:Menv), (i:Id), (m:Mdescr)) estende μ associando ad i il valore m
- mdefined(μ, i) dice se μ è definita per i
- instantiate((μ:Menv), (l: Loc)) crea un nuovo ambiente μ1 diverso da μ perché tutte le descrizioni di metodi contengono l'oggetto l

49

La heap ζ

- ζ è una funzione da locazioni a descrizioni di istanza (oggetto)
  - ζ : Heap
  - Heap = Loc -> Odescr
- cos'è una descrizione di oggetto?

Odescr = Id * Frame * Menv

classe — variabili di istanza — metodi di istanza

50

Operazioni sulla heap

- newheap() genera una heap vuota
- newloc(ζ) genera una nuova locazione in ζ
- hbind((ζ:Heap), (l:Loc), (o:Odescr)) estende ζ associando ad l il valore o
- un oggetto viene creato con l'espressione new Id
  - genera il valore o:Odescr a partire dalla classe Id
  - l = newloc(ζ)
  - hbind((ζ:Heap), (l:Loc), (o:Odescr))
  - restituisce l

51

La pila di attivazioni σ

- σ è una pila di records di attivazione di metodi
  - σ : Astack
  - Astack = Stack (Record)
- il record di attivazione
  - oggetto o classe a cui il metodo appartiene
  - pila di frames (blocchi annidati)

Record = (Id | Loc) * Stack(Frame)

classe — oggetto — variabili locali

52

Operazioni sulle pile (record, frame)

- emptystack() genera una pila vuota
- top((π:Stack(x)) restituisce l'elemento di tipo x in testa a π
- pop((π:Stack(x)) modifica π eliminando l'elemento in testa
- push((π:Stack(x),(e:x)) modifica π inserendo l'elemento e in testa
- empty((π:Stack(x)) verifica se π è vuota

53

Le strutture dello stato (riepilogo 1)

- Ambiente delle classi

Cenv = Id -> Cdescr

Cdescr = Id * Frame * Menv * Frame* Menv

- cemptyenv()
- cbind((ρ:Cenv), (i:Id), (c:Cdescr))
- cdefined((ρ:Cenv),(i:Id))

- Heap

Heap = Loc -> Odescr

Odescr = Id * Frame * Menv

- newheap()
- newloc ((ζ:Heap))
- hbind((ζ:Heap), (l:Loc), (o:Odescr))

54

Le strutture dello stato (riepilogo 2)

- Pila delle attivazioni

Astack = Stack (Record)

Record = (Id | Loc) * Stack(Frame)

- operazioni delle pile

Le strutture (ausiliarie) dello stato (riepilogo 3)

- Ambiente dei metodi

Menv = Id -> Mdescr

Mdescr = Idlist * Blocco * (Loc | Id)

- memptyenv()
- mbind((μ:Menv), (i:Id), (m:Mdescr))
- mdefined ((μ:Menv), (i:Id))
- instantiate((μ:Menv), (l: Loc))

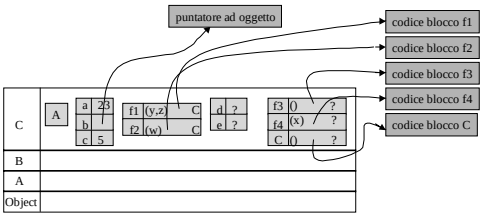
- Frames

Frame = Id -> Val

Val = (Bool | Int | Loc)

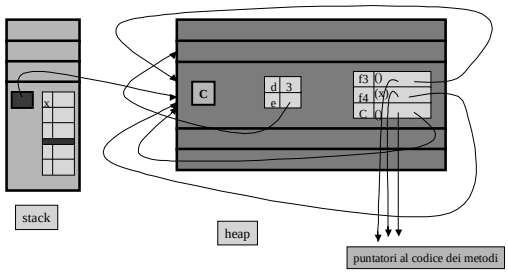
- newframe()
- copy((φ:Frame))
- bind((φ:Frame) , (i:Id), (v:Val))
- update((φ:Frame) , (i:Id), (v:Val))
- defined((φ:Frame), (i:Id))

Uno stato (1)



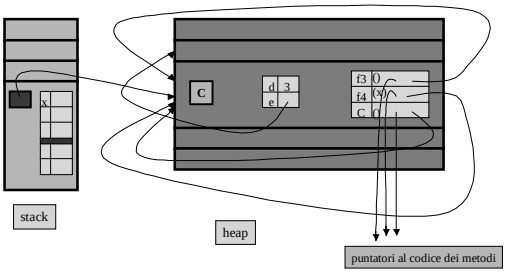
- parte statica (classi)

Uno stato (2)



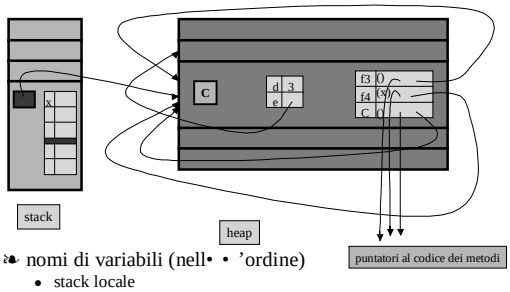
- parte dinamica (pila e heap)
  - è in esecuzione il metodo f4 di una istanza di C

Cosa si “vede” in questo stato



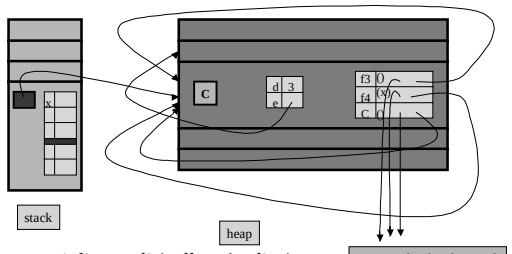
- tutti i nomi delle classi
  - attraverso queste, le variabili ed i metodi statici

Cosa si “vede” in questo stato



- nomi di variabili (nell• • ’ordine)
  - stack locale
  - frame dell’oggetto a cui appartiene il metodo
  - frames (statici) lungo la catena di sottoclassi

Cosa si “vede” in questo stato



- nomi di metodi (nell• • ’ordine)
  - ambiente di metodi dell’oggetto a cui appartiene il metodo
    - lui incluso (ricorsione)
  - ambienti di metodi (statici) lungo la catena di sottoclassi

Semantica Operazionale di un frammento di Java: le regole di transizione

estensione (con piccole varianti) di quella in

Barbuti, Mancarella, Turini,
Elementi di Semantica Operazionale,
appunti di Fondamenti di Programmazione

Regole di transizione

- regole che ci dicono come viene modificato uno stato per effetto dell’esecuzione di un certo costrutto del linguaggio
- diverse relazioni di transizione, in corrispondenza dei diversi tipi di costrutti del linguaggio
  - espressioni, comandi, dichiarazioni di variabile, dichiarazioni di metodi, dichiarazioni di classe, ecc.

Dichiarazione di classe

```
Class_decl := class Id extends Id {
   Static_var_decl_list
   Static_meth_decl_list
   Inst_var_decl_list
   Inst_meth_decl_list
   Costruttore  }
```

- assunzioni
  - c’è sempre extends
    - ogni classe estende la classe Object
  - le dichiarazioni di variabili e metodi statici sono “raggruppate”
  - esiste sempre il costruttore

Semantica informale delle classi

- variabili e metodi statici appartengono alla classe
- variabili e metodi di istanza appartengono agli oggetti (istanze della classe)
- una classe definisce un tipo (nome della classe)
  - gli oggetti istanza hanno quel tipo
- relazioni di sottoclasse ed ereditarietà
  - gerarchia di tipi, realizzata con extends
  - la radice della gerarchia è la classe predefinita Object

Ereditarietà

- se c1 è una sottoclasse (estende) c2
  - variabili e metodi statici di c2 (e delle sue superclassi) sono visibili direttamente da c1
  - variabili e metodi di istanza di c2 (e delle sue superclassi) diventano anche variabili e metodi di istanza di c1
- a meno di overriding

Overriding

- se c1 è una sottoclasse (estende) c2
  - un metodo di istanza di c2 (e delle sue superclassi) può essere ridefinito (stesso nome e tipi) in c1
  - l'idea è quella di definire un metodo più specializzato, che rimpiazza quello definito in una delle superclassi
- l'overriding non "dovrebbe" essere possibile per le variabili e per i metodi statici
  - ma questo non è sempre garantito dal compilatore
  - comportamenti complessi e poco prevedibili
  - assumeremo che non si verifichi mai

Inizializzazioni

- Java permette di inizializzare (al valore di una espressione) sia le variabili statiche che quelle di istanza in una dichiarazione
  - le inizializzazioni delle variabili statiche vengono effettuate la prima volta che si usa una classe
  - le inizializzazioni delle variabili di istanza vengono effettuate all'atto della creazione di una istanza
  - in tutti e due i casi, l'inizializzazione può riguardare anche le superclassi
- per semplicità, decidiamo di non avere inizializzazioni nelle dichiarazioni di variabile all'interno di classi
- per questo ci aspettiamo che esista sempre un costruttore
  - eseguito quando si crea un'istanza

Il costruttore

- i costruttori sono usati per la vera inizializzazione delle variabili di istanza
- anche per i costruttori esiste un meccanismo di ereditarietà
- se c è una classe che ha come superclassi (nell'ordine) le classi c1, c2, …, cn=Object,
  - all'atto della creazione di una istanza di c
  - si eseguono (nell'ordine) i costruttori di cn, …, c2, c1, c

Dichiarazione di classe

```
Class_decl := class Id extends Id {
   Static_var_decl_list
   Static_meth_decl_list
   Inst_var_decl_list
   Inst_meth_decl_list
   Costruttore  }
```

Cenv = Id -> Cdescr

Cdescr = Id * Frame * Menv * Frame* Menv

- la relazione di transizione

Class_decl * Cenv $\rightarrow_{cdecl}$ Cenv

Dichiarazione di classe

Cenv = Id -> Cdescr

Cdescr = Id * Frame * Menv * Frame* Menv

Class_decl * Cenv $\rightarrow_{cdecl}$ Cenv

$\rho(b) = (\_,\_,\_, \varphi, \mu)$ $\mu(b) = (\_, c1, \_)$

$<svdl, newframe()> \rightarrow_{vdecl} \varphi_1$ $<ivdl, copy(\varphi)> \rightarrow_{vdecl} \varphi_2$

$<smdl, a, memptyenv()> \rightarrow_{mdecl} \mu_1$ $<imdl, a, \mu> \rightarrow_{mdecl} \mu_2$

$<$*class* a *extends* b {svdl smdl ivdl imdl c}, $\rho> \rightarrow_{cdecl}$
cbind(ρ, a, (b, φ_1, μ_1 , φ_2, mbind(μ_2, a, ([], c1 c, a))))

Commenti

$\rho(b) = (\_,\_,\_, \varphi, \mu)$ $\mu(b) = (\_,c1,\_)$

$<svdl, newframe()> \rightarrow_{vdecl} \varphi_1$ $<ivdl, copy(\varphi)> \rightarrow_{vdecl} \varphi_2$

$<smdl, a, memptyenv()> \rightarrow_{mdecl} \mu_1$ $<imdl, a, \mu> \rightarrow_{mdecl} \mu_2$

$<$*class* a *extends* b {svdl smdl ivdl imdl c}, $\rho> \rightarrow_{cdecl}$
cbind(ρ, a, (b, φ_1, μ_1 , φ_2, mbind(μ_2, a, ([], c1 c, a))))

- il frame delle variabili di istanza viene costruito a partire da una copia di quello della superclasse
  - i frames sono strutture modificabili
  - è necessario costruire la copia
- l'ambiente dei metodi di istanza viene costruito a partire da quello della superclasse
  - gli ambienti di metodi sono funzioni
  - l'eventuale overriding è realizzato automaticamente da mbind
- il costruttore è ottenuto componendo (sintatticamente) il corpo del costruttore della superclasse con quello "nuovo"

Dichiarazioni di metodi

`Method_decl := Id (Idlist) Blocco`

- guardiamo solo la dichiarazione singola
  - per una lista di dichiarazioni, si ripete a partire dal risultato precedente

Menv = Id -> Mdescr

Mdescr = Idlist * Blocco * (Loc | Id)

- la relazione di transizione

Method_decl * Id * Menv $\rightarrow_{mdecl}$ Menv

Dichiarazione di metodo

Menv = Id -> Mdescr

Mdescr = Idlist * Blocco * (Loc | Id)

Method_decl * Id * Menv $\rightarrow_{mdecl}$ Menv

<m (idlist) blocco, c, μ> $\rightarrow_{mdecl}$
mbind(μ, m, (idlist, blocco, c))

- come già osservato, l'eventuale overriding è realizzato automaticamente da mbind

Dichiarazioni di variabili

`Var_decl := Type Id;`

- come già osservato, le dichiarazioni non hanno inizializzazione
- guardiamo solo la dichiarazione singola
  - per una lista di dichiarazioni, si ripete a partire dal risultato precedente

Frame = Id -> Val

Val = (Bool | Int | Loc)

- la relazione di transizione

Var_decl * Frame $\rightarrow_{vdecl}$ Frame

Dichiarazione di variabile

Frame = Id -> Val
Val = (Bool | Int | Loc)
Var_decl * Frame $\rightarrow_{vdecl}$ Frame

<t id;, φ> $\rightarrow_{vdecl}$ bind(φ, id, default(t))

- come già osservato, non ci deve essere overriding
  - id non deve essere definito in φ
- in realtà un Frame è una struttura modificabile
  - bind modifica l'argomento
- la funzione default genera un valore di tipo t
  - 0 se t è intero
  - null se t è il tipo di un oggetto

Espressioni

- dato che non abbiamo previsto inizializzazioni per le variabili (statiche e di istanza) all'interno delle classi, le espressioni possono comparire solo all'interno dei metodi
- consideriamo solo alcuni dei casi di espressioni
  - quelli che riguardano oggetti

Espressioni

`Expr := new Id` | (creazione di istanza)
`Path Id` (accesso al valore di una variabile)

`Path := Id. Objpath` | (si parte dalla classe Id)
`this. Objpath` | (si parte dall'oggetto corrente)
`Objpath`

`Objpath :=` | (cammino vuoto)
`Id. Objpath` (si parte dall'oggetto Id)

- ambigua, ma....

La relazione di transizione per le espressioni

`Expr := new Id` | (creazione di istanza)
`Path Id` (accesso al valore di una variabile)

Expr * Cenv * Heap * Astack $\rightarrow_{expr}$ Val * Heap

Creazione di oggetti

Expr * Cenv * Heap * Astack $\rightarrow_{expr}$ Val * Heap

$\rho(a) = (\_,\_,\_, \varphi, \mu)$ $l = newloc(\zeta)$
$\mu' = instantiate(\mu, l)$ $\varphi' = copy(\varphi)$
$\zeta'' = hbind(\zeta, l, (a, \varphi', \mu'))$ $\mu'(a) = (\_, b, \_)$
$\sigma' = push(\sigma, (l, push(emptystack(), newframe())))$
$<b, \rho, \zeta'', \sigma'> \rightarrow_{com} <\zeta''', \sigma'', \rho'>$ $pop(\sigma'')$

$<$*new* a, $\rho, \zeta, \sigma> \rightarrow_{expr} <l, \zeta'''>$

Creazione di oggetti: commenti

$\rho(a) = (\_,\_,\_, \varphi, \mu)$ $l = newloc(\zeta)$
$\mu' = instantiate(\mu, l)$ $\varphi' = copy(\varphi)$
$\zeta'' = hbind(\zeta, l, (a, \varphi', \mu'))$ $\mu'(a) = (\_, b, \_)$
$\sigma' = push(\sigma, (l, push(emptystack(), newframe())))$
$<b, \rho, \zeta'', \sigma'> \rightarrow_{com} <\zeta''', \sigma'', \rho'>$ $pop(\sigma'')$

$<$*new* a, $\rho, \zeta, \sigma> \rightarrow_{expr} <l, \zeta'''>$

- l'oggetto contiene
  - il nome della classe
  - una copia del frame delle variabili di istanza
  - una specializzazione sull'oggetto dell'ambiente di metodi di istanza
- il costruttore viene eseguito come una invocazione di metodo
  - non facciamo vedere i possibili effetti "nascosti" sui frames di variabili statiche (modifica di ρ da parte di $\rightarrow_{com}$)

Risoluzione di nomi (semplici)

- il riferimento ad un identificatore (non qualificato) può in generale essere risolto
  - nella pila di frames sulla testa dello stack delle attivazioni
  - nei frames delle variabili di istanza di un oggetto
  - nei frames di variabili statiche di una classe
- il modo di effettuare la ricerca dipende da dove essa inizia
  - un metodo (quello correntemente attivo)
  - un oggetto
  - una classe

Ide * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{naming}$ Val

Risoluzione di nomi 1

Ide * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{naming}$ Val

$\rho(c) = (c1, \varphi, \_, \_, \_)$ defined(φ, i)

$<i, c{:}Ide, \rho, \zeta, \sigma> \rightarrow_{naming} \varphi(i)$

$\rho(c) = (c1, \varphi, \_, \_, \_)$ not defined(φ, i)
$<i, c1, \rho, \zeta, \sigma> \rightarrow_{naming} v$

$<i, c{:}Ide, \rho, \zeta, \sigma> \rightarrow_{naming} v$

Risoluzione di nomi 2

Ide * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{naming}$ Val

$\zeta(l) = (c, \varphi, \_)$ defined(φ, i)

$<i, l{:}Loc, \rho, \zeta, \sigma> \rightarrow_{naming} \varphi(i)$

$\zeta(l) = (c, \varphi, \_)$ not defined(φ, i)
$<i, c, \rho, \zeta, \sigma> \rightarrow_{naming} v$

$<i, l{:}Loc, \rho, \zeta, \sigma> \rightarrow_{naming} v$

Risoluzione di nomi 3

Ide * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{naming}$ Val

$$\frac{top(\sigma) = (x, \tau) \quad \varphi = top(\tau) \quad defined(\varphi, i)}{<i, met, \rho, \zeta, \sigma> \rightarrow_{naming} \varphi(i)}$$

$$\frac{top(\sigma) = (x, \tau) \quad \varphi = top(\tau) \quad not\ defined(\varphi, i) \quad \sigma' = push(pop(\sigma), (x, pop(\tau))) \quad <i, met, \rho, \zeta, \sigma'> \rightarrow_{naming} v}{<i, met, \rho, \zeta, \sigma> \rightarrow_{naming} v}$$

$$\frac{top(\sigma) = (x, \tau) \quad empty(\tau) \quad <i, x, \rho, \zeta, \sigma> \rightarrow_{naming} v}{<i, met, \rho, \zeta, \sigma> \rightarrow_{naming} v}$$

Qualificazione dei nomi (paths)

`Path := Id. Objpath` | (si parte dalla classe Id)
`this. Objpath` | (si parte dall'oggetto corrente)
`Objpath`

`Objpath :=` | (cammino vuoto)
`Id. Objpath` (si parte dall'oggetto Id)

- la semantica di un path determina il punto di partenza di una operazione di naming

Path * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{path}$ (Ide | Loc | met)

Qualificazione dei nomi 1

Path * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{path}$ (Ide | Loc | met)

$$\frac{}{<, x, \rho, \zeta, \sigma> \rightarrow_{path} x}$$

$$\frac{top(\sigma) = (o, \_)}{<this, met, \rho, \zeta, \sigma> \rightarrow_{path} o}$$

$$\frac{cdefined(\rho, c)}{<c, met, \rho, \zeta, \sigma> \rightarrow_{path} c}$$

Qualificazione dei nomi 2

Path * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{path}$ (Ide | Loc | met)

$$\frac{not\ cdefined(\rho, o) \quad <o, x, \rho, \zeta, \sigma> \rightarrow_{naming} o'}{<o, x, \rho, \zeta, \sigma> \rightarrow_{path} o'}$$

$$\frac{<p, x, \rho, \zeta, \sigma> \rightarrow_{path} p' \quad <pl, p', \rho, \zeta, \sigma> \rightarrow_{path} o}{<p.pl, x, \rho, \zeta, \sigma> \rightarrow_{path} o}$$

Come trovare il valore di un nome qualificato

`Expr := Path Id`
Expr * Cenv * Heap * Astack $\rightarrow_{expr}$ Val * Heap

$$\frac{<p, met, \rho, \zeta, \sigma> \rightarrow_{path} o \quad <a, o, \rho, \zeta, \sigma> \rightarrow_{naming} v}{<p\ a, \rho, \zeta, \sigma> \rightarrow_{expr} <v, \zeta>}$$

- i controlli statici garantiscono che il nome (nel frame opportuno) abbia già ricevuto un valore (inizializzazione o assegnamento) prima dell'uso

I comandi

- un blocco (corpo di un metodo) è una sequenza di dichiarazioni e comandi (che possono anche essere blocchi nidificati)
- la composizione sequenziale viene trattata in modo standard
  - nel costrutto c;cl valutiamo cl nello stato risultante dalla valutazione di c
- trattiamo anche le dichiarazioni di variabili (locali al metodo) come comandi
- vediamo solo alcuni casi di comandi
  - condizionali e loops hanno la semantica vista nel primo corso

Com * Cenv * Heap * Astack $\rightarrow_{com}$ Heap * Astack * Cenv

Sintassi dei comandi

`Com := {Com}` | (blocco annidato)
`Path Ide = Expr` | (assegnamento)
`Type Ide = Expr` | (dichiarazione)
`Path Id (Expr_list)` | (invocazione di metodo)

Com * Cenv * Heap * Astack $\rightarrow_{com}$ Heap * Astack * Cenv

Semantica dei blocchi

`Com := {Com}` (blocco annidato)
Com * Cenv * Heap * Astack $\rightarrow_{com}$ Heap * Astack * Cenv

$top(\sigma) = (x, \tau)$ $\quad \sigma' = push(pop(\sigma), (x, push(\tau, newframe())))$
$\langle c, \rho, \zeta, \sigma' \rangle \rightarrow_{com} \langle \zeta', \sigma'', \rho' \rangle$ $\quad top(\sigma'') = (x, \tau')$

$\langle \{c\}, \rho, \zeta, \sigma \rangle \rightarrow_{com} \langle \zeta', push(pop(\sigma''), (x, pop(\tau'))), \rho' \rangle$

Semantica delle dichiarazioni

`Com := Type Ide = Expr` (dichiarazione)
Com * Cenv * Heap * Astack $\rightarrow_{com}$ Heap * Astack * Cenv

$top(\sigma) = (x, \tau)$ $\quad \varphi = top(\tau)$
$\langle e, \rho, \zeta, \sigma \rangle \rightarrow_{expr} \langle v, \zeta' \rangle$ $\quad \varphi' = bind(\varphi, i, v)$
$\sigma' = push(pop(\sigma), (x, push(pop(\tau), \varphi')))$

$\langle t\ i = e, \rho, \zeta, \sigma \rangle \rightarrow_{com} \langle \zeta', \sigma', \rho \rangle$

- la bind "modifica" φ dentro σ

Semantica dell'assegnamento 1

`Com := Path Ide = Expr` (assegnamento)
Com * Cenv * Heap * Astack $\rightarrow_{com}$ Heap * Astack * Cenv

- l'identificatore per cui va modificata l'associazione in un frame (a parte la qualificazione) può in generale essere
  - nella pila di frames sulla testa dello stack delle attivazioni
  - nei frames delle variabili di istanza di un oggetto
  - nei frames di variabili statiche di una classe
- il modo di effettuare la ricerca dipende da dove essa inizia
  - un metodo (quello correntemente attivo)
  - un oggetto
  - una classe
- simile al naming

Ide * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{update}$ Heap * Astack * Cenv

Semantica dell'assegnamento 2

`Com := Path Ide = Expr` (assegnamento)
Com * Cenv * Heap * Astack $\rightarrow_{com}$ Heap * Astack * Cenv

$\langle p, met, \rho, \zeta, \sigma \rangle \rightarrow_{path} c$
$\langle e, \rho, \zeta, \sigma \rangle \rightarrow_{expr} \langle v, \zeta'' \rangle$
$\langle i, v, c, \rho, \zeta'', \sigma \rangle \rightarrow_{update} \langle \zeta', \sigma', \rho' \rangle$

$\langle p\ i = e, \rho, \zeta, \sigma \rangle \rightarrow_{com} \langle \zeta', \sigma', \rho' \rangle$

Ide * Val * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{update}$ Heap * Astack * Cenv

Update 1

Ide * Val * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{update}$ Heap * Astack * Cenv

$\rho(c) = (c1, \varphi_1, \mu_1, \varphi_2, \mu_2)$ $\quad$ defined(φ_1, i) $\quad \varphi' = update(\varphi_1, i, v)$
$\rho' = cbind(\rho, c, (c1, \varphi', \mu_1, \varphi_2, \mu_2))$

$\langle i, v, c{:}Ide, \rho, \zeta, \sigma \rangle \rightarrow_{update} \langle \zeta, \sigma, \rho' \rangle$

$\rho(c) = (c1, \varphi, \_, \_, \_)$ $\quad$ not defined(φ, i)
$\langle i, v, c1, \rho, \zeta, \sigma \rangle \rightarrow_{update} \langle \zeta', \sigma', \rho' \rangle$

$\langle i, v, c{:}Ide, \rho, \zeta, \sigma \rangle \rightarrow_{update} \langle \zeta', \sigma', \rho' \rangle$

Update 2

Ide * Val * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{update}$ Heap * Astack * Cenv

$\zeta(l) = (c, \varphi, \mu)$ defined(φ, i) $\zeta' = hbind(\zeta, l, (c, update(\varphi, i, v), \mu))$

$<i, v, l:Loc, \rho, \zeta, \sigma> \rightarrow_{update} <\zeta', \sigma, \rho>$

$\zeta(l) = (c, \varphi, \_)$ not defined(φ, i)
$<i, v, c, \rho, \zeta, \sigma> \rightarrow_{update} <\zeta', \sigma', \rho'>$

$<i, v, l:Loc, \rho, \zeta, \sigma> \rightarrow_{update} <\zeta', \sigma', \rho'>$

Update 3

Ide * Val * (Ide | Loc | met) * Cenv * Heap * Astack $\rightarrow_{update}$ Heap * Astack * Cenv

$top(\sigma) = (x, \tau)$ $\varphi = top(\tau)$ defined(φ, i)
$\sigma' = push(pop(\sigma), (x, push(pop(\tau), update(\varphi, i, v))))$

$<i, v, met, \rho, \zeta, \sigma> \rightarrow_{update} <\zeta, \sigma', \rho>$

$top(\sigma) = (x, \tau)$ $\varphi = top(\tau)$ not defined(φ, i)
$\sigma' = push(pop(\sigma), (x, pop(\tau)))$ $<i, v, met, \rho, \zeta, \sigma'> \rightarrow_{update} <\zeta'', \sigma'', \rho'>$
$top(\sigma'') = (x, \tau')$ $\sigma''' = push(pop(\sigma''), (x, push(\tau', \varphi)))$

$<i, v, met, \rho, \zeta, \sigma> \rightarrow_{update} <\zeta'', \sigma''', \rho'>$

$top(\sigma) = (x, \tau)$ empty(τ) $<i, v, x, \rho, \zeta, \sigma> \rightarrow_{update} <\zeta'', \sigma'', \rho'>$

$<i, v, met, \rho, \zeta, \sigma> \rightarrow_{update} <\zeta'', \sigma'', \rho'>$

Invocazione di metodi

`Com := Path.Id (Expr_list)`

- path vuoto
  - paths trattati come nelle variabili
- un solo parametro

- vediamo, per prima cosa, come si ritrova un metodo di nome f
  - a partire da un oggetto o da una classe

Ide * (Ide | Loc) * Cenv * Heap $\rightarrow_{fmet}$
Ide * Blocco * (Ide | Loc)

Ricerca dei metodi 1

Ide * (Ide | Loc) * Cenv * Heap $\rightarrow_{fmet}$
Ide * Blocco * (Ide | Loc)

$\rho(c) = (c1, \_, \mu, \_, \_)$ mdefined(f, μ)

$<f, c:Ide, \rho, \zeta> \rightarrow_{fmet} \mu(f)$

$\rho(c) = (c1, \_, \mu, \_, \_)$ not mdefined(f, μ)
$<f, c1, \rho, \zeta> \rightarrow_{fmet} md$

$<f, c:Ide, \rho, \zeta> \rightarrow_{fmet} md$

Ricerca dei metodi 2

Ide * (Ide | Loc) * Cenv * Heap $\rightarrow_{fmet}$
Ide * Blocco * (Ide | Loc)

$\zeta(l) = (c, \_, \mu)$ mdefined(f, μ)

$<f, l:Loc, \rho, \zeta> \rightarrow_{fmet} \mu(f)$

$\zeta(l) = (c, \_, \mu)$ not mdefined(f, μ)
$<f, c, \rho, \zeta> \rightarrow_{fmet} md$

$<f, l:Loc, \rho, \zeta> \rightarrow_{fmet} md$

Invocazione di metodi

`Com := Id (Expr)`
Com * Cenv * Heap * Astack $\rightarrow_{com}$ Heap * Astack * Cenv
Ide * (Ide | Loc) * Cenv * Heap $\rightarrow_{fmet}$ Ide * Blocco * (Ide | Loc)

$top(\sigma) = (x, \_)$ $<f, x, \rho, \zeta> \rightarrow_{fmet} (par, b, y)$
$<e, \rho, \zeta, \sigma> \rightarrow_{expr} <v, \zeta'>$ $\varphi =$ bind(newframe(), par, v)
$\sigma' = push(\sigma, (y, push(emptystack(), \varphi)))$
$<b, \rho, \zeta', \sigma'> \rightarrow_{com} <\zeta'', \sigma'', \rho'>$

$<f(e), \rho, \zeta, \sigma> \rightarrow_{com} <\zeta'', pop(\sigma''), \rho'>$

Invocazione di metodi: commenti

$top(\sigma) = (x, \_)$ $\quad <f, x, \rho, \zeta> \rightarrow_{fmet} (par, b, y)$

$<e, \rho, \zeta, \sigma> \rightarrow_{expr} <v, \zeta' >$ $\quad \varphi = bind(newframe(), par, v)$

$\sigma' = push(\sigma, (y, push(emptystack(), \varphi)))$

$<b, \rho, \zeta', \sigma'> \rightarrow_{com} <\zeta'', \sigma'', \rho' >$

$<f(e), \rho, \zeta, \sigma)> \rightarrow_{com} <\zeta'', pop(\sigma''), \rho' >$

- si effettua la ricerca del metodo a partire dalla classe o dall'oggetto contenuti nella testa della pila
- si valuta il parametro attuale
- si crea un nuovo stack di frames, il cui unico frame contiene l'associazione tra parametro formale e valore del parametro attuale
- si pusha sulla pila il record che contiene la classe o l'oggetto associato al metodo e la pila di frames
- si valuta il corpo del metodo

Il naming

- naming
  - tutti gli usi di nomi all'interno dei metodi (inclusi quelli usati nei paths)
    - variabili locali, variabili di istanza, variabili statiche, metodi di istanza, metodi statici

  sono staticamente controllati per verificarne l'esistenza in accordo con le regole di visibilità
    - quelle che abbiamo "implementato" nei vari meccanismi di naming
- le regole di visibilità tengono anche conto degli attributi private, public, protected
- il meccanismo dei packages (con esportazioni ed importazioni)
  - serve per raggruppare insiemi di classi

  introduce ulteriori restrizioni

I tipi

- ogni entità (variabile, metodo) ha un tipo determinato staticamente
  - type checking statico
- nell'assegnamento e nel passaggio di parametri il tipo del valore è garantito solo essere un sottotipo di quello della variabile
- per le variabili che denotano oggetti, ci può essere una differenza tra
  - il tipo apparente = quello della dichiarazione della variabile o del parametro formale
  - il tipo effettivo a run-time = quello costruito valutando l'espressione in un assegnamento o il parametro attuale
  - il type checking statico può solo tener conto del tipo apparente
  - la differenza tra tipo apparente e tipo effettivo per gli oggetti si riflette anche sulla verifica dei nomi, che devono essere visibili secondo il tipo apparente

Altre osservazioni generali

- l'esecuzione inizia con l'invocazione del metodo (statico) main presente nell'ultima dichiarazione di classe
- astrazioni sui dati utili (come le stringhe e gli arrays) sono definite da classi "primitive"
- modificabilità delle strutture dati e condivisione (sharing)
  - gli oggetti di una classe possono essere
    - modificabili, se hanno delle operazioni che li modificano (per esempio, gli arrays)
    - non modificabili (per esempio, le stringhe)
  - un oggetto è condiviso da due entità se può essere raggiunto da entrambi
    - se è modificabile, le modifiche si riflettono su tutte le entità che lo "contengono"
- effetti laterali (all'esterno) in un metodo
  - attraverso i parametri può solo modificare valori di tipo oggetto
  - può modificare variabili di istanza e variabili (statiche) di classe che vede

Il garbage collector

- la gestione dinamica della heap è garantita dalla presenza di un garbage collector nella JVM
  - quando la heap è piena e non sarebbe più possibile creare dinamicamente nuovi oggetti, il garbage collector riconosce tutti gli oggetti che non sono più raggiungibili a partire dalla pila e dall'ambiente delle classi e li "raccoglie", rendendo disponibile la memoria relativa
- è forse la principale causa del successo di Java rispetto ad altri linguaggi orientati ad oggetti (C++)
  - dove la restituzione degli oggetti "inutili" è a carico del programmatore, con gravi rischi nel caso di oggetti condivisi
- prima di Java, il garbage collector era presente quasi esclusivamente in linguaggi funzionali (LISP, ML) e logici (PROLOG)
  - in cui le strutture allocate sulla heap sono molto più semplici e regolari

Java e affidabilità

- la affidabilità
  - intesa come protezione da errori a tempo di esecuzione

  è garantita al meglio in Java da vari meccanismi
- il controllo dei nomi statico
  - a run time non è possibile tentare di accedere un nome non definito
- il controllo dei tipi statico
  - a run time non si possono verificare errori di tipo (ma...)
- la presenza del garbage collector
  - non è possibile tentare di accedere oggetti non più presenti nella heap
- la presenza di controlli a tempo di esecuzione, con sollevamento di eccezioni
  - controllo sugli indici per gli arrays
  - accesso a puntatori ad oggetti vuoti (null)
  - conversioni forzate di tipo (casting)

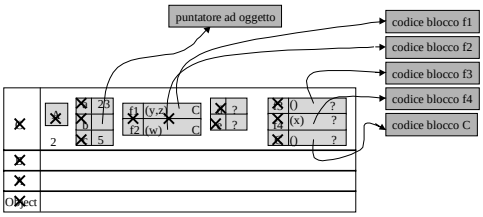
Risoluzione dei nomi nella compilazione

- il compilatore Java
  - anche per fare il controllo dei nomi

  costruisce staticamente delle tabelle che corrispondono ai frames (della pila, delle variabili di istanza, delle variabili statiche), agli ambienti dei metodi (statici e di istanza) ed all'ambiente delle classi
- la posizione di un nome (di variabile, di metodo o di classe) all'interno di queste tabelle è decisa una volta per tutte a tempo di compilazione
  - i nomi possono tutti scomparire dalle tabelle, che diventano vettori di valori, descrizioni di classe o descrizioni di metodi
  - i riferimenti ai nomi (tipicamente nel codice dei metodi) possono essere rimpiazzati da "displacements" (posizioni) in una opportuna tabella
- una ulteriore semplificazione è possibile per i metodi, dato che, a differenza dei valori delle variabili nei frames, essi non possono essere modificati
  - il riferimento al nome può essere direttamente rimpiazzato dal puntatore al codice e la tabella può essere eliminata
- questo non è possibile per i metodi di istanza, a causa della possibile differenza tra tipo apparente e tipo effettivo (dispatching)

109

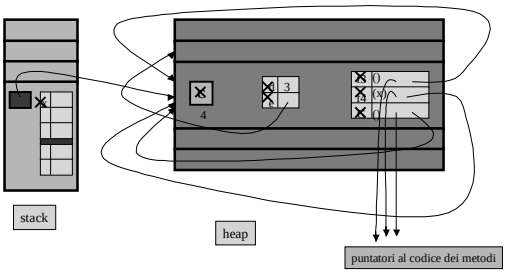
Come diventa lo stato (1)



- parte statica (classi)

110

Come diventa lo stato (2)



- parte dinamica (pila e heap)
  - è in esecuzione il metodo f4 di una istanza di C

111

Astrazione procedurale ed eccezioni

112

Procedure stand alone

- indipendenti da specifici oggetti
- come si realizzano in Java
  - insieme di metodi statici definiti dentro una classe che non ha
    - variabili e metodi di istanza
    - costruttore
  - può contenere variabili statiche
    - condivise dalle varie attivazioni di metodi
- una procedura è un mapping da un insieme di argomenti di ingresso ad un insieme di risultati
  - con possibile modifica di alcuni degli argomenti di ingresso
    - solo se sono oggetti
  - possibili effetti laterali su variabili di classe o di istanza visibili

113

Astrazione via specifica

- con la specifica, astraiamo dall'implementazione della procedura
- località
  - l'implementazione di una astrazione può essere letta o scritta senza esaminare le implementazioni delle altre astrazioni
    - utile durante lo sviluppo (anche da parte di più persone) e la manutenzione
- modificabilità
  - un'astrazione può essere reimplementata senza richiedere modifiche alle astrazioni che la utilizzano
    - utile durante la manutenzione per ridurre gli effetti indotti da una modifica

114

Un esempio di specifica

```
public class Arrays {
  // OVERVIEW: La classe fornisce un insieme di
  // procedure utili per manipolare arrays di int
  public static int search (int[] a, int x)
     // EFFECTS: se x occorre in a, ritorna un
   // indice in cui occorre, altrimenti -1
  public static int searchSorted (int[] a, int x)
     // REQUIRES: a è ordinata in modo crescente
     // EFFECTS: se x occorre in a, ritorna un
   // indice in cui occorre, altrimenti -1
  public static void sort (int[] a)
     // MODIFIES: a
     // EFFECTS: riordina gli elementi di a in
   // modo crescente, per esempio
     // se a=[3,1,6,1]  a_post=[1,1,3,6]
}
```


Un esempio: commenti 1

```
public class Arrays {
  // OVERVIEW: La classe fornisce un insieme di
  // procedure utili per manipolare arrays di int
  ....
}
```

- la classe compare nella specifica, perché i metodi dovranno essere reperiti usando il nome della classe

Un esempio: commenti 2

```
...
  public static int search (int[] a, int x)
     ...
  public static int searchSorted (int[] a, int x)
     ...
  public static void sort (int[] a)
     ...
```

- gli headers dei metodi (codice Java) sono la parte sintattica della specifica del metodo
- specificano (in aggiunta alla visibilità)
  - nome del metodo
  - nomi e tipi dei parametri formali
  - tipo del risultato
  - *search: int array * int -> int*
- dovrebbero anche elencare le eventuali eccezioni sollevate dalla procedura
  - ignorate per ora

Un esempio: commenti 3

```
...
  public static int search (int[] a, int x)
     // EFFECTS: se x occorre in a, ritorna un
   // indice in cui occorre, altrimenti -1
  public static int searchSorted (int[] a, int x)
     // REQUIRES: a è ordinata in modo crescente
     // EFFECTS: se x occorre in a, ritorna un
   // indice in cui occorre, altrimenti -1
...
```

- la clausola REQUIRES descrive le condizioni che devono essere verificate sui parametri di ingresso perché la procedura sia definita
  - possono esserci inputs impliciti (variabili visibili, files, etc.)
- se la clausola REQUIRES non è presente, la procedura è *totale* (esempio, *search*)
  - è definita per tutti gli inputs corretti rispetto al tipo
- altrimenti è *parziale* (esempio, *searchSorted*)

Un esempio: commenti 4

```
...
  public static void sort (int[] a)
     // MODIFIES: a
     // EFFECTS: riordina gli elementi di a in
   // modo crescente, per esempio
     // se a=[3,1,6,1]  a_post=[1,1,3,6]
...
```

- la clausola MODIFIES elenca tutti i parametri di ingresso che vengono modificati
  - compresi gli inputs impliciti (variabili visibili, files, etc.)
- se esistono parametri di ingresso che vengono modificati
  - la procedura produce effetti laterali

Un esempio: commenti 5

```
...
  public static int searchSorted (int[] a, int x)
     // REQUIRES: a è ordinata in modo crescente
     // EFFECTS: se x occorre in a, ritorna un
   // indice in cui occorre, altrimenti -1
  public static void sort (int[] a)
     // MODIFIES: a
     // EFFECTS: riordina gli elementi di a in
   // modo crescente, per esempio
     // se a=[3,1,6,1]  a_post=[1,1,3,6]
...
```

- la clausola EFFECTS descrive le proprietà degli outputs e le modifiche effettuate su tutti gli inputs elencati nella clausola MODIFIES
  - compresi gli inputs impliciti
- si suppone che siano verificate le proprietà specificate in REQUIRES
- *a_post* rappresenta il valore di *a* dopo il ritorno del metodo

Specifica ed implementazione

- per prima cosa si definisce la specifica
  - "scheletro" formato da headers e commenti
  - manca il codice dei corpi dei metodi
    - che può essere sviluppato in un momento successivo ed indipendentemente dallo sviluppo dei "moduli" che usano le procedure specificate
- comunque, l'implementazione deve "soddisfare" la specifica

Esempi di implementazione 1

```
public class Arrays {
  // OVERVIEW:  La classe fornisce un insieme di
  // procedure utili per manipolare arrays di int
  ...
  public static int searchSorted (int[] a, int x)
      // REQUIRES: a è ordinata in modo crescente
      // EFFECTS: se x occorre in a, ritorna un
    // indice in cui occorre, altrimenti -1
    // usa la ricerca lineare
  {if (a == null) return -1;
  for (int i = 0; i < a.length; i++)
          if (a[i] == x) return i;
     else if (a[i] > x) return -1;}
...
}
```

- la specifica (effects) è sottodeterminata
  - possiamo ottenere risultati diversi con diverse implementazioni
- se la precondizione non è soddisfatta, l'implementazione non è corretta

Esempi di implementazione 2.1

```
public class Arrays {
  // OVERVIEW: ...
 public static void sort (int[] a)
      // MODIFIES: a
      // EFFECTS: riordina gli elementi di a in modo
   // crescente, se a=[3,1,6,1]  a_post=[1,1,3,6]
    // usa il QuickSort
  {if (a == null) return;
  quickSort(a, 0, a.length - 1);}
...
}
```

- dobbiamo inserire nella classe il metodo `quickSort`
  - che può essere inserito come private
    - non visibile al di fuori della classe
- per i metodi private, potrebbe essere sufficiente l'implementazione
  - non esistono utenti esterni alla classe
- diamo anche la specifica, che può essere utile nella manutenzione

Esempi di implementazione 2.2

```
private static void quickSort (int[] a, int mi, int ma)
      // REQUIRES: a non è null, 0<=mi, ma<a.length
      // MODIFIES: a
      // EFFECTS: riordina gli elementi tra a[mi] e
   // a[ma] in modo crescente
  {if (mi >= ma) return;
  int mid = partition(a, mi, ma);
  quickSort(a, mi, mid);
  quickSort(a, mid + 1, ma);}
```

- dobbiamo inserire nella classe anche il metodo private `partition`
- quando possibile,
  - se non è troppo costoso (vedi `searchSorted`)

  l'implementazione dovrebbe verificare esplicitamente la precondizione
- la precondizione di quickSort è semplice da verificare, ma non lo facciamo
  - perché è destinata ad essere usata solo nel contesto di questa classe (private)
  - sappiamo che è sempre invocata in modo corretto

Esempi di implementazione 2.3

```
private static void partition (int[] a, int mi, int ma)
      // REQUIRES: a non è null, 0<=mi<ma<a.length
      // MODIFIES: a
      // EFFECTS: riordina gli elementi tra a[mi] e
   // a[ma] in due gruppi mi..ris e ris+1..ma, tali
   // che tutti gli elementi del secondo gruppo sono
   // >= di quelli del primo; ritorna ris
  {int x = a[mi];
  while (true) {
    while (a[ma] > x) ma--;
    while (a[mi] < x) mi++;
    if (mi < ma) {
       int temp = a [mi]; a[mi] = a[ma]; a[ma] = temp;
       ma--; mi++;}
    else return ma; }
  }}
```

- come in quickSort non verifichiamo la precondizione perché sappiamo che è sempre invocata in modo corretto nel suo contesto di uso

Procedure ed eccezioni

- durante l'esecuzione di una procedura si possono verificare varie situazioni che possiamo considerare eccezionali
  - generazione di errori a run time la cui presenza non può essere verificata a tempo di compilazione
    - accesso ad un elemento di un array con indice "scorretto"
    - accesso a puntatori ad oggetti vuoti (null)
    - impossibilità di effettuare conversioni forzate di tipo (casting)
  - non è verificata la precondizione (procedure parziali)
    - potrebbe succedere di tutto, dal ritorno di risultati privi di significato, alla non terminazione, al danneggiamento di dati permanenti
  - anche se la precondizione è verificata, possono esserci valori degli inputs, per i quali la procedura ha un comportamento particolare
    - per esempio, ritorna valori speciali con cui si informa il chiamante della situazione

Precondizione non soddisfatta

- non è verificata la precondizione
  - potrebbe succedere di tutto, dal ritorno di risultati privi di significato, alla non terminazione, al danneggiamento di dati permanenti

```
public static int mcd (int n, int d)
    // REQUIRES: n, d > 0
    // EFFECTS: ritorna il massimo comun divisore
  // di n e d
```

- chiunque utilizzi la procedura deve preoccuparsi di verificare che i dati passati verifichino la precondizione
  - chi lo garantisce?
- chi implementa la procedura può ignorare i casi non previsti

Comportamenti particolari

- anche se la precondizione è verificata, possono esserci valori degli inputs, per i quali la procedura ha un comportamento particolare
  - per esempio, ritorna valori speciali con cui si informa il chiamante della situazione

```
public static int search (int[] a, int x)
    // EFFECTS: se x occorre in a, ritorna un
  // indice in cui occorre, altrimenti -1

public static int fact (int n)
    // EFFECTS: se n>0, ritorna n!, altrimenti 0
```

- il chiamante deve comunque trattare in modo speciale il valore che codifica la situazione particolare
  - chi lo garantisce?

Robustezza

- le procedure parziali e la codifica di situazioni particolari portano a programmi poco robusti
- un programma robusto si comporta in modo ragionevole anche in presenza di "errori" (graceful degradation)
  - per esempio, continua dopo il verificarsi dell'errore con un comportamento ben-definito che approssima quello normale
  - come minimo, termina con un messaggio di errore "informativo" senza danneggiare dati permanenti
- cosa serve?
  - un meccanismo (o approccio) che trasferisca l'informazione al chiamante in tutte queste situazioni
  - distinguendo le varie situazioni
  - con una gestione delle situazioni "strane" separata dal flusso di controllo normale della procedura

Il meccanismo delle eccezioni

- una procedura può terminare
  - normalmente, ritornando un risultato
  - in modo eccezionale
    - ci possono essere diverse terminazioni eccezionali
    - in Java, corrispondono a diversi tipi di eccezioni
    - il nome del tipo di eccezione viene scelto da chi specifica la procedura per fornire informazione sulla natura del problema
- le eccezioni giocano un ruolo molto importante nell'astrazione via specifica
  - la specifica del comportamento deve riguardare anche le terminazioni eccezionali

Eccezioni nella specifica

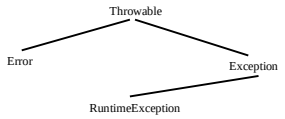
```
public static int fact (int n) throws NonpositiveExc
    // EFFECTS: se n>0, ritorna n!
  // altrimenti solleva NonpositiveExc

public static int searchSorted (int[] a, int x) throws
  NullPointerExc, NotFoundExc
    // REQUIRES: a è ordinato in modo crescente
      // EFFECTS: se a è null solleva NullPointerExc
  // se x non occorre in a solleva NotFoundExc
  // altrimenti ritorna un indice in cui occorre
```

- le procedure possono continuare ad essere parziali
  - verificare la precondizione e sollevare un'eccezione ridurrebbe in modo inaccettabile l'efficienza di `searchSorted`
  - la specifica del comportamento eccezionale presume comunque che l'eventuale precondizione sia soddisfatta

Le eccezioni in Java

- i tipi di eccezione sono particolari classi che
  - contengono solo il costruttore
    - ci possono essere più costruttori overloaded
  - sono definite in "moduli" separati da quelli che contengono i metodi che le possono sollevare
- le eccezioni sono oggetti
  - creati eseguendo new di un exception type
    - e quindi eseguendo il relativo costruttore
- esiste una gerarchia "predefinita" di tipi relativi alle eccezioni
  - nuovi tipi di eccezioni sono collocati nella gerarchia con l'usuale extends

La gerarchia di tipi per le eccezioni



- se un nuovo tipo di eccezione estende la classe `Exception`
  - l'eccezione è checked
- se un nuovo tipo di eccezione estende la classe `RuntimeException`
  - l'eccezione è unchecked

Eccezioni checked e unchecked

- se una procedura può sollevare una eccezione checked
  - deve elencarla nel suo header
    - che fa parte anche della specifica
    - altrimenti si verifica un errore a tempo di compilazione
- se una procedura può sollevare una eccezione unchecked
  - può non elencarla nel suo header
    - il suggerimento è di elencarla sempre, per rendere completa la specifica
- se una procedura chiamata da p ritorna sollevando una eccezione
  - se l'eccezione è checked
    - p deve gestire l'eccezione (try and catch, vedi dopo)
    - se l'eccezione (o uno dei suoi supertipi) è elencata tra quelle sollevabili da p, può essere propagata alla procedura che ha chiamato p
  - se l'eccezione è unchecked
    - può essere comunque gestita o propagata

Eccezioni primitive

- ne esistono numerose, sia checked che unchecked
  - `NullPointerException` e `IndexOutOfBoundsException` sono unchecked
  - `IOException` è checked

Definire tipi di eccezione

```
public class NuovoTipoDiEcc extends Exception {
  public NuovoTipoDiEcc(string s) {super(s);}
}
```

- è checked
- definisce solo un costruttore
  - come sempre invocato quando si crea una istanza con la new
  - il costruttore può avere parametri
- il corpo del costruttore riutilizza semplicemente il costruttore del supertipo
  - perché deve passargli il parametro
- una new di questa classe provoca la creazione di un nuovo oggetto che "contiene" la stringa passata come parametro

Costruire oggetti eccezione

```
public class NuovoTipoDiEcc extends Exception {
  public NuovoTipoDiEcc(string s) {super(s);}
}
```

- una new di questa classe provoca la creazione di un nuovo oggetto che "contiene" la stringa passata come parametro

```
Exception e = new NuovoTipoDiEcc ("Questa è la ragione") ;
String s = e.toString() ;
```

- la variabile `s` punta alla stringa
  `"NuovoTipoDiEcc: Questa è la ragione"`

Sollevare eccezioni

- una procedura può terminare
  - (ritorno normale) con un `return`
  - (ritorno di una eccezione) con un `throw`

```
public static int fact (int n) throws NonpositiveExc
    // EFFECTS: se n>0, ritorna n!
  // altrimenti solleva NonpositiveExc
{ if (n <= 0) throw new NonPositiveExc("Num.fact");
 ...}
```

- la stringa contenuta nell'eccezione è utile soprattutto quando il programma non è in grado di "gestire" l'eccezione
  - permette all'utente di identificare la procedura che la ha sollevata
  - può comparire nel messaggio di errore che si stampa subito prima di forzare la terminazione dell'esecuzione

Gestire eccezioni

- quando una procedura termina con un `throw`
  - l'esecuzione non riprende con quello che segue la chiamata
  - ma il controllo viene trasferito ad un pezzo di codice preposto alla gestione dell'eccezione
- due possibilità per la gestione
  - gestione esplicita quando l'eccezione è sollevata all'interno di uno statement `try`
    - in generale, quando si ritiene di poter recuperare uno stato consistente e di portare a termine una esecuzione quasi "normale"
  - gestione di default, mediante propagazione dell'eccezione alla procedura chiamante
    - possibile solo per eccezioni non checked o per eccezioni checked elencate nell'header della procedura che riceve l'eccezione

Gestire esplicita delle eccezioni

- gestione esplicita quando l'eccezione è sollevata all'interno di uno statement `try`
- codice per gestire l'eccezione `NonPositiveExc` eventualmente sollevata da una chiamata di `fact`

```
try { x = Num.fact (y); }
catch (NonPositiveExc e) {
  // qui possiamo usare e, cioè l'oggetto eccezione
}
```

- la clausola `catch` non deve necessariamente identificare il tipo preciso dell'eccezione, ma basta un suo supertipo

```
try { x = Arrays.searchSorted (v, y); }
catch (Exception e) { s.Println(e); return;}
  // s è una PrintWriter
```

- segnala l'informazione sia su `NullPointerExc` che su `NotFoundExc`

`Try` e `Catch` annidati

```
try { ...;
  try { x = Arrays.searchSorted (v, y); }
  catch (NullPointerExc e) {
  throw new NotFoundExc ();}
    }
catch (NotFoundExc b ) {...}
```

- la clausola `catch` nel `try` più esterno cattura l'eccezione `NotFoundExc` se è sollevata da `searchSorted` o dalla clausola `catch` più interna

Catturare eccezioni unchecked

- le eccezioni unchecked sono difficili da catturare
  - una qualunque chiamata di procedura può sollevarle
  - difficile sapere da dove vengono

```
try { x = y[n]; i = Arrays.searchSorted (v, x); }
catch (IndexOutOfBoundsException e) {
  // cerchiamo di gestire l'eccezione pensando che sia
  // stata sollevata da x = y[n]
}
// continuiamo supponendo di aver risolto il problema
```

- ma l'eccezione poteva venire dalla chiamata a `searchSorted`
- l'unico modo per sapere con certezza da dove viene è restringere lo scope del comando `try`

Aspetti metodologici

- gestione delle eccezioni
  - riflessione
  - mascheramento
- quando usare le eccezioni
- come scegliere tra checked e unchecked
- defensive programming

Gestione delle eccezioni via riflessione

- se una procedura chiamata da p ritorna sollevando una eccezione, anche p termina sollevando un'eccezione
  - usando la propagazione automatica
    - della stessa eccezione (`NullPointerException`)
  - catturando l'eccezione e sollevandone un'altra
    - possibilmente diversa (`EmptyException`)

```
public static int min (int[] a) throws NullPointerException, EmptyException
    // EFFECTS: se a è null solleva NullPointerException
  // se a è vuoto solleva EmptyException
  // altrimenti ritorna il minimo valore in a
{int m;
try { m = a[0]}
catch(IndexOutOfBoundsException e) {
  throws new EmptyException("Arrays.min");}
for (int i = 1; i < a.length ; i++)
  if (a[i] < m) m = a[i];
return m;}
```

Gestione delle eccezioni via mascheramento

- se una procedura chiamata da p ritorna sollevando una eccezione, p gestisce l'eccezione e ritorna in modo normale

```
public static boolean sorted (int[] a) throws NullPointerException
    // EFFECTS: se a è null solleva NullPointerException
  // se a è ordinato in senso crescente ritorna true
  // altrimenti ritorna false
{int prec;
try { prec = a[0]}
catch(IndexOutOfBoundsException e) { return true;}
for (int i = 1; i < a.length ; i++)
  if (prec <= a[i]) prec = a[i]; else return false;
return true;}
```

- come nell'esempio precedente, usiamo le eccezioni (catturate) al posto di un test per verificare se a è vuoto

Quando usare le eccezioni

- le eccezioni non sono necessariamente errori
  - ma metodi per richiamare l'attenzione del chiamante su situazioni particolari (classificate dal progettista come eccezionali)
- comportamenti che sono errori ad un certo livello, possono non esserlo affatto a livelli di astrazione superiore
  - `IndexOutOfBoundsException` segnala chiaramente un errore all'interno dell'espressione `a[0]` ma non necessariamente per le procedure `min` e `sort`
- il compito primario delle eccezioni è di ridurre al minimo i vincoli della clausola REQUIRES nella specifica
  - dovrebbe restare solo se
    - la condizione è troppo complessa da verificare (efficienza)
    - il contesto d'uso limitato del metodo (private) ci permette di convincerci che tutte le chiamate della procedura la soddisfano
- vanno usate per evitare di codificare informazione su terminazioni particolari nel normale risultato

Checked o unchecked

- le eccezioni checked offrono una maggiore protezione dagli errori
  - sono più facili da catturare
  - il compilatore controlla che l'utente le gestisca esplicitamente o per lo meno le elenchi nell'header, prevedendone una possibile propagazione automatica
    - se non è così, viene segnalato un errore
- le eccezioni checked possono essere (per la stessa ragione) pesanti da gestire in quelle situazioni in cui siamo ragionevolmente sicuri che l'eccezione non verrà sollevata
  - perché esiste un modo conveniente ed efficiente di evitarla
  - per il contesto di uso limitato
  - solo in questi casi si dovrebbe optare per una eccezione unchecked

Defensive programming

- l'uso delle eccezioni facilita uno stile di progettazione e programmazione che protegge rispetto agli errori
  - anche se non sempre un'eccezione segnala un errore
- fornisce una metodologia che permette di riportare situazioni di errore in modo ordinato
  - senza disperdere tale compito nel codice che implementa l'algoritmo
- nella programmazione defensive, si incoraggia il programmatore a verificare l'assenza di errori ogniqualvolta ciò sia possibile
  - ed a riportarli usando il meccanismo delle eccezioni
  - un caso importante legato alle procedure parziali

Quando una procedura non soddisfa la sua precondizione

- con le eccezioni le procedure tendono a diventare totali
  - ma non è sempre possibile
- chi chiama la procedura dovrebbe farsi carico di effettuare tale controllo
  - sollevando una eccezione unchecked
    - non elencata nell'header e non considerata negli EFFECTS, perché si riferisce ad un caso che non soddisfa REQUIRES
    - questa eccezione può essere catturata, magari ad un livello superiore
      - si suggerisce di usare in questi casi una eccezione generica unchecked `FailureException`

Astrazioni sui dati : Specifica ed Implementazione di Tipi di Dato Astratti in Java

Specifica ed Implementazione di Tipi di Dato Astratti in Java

- cos'è un tipo di dato astratto
- specifica di tipi di dati astratti
  - un tipo modificabile (`IntSet`)
  - qualche tipo primitivo
  - un tipo non modificabile (`Poly`)
- implementazione di tipi di dati astratti
  - definire la rappresentazione
    - le restrizioni nell'uso di Java
    - un record type (l'eccezione!)
  - l'implementazione di `IntSet` e `Poly`
- alcuni metodi "ereditabili"

Perché l'astrazione sui dati

- il più importante tipo di astrazione
- il nucleo (non esclusivo) della programmazione orientata ad oggetti
- lo scopo è quello di estendere il linguaggio
  - nel nostro caso, Java

  con nuovi tipi di dato
- quali?
  - dipende dall'applicazione
    - interprete: stacks e tabelle di simboli
    - applicazione bancaria: conti
    - applicazioni numeriche: matrici

Cos'è un'astrazione sui dati

- in ogni caso è
  - un insieme di oggetti
    - stacks, conti, matrici
  - + un insieme di operazioni
    - per crearli e manipolarli
- i nuovi tipi di dato dovrebbero incorporare i due meccanismi di astrazione
  - parametrizzazione
    - simile al caso delle astrazioni procedurali
    - quella più importante si realizza con il polimorfismo
      - che vedremo più avanti
  - specifica
    - porta a vedere necessariamente insieme oggetti ed operazioni

Astrazione sui dati via specifica

- con la specifica, astraiamo dall'implementazione del tipo di dato
  - dalla sua rappresentazione
- se avessimo solo la rappresentazione degli oggetti non sarebbe possibile
  - o l'utente opera direttamente sulla rappresentazione
    - non ci sarebbe astrazione
    - se la rappresentazione viene modificata, la modifica si ripercuote su tutti gli utenti
  - oppure non si rende possibile la manipolazione degli oggetti del nuovo tipo
- avendo gli oggetti insieme alle operazioni, l'astrazione diventa possibile
  - la rappresentazione è nascosta all'utente esterno, mentre è visibile all'implementazione delle operazioni
  - se una rappresentazione viene modificata, devono essere modificate le implementazioni delle operazioni, ma non le astrazioni che la utilizzano
    - è il tipo di modifica più comune durante la manutenzione

Gli ingredienti della specifica di un tipo di dato astratto

- Java (parte sintattica della specifica)
  - classe o interfaccia
    - per ora solo classi
  - nome per il tipo
    - nome della classe
  - operazioni
    - metodi di istanza
    - incluso il(i) costruttore(i)
- la specifica del tipo descrive proprietà generali degli oggetti
  - per esempio la modicabilità
- per il resto la specifica è essenzialmente una specifica dei metodi
  - strutturata come già abbiamo visto per le astrazioni procedurali
  - l'oggetto su cui i metodi operano è indicato nella specifica da this

Formato della specifica

```
public class NuovoTipo {
  // OVERVIEW: Gli oggetti di tipo NuovoTipo
  // sono collezioni modificabili di ..

  // costruttori
  public NuovoTipo ()
     // EFFECTS: ...

  // metodi
  // specifiche degli altri metodi
}
```

L'insieme di interi 1

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 // costruttore
  public IntSet ()
     // EFFECTS: inizializza this a vuoto
 // metodi
  public void insert (int x)
     // MODIFIES: this
     // EFFECTS: aggiunge x a this
  public void remove (int x)
     // MODIFIES: this
     // EFFECTS: toglie x da this
  public boolean isIn (int x)
     // EFFECTS: se x appartiene a this ritorna
   // true, altrimenti false
  ...}
```


L'insieme di interi 2

```
public class IntSet {
 ...
 // metodi
 ...
 public int size ()
     // EFFECTS: ritorna la cardinalità di this
 public int choose () throws EmptyException
     // EFFECTS: se this è vuoto, solleva
   // EmptyException, altrimenti ritorna un
   // elemento qualunque contenuto in this
}
```


IntSet: commenti 1

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
  ....
}
```

- gli oggetti della classe sono descritti nella specifica in termini di concetti noti
  - in questo caso, gli insiemi matematici
- gli stessi concetti sono anche utilizzati nella specifica dei metodi
  - aggiungere, togliere elementi
  - appartenenza, cardinalità

IntSet: commenti 2

```
public class IntSet {
 // OVERVIEW: ...
 // costruttore
  public IntSet ()
     // EFFECTS: inizializza this a vuoto
 ...}
```

- un solo costruttore (senza parametri)
  - inizializza this (l'oggetto nuovo)
  - non è possibile vedere lo stato dell'oggetto tra la creazione e l'inizializzazione
    - la specifica non ha una clausola MODIFIES

IntSet: commenti 3

```
public class IntSet {
 ...
 // metodi
  public void insert (int x)
     // MODIFIES: this
     // EFFECTS: aggiunge x a this
  public void remove (int x)
     // MODIFIES: this
     // EFFECTS: toglie x da this
 ...}
```

- modificatori
  - modificano lo stato del proprio oggetto (`MODIFIES: this`)
  - notare che nè `insert` nè `remove` sollevano eccezioni
    - se si inserisce un elemento che c'è già
    - se si rimuove un elemento che non c'è

IntSet: commenti 4

```
 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 int choose () throws EmptyException
     // EFFECTS: se this è vuoto, solleva
   // EmptyException, altrimenti ritorna un
   // elemento qualunque contenuto in this...}
```

- osservatori
  - non modificano lo stato del proprio oggetto
  - `choose` può sollevare un'eccezione (se l'insieme è vuoto)
    - `EmptyException` può essere unchecked, perché l'utente può utilizzare `size` per evitare di farla sollevare
    - `choose` è sottodeterminata (implementazioni corrette diverse possono dare diversi risultati)

Specifica di un tipo “primitivo”

- le specifiche sono ovviamente utili per capire ed utilizzare correttamente i tipi di dato “primitivi” di Java
- vedremo, come esempio, il caso dei vettori
  - `Vector`
  - arrays dinamici che possono crescere e accorciarsi
  - sono definiti nel package `java.util`

`Vector` 1

```
public class Vector {
 // OVERVIEW: un Vector è un array modificabile
 // di dimensione variabile i cui elementi sono
 // di tipo Object: indici tra 0 e size - 1
 // costruttore
  public Vector ()
     // EFFECTS: inizializza this a vuoto
 // metodi
  public void add (Object x)
      // MODIFIES: this
      // EFFECTS: aggiunge una nuova posizione a
   // this inserendovi x
  public int size ()
      // EFFECTS: ritorna il numero di elementi di
   // this
  ...}
```


`Vector` 2

```
...
 public Object get (int n) throws IndexOutOfBoundsException
      // EFFECTS: se n<0 o n>= this.size solleva
    // IndexOutOfBoundsException, altrimenti
    // ritorna l'oggetto in posizione n in this
 public void set (int n, Object x) throws IndexOutOfBoundsException
    // MODIFIES: this
      // EFFECTS: se n<0 o n>= this.size solleva
    // IndexOutOfBoundsException, altrimenti
    // modifica this sostituendovi l'oggetto x in
    // posizione n
 public void remove (int n) throws IndexOutOfBoundsException
      // MODIFIES: this
      // EFFECTS: se n<0 o n>= this.size solleva
    // IndexOutOfBoundsException, altrimenti
    // modifica this eliminando l'oggetto in
    // posizione n
}
```


`Vector`: commenti 1

```
public class Vector {
 // OVERVIEW: un Vector è un array modificabile
 // di dimensione variabile i cui elementi sono
 // di tipo Object: indici tra 0 e size - 1
  ....
}
```

- gli oggetti della classe sono descritti nella specifica in termini di concetti noti
  - in questo caso, gli arrays
- gli stessi concetti sono anche utilizzati nella specifica dei metodi
  - indice, elemento identificato dall’indice
- il tipo è modificabile
  - come l’array
- notare che gli elementi sono di tipo Object
  - per esempio, non possono essere int, bool e char

`Vector`: commenti 2

```
public class Vector {
 // OVERVIEW: un Vector è un array modificabile
 // di dimensione variabile i cui elementi sono
 // di tipo Object: indici tra 0 e size - 1
 // costruttore
  public Vector ()
     // EFFECTS: inizializza this a vuoto
 ...}
```

- un solo costruttore (senza parametri)
  - inizializza this (l’oggetto nuovo) ad un “array” vuoto

`Vector`: commenti 3

```
public void add (Object x)
      // MODIFIES: this
      // EFFECTS: aggiunge una nuova posizione a
   // this inserendovi x
public void set (int n, Object x) throws IndexOutOfBoundsException
   // MODIFIES: this
      // EFFECTS: se n<0 o n>= this.size solleva
   // IndexOutOfBoundsException, altrimenti modifica
   // this sostituendovi l'oggetto x in posizione n
public void remove (int n) throws IndexOutOfBoundsException
      // MODIFIES: this
      // EFFECTS: se n<0 o n>= this.size solleva
   // IndexOutOfBoundsException, altrimenti modifica
   // this eliminando l'oggetto in posizione n
```

- sono modificatori
  - modificano lo stato del proprio oggetto (`MODIFIES: this`)
  - `set` e `remove` possono sollevare un’eccezione primitiva unchecked

Vector: commenti 4

```
public int size ()
    // EFFECTS: ritorna il numero di elementi di
  // this
public Object get (int n) throws IndexOutOfBoundsException
    // EFFECTS: se n<0 o n>= this.size solleva
  // IndexOutOfBoundsException, altrimenti
  // ritorna l'oggetto in posizione n in this
public Object lastElement ()
    // EFFECTS: ritorna l'ultimo oggetto in this
```

- sono osservatori
  - non modificano lo stato del proprio oggetto
  - get può sollevare un'eccezione primitiva unchecked

I polinomi 1

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // cofficienti interi non modificabile
 // esempio: c0 + c1*x + c2*x^2 + ...

 // costruttori
  public Poly ()
    // EFFECTS: inizializza this al polinomio 0
  public Poly (int c, int n) throws
    NegativeExponentExc
     // EFFECTS: se n<0 solleva NegativeExponentExc
   // altrimenti inizializza this al polinomio cx^n

 // metodi

  ...}
```


I polinomi 2

```
public class Poly {
  ...
  // metodi
   public int degree ()
      // EFFECTS: ritorna 0 se this è il polinomio
    // 0, altrimenti il più grande esponente con
    // coefficiente diverso da 0 in this
   public int coeff (int d)
      // EFFECTS: ritorna il coefficiente del
    // termine in this che ha come esponente d
   public Poly add (Poly q) throws
      NullPointerException
     // EFFECTS: q=null solleva NullPointerException
   // altrimenti ritorna this + q
  ...}
```


I polinomi 3

```
public class Poly {
  ...
  // metodi
   ...
   public Poly mul (Poly q) throws
     NullPointerException
     // EFFECTS: q=null solleva NullPointerException
   // altrimenti ritorna this * q
   public Poly sub (Poly q) throws
     NullPointerException
     // EFFECTS: q=null solleva NullPointerException
   // altrimenti ritorna this - q
   public Poly minus ()
     // EFFECTS: ritorna -this
}
```


Poly: commenti 1

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // cofficienti interi non modificabile
 // esempio: c0 + c1*x + c2*x^2 + ...
  ...}
```

- gli oggetti della classe sono descritti nella specifica in termini di concetti noti
  - in questo caso, i polinomi
- gli stessi concetti sono anche utilizzati nella specifica dei metodi
  - operazioni di +, *, e -

Poly: commenti 2

```
public class Poly {
 // OVERVIEW: ...
 // costruttori
  public Poly ()
    // EFFECTS: inizializza this al polinomio 0
  public Poly (int c, int n) throws
    NegativeExponentExc
     // EFFECTS: se n<0 solleva NegativeExponentExc
   // altrimenti inizializza this al polinomio cx^n
 ...}
```

- due costruttori overloaded
  - stesso nome (quello della classe, in questo caso)
  - diverso numero o tipo di parametri
    - se no, errore di compilazione
  - la scelta tra metodi overloaded viene effettuata in base al numero e tipo di parametri
    - eventualmente a run time scegliendo il più specifico

Poly: commenti 3

```
public class Poly {
 // OVERVIEW: ...
 // costruttori
  public Poly ()
     // EFFECTS: inizializza this al polinomio 0
  public Poly (int c, int n) throws
    NegativeExponentExc
     // EFFECTS: se n<0 solleva NegativeExponentExc
   // altrimenti inizializza this al polinomio cx^n
...}
```

- l'eccezione `NegativeExponentExc` non è definita qui
  - nello stesso package di `Poly`?
  - può essere unchecked, perché non è probabile che l'utente usi esponenti negativi

Poly: commenti 4

```
// metodi
  public int degree ()
     // EFFECTS: ritorna 0 se this è il polinomio
   // 0, altrimenti il più grande esponente con
   // coefficiente diverso da 0 in this
  public int coeff (int d)
     // EFFECTS: ritorna il coefficiente del
   // termine in this che ha come esponente d
  public Poly add (Poly q) throws
    NullPointerException
     // EFFECTS: q=null solleva NullPointerException
  // altrimenti ritorna this + q
...
```

- non ci sono modificatori
  - il tipo è non modificabile!
  - `degree` e `coeff` sono osservatori
  - `add`, `mul`, `sub` e `minus` ritornano nuovi oggetti di tipo `Poly`

Proprietà della Specifica

- per prima cosa si definisce la specifica
  - "scheletro" formato da headers e commenti
  - mancano la rappresentazione degli oggetti ed il codice dei corpi dei metodi
    - che possono essere sviluppati in un momento successivo ed indipendentemente dallo sviluppo dei "moduli" che usano il nuovo tipo di dato
    - è molto importante riuscire a differire le scelte relative alla rappresentazione
  - se aggiungiamo corpi contenenti return ben-tipati alla specifica dei metodi
    - la specifica può essere compilata
    - possono essere compilate implementazioni di moduli che la utilizzano (errori rilevati subito dall'analisi statica)

Implementazione di un metodo che usa IntSet

```
public static IntSet getElements (int[] a) throws
    NullPointerException
     // EFFECTS: a=null solleva NullPointerException
   // altrimenti, restituisce un insieme che contiene
   // tutti e soli gli interi presenti in a
{IntSet s = new IntSet();
 for (int i = 0; i < a.length; i++)
        s.insert(a[i]);
 return s;
}
```

- scritta solo conoscendo la specifica di `IntSet`
  - non accede all'implementazione
    - non esiste ancora
    - anche se ci fosse non potrebbe "vederla"
  - costruisce, accede e modifica l'oggetto solo attraverso i metodi (incluso il costruttore)

Implementazione di un metodo che usa Poly

```
public static Poly diff (Poly p) throws
    NullPointerException
     // EFFECTS: p=null solleva NullPointerException
   // altrimenti, restituisce il polinomio che risulta
   // dalla differenziazione di p
{Poly q = new Poly();
 for (int i = 0; i <= p.degree(); i++)
        q = q.add(new Poly(p.coeff(i)*i, i-1));
 return q;
}
```

- scritta solo conoscendo la specifica di `Poly`
  - non accede all'implementazione
    - non esiste ancora
    - anche se ci fosse non potrebbe "vederla"
  - costruisce ed accede l'oggetto solo attraverso i metodi (incluso il costruttore)

Astrazioni sui dati: implementazione

- scelta fondamentale è quella della rappresentazione (rep)
  - come i valori del tipo astratto sono implementati in termini di altri tipi
    - tipi primitivi o già implementati
    - nuovi tipi astratti che facilitano l'implementazione del nostro
      - tali tipi vengono specificati
      - iterazione del processo di decomposizione basato su astrazioni
  - la scelta deve tener nel dovuto conto la possibilità di implementare in modo efficiente i costruttori e gli altri metodi
- poi viene l'implementazione dei costruttori e dei metodi

La rappresentazione

- i linguaggi che permettono la definizione di tipi di dato astratti hanno meccanismi molto diversi tra loro per definire come
  - i valori del nuovo tipo sono implementati in termini di valori di altri tipi
- in Java, gli oggetti del nuovo tipo sono semplicemente collezioni di valori di altri tipi
  - definite (nella implementazione della classe) da un insieme di variabili di istanza private
    - accessibili solo dai costruttori e dai metodi della classe
- diversi meccanismi nei paradigmi funzionale e imperativo (senza oggetti)

181

Definire un tipo in ML

- i valori di un tipo sono alberi etichettati (termini), che hanno sulle foglie i valori dei tipi utilizzati
- un tipo descrive l'insieme di tutti i possibili valori mediante definizioni di tipo
  - date per casi
  - possibilmente ricorsive
- i polinomi in ML

```
type poly = Term of int * int | Plus of poly * poly
```

  - comprende anche valori "non legali"
    - termini diversi con lo stesso coefficiente
    - le operazioni si preoccupano di generare solo i valori buoni
  - mostra esplicitamente che il tipo è ricorsivo
  - descrive esplicitamente tutti i valori

182

Definire un tipo in PASCAL

- definizioni di tipo simili a quelle dei linguaggi funzionali
  - espresse prevalentemente in termini di strutture dati primitive
    - array, record, puntatori
  - la ricorsione è realizzata di solito con records e puntatori

183

Definire un tipo in Java

- un insieme di variabili
  - di istanza
    - devono essere dell'oggetto e non della classe
  - private
    - devono essere accessibili solo dai costruttori e dai metodi della classe
- i valori espliciti che si vedono sono solo quelli costruiti dai costruttori
  - più o meno i casi base di una definizione ricorsiva
- gli altri valori sono eventualmente calcolati dai metodi
  - rimane nascosta l'eventuale struttura ricorsiva

184

Usi "corretti" delle classi in Java

- nella definizione di astrazioni procedurali
  - le classi contengono essenzialmente metodi statici
    - eventuali variabili statiche possono servire per avere dati condivisi fra le varie attivazioni dei metodi
      - procedure con stato interno
    - variabili e metodi di istanza (inclusi i costruttori) non dovrebbero esistere, perchè la classe non sarà mai usata per creare oggetti
- nella definizione di astrazioni sui dati
  - le classi contengono essenzialmente metodi di istanza e variabili di istanza private
    - eventuali variabili statiche possono servire (ma è sporco!) per avere informazione condivisa fra oggetti diversi
    - eventuali metodi statici non possono comunque vedere l'oggetto e servono solo a manipolare le variabili statiche

185

I tipi record in Java

- Java non ha un meccanismo primitivo per definire tipi record
  - ma è facilissimo definirli
  - anche se con una deviazione dai discorsi metodologici che abbiamo fatto
    - la rappresentazione non è nascosta (non c'è astrazione!)
    - non ci sono metodi
    - di fatto non c'è specifica separata dall'implementazione

186

Un tipo record

```
class Pair {
 // OVERVIEW: un tipo record
  int coeff;
  int exp;
 // costruttore
  Pair (int c, int n)
     // EFFECTS: inizializza il "record" con i
   // valori di c ed n
  { coeff = c; exp = n;}
}
```

- la rappresentazione non è nascosta
  - dopo aver creato un'istanza si accedono direttamente i "campi del record"
- la visibilità della classe e del costruttore è ristretta al package in cui figura
- non ci sono metodi diversi dal costruttore

Implementazione di `IntSet` 1

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 private Vector els; // la rappresentazione
 // costruttore
  public IntSet ()
     // EFFECTS: inizializza this a vuoto
 {els = new Vector();}
 ...}
```

- un insieme di interi è rappresentato da un `Vector`
  - più adatto dell'`Array`, perché l'insieme ha dimensione variabile
- gli elementi di un `Vector` sono di tipo `Object`
  - non possiamo memorizzarci valori di tipo `int`
  - usiamo oggetti di tipo `Integer`
    - interi visti come oggetti

Implementazione di `IntSet` 2

```
  public void insert (int x)
     // MODIFIES: this
     // EFFECTS: aggiunge x a this
   {Integer y = new Integer(x);
   if (getIndex(y) < 0) els.add(y); }
  private int getIndex (Integer x)
     // EFFECTS: se x occorre in this ritorna la
   // posizione in cui si trova, altrimenti -1
    {for (int i = 0; i < els.size(); i++)
   if (x.equals(els.get(i))) return i;
   return -1; }
```

- non abbiamo occorrenze multiple di elementi
  - si semplifica l'implementazione di `remove`
- il metodo privato ausiliario `getIndex` ritorna un valore speciale e non solleva eccezioni
  - va bene perché è privato
- notare l'uso del metodo `equals` su `Integer`

Implementazione di `IntSet` 3

```
public void remove (int x)
    // MODIFIES: this
    // EFFECTS: toglie x da this
  {int i = getIndex(new Integer(x));
  if (i < 0) return;
  els.set(i, els.lastElement());
  els.remove(els.size() - 1);}
public boolean isIn (int x)
    // EFFECTS: se x appartiene a this ritorna
  // true, altrimenti false
  { return getIndex(new Integer(x)) >= 0; }
```

- nella rimozione, se l'elemento c'è, ci scrivo sopra l'ultimo corrente ed elimino l'ultimo elemento

Implementazione di `IntSet` 4

```
public int size ()
 // EFFECTS: ritorna la cardinalità di this
 {return els.size(); }
public int choose () throws EmptyException
 // EFFECTS: se this è vuoto, solleva
 // EmptyException, altrimenti ritorna un
 // elemento qualunque contenuto in this
 {if (els.size() == 0) throw
    new EmptyException("IntSet.choose");
 return els.lastElement(); }
```

- anche se `lastElement` potesse sollevare un'eccezione, qui non può succedere

Prima implementazione di `Poly` 1

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // coffficienti interi non modificabile
 // esempio: c0 + c1*x + c2*x^2 + ...
 private int[] termini; // la rappresentazione
 private int deg; // la rappresentazione
```

- i polinomi non cambiano la dimensione
  - `Array` invece che `Vector`
  - l'elemento in posizione i contiene il coefficiente del termine che ha esponente i
  - va bene solo per polinomi non sparsi
- per comodità (efficienza) ci teniamo traccia nella rappresentazione del degree del polinomio
  - variabile di tipo `int`

Prima implementazione di `Poly` 2

```
// costruttori
public Poly ()
 // EFFECTS: inizializza this al polinomio 0
 {termini = new int[1]; deg = 0; }
public Poly (int c, int n) throws
    NegativeExponentExc
      // EFFECTS: se n<0 solleva NegativeExponentExc
   // altrimenti inizializza this al polinomio cx^n
 if (n < 0) throw new NegativeExponentExc ("Poly(int,int) constructor");
 if (c == 0)
   {termini = new int[1]; deg = 0; return; }
 termini = new int[n+1];
 for (int i = 0; i < n; i++) termini[i] = 0;
 termini[n] = c; deg = n; }
private Poly (int n)
 {termini = new int[n+1]; deg = n; }
```

- il polinomio vuoto è rappresentato da un array di un elemento contenente 0
- un costruttore privato di comodo

193

Prima implementazione di `Poly` 3

```
public int degree ()
 // EFFECTS: ritorna 0 se this è il polinomio
 // 0, altrimenti il più grande esponente con
 // coefficiente diverso da 0 in this
 {return deg; }
public int coeff (int d)
 // EFFECTS: ritorna il coefficiente del
 // termine in this che ha come esponente d
 {if (d < 0 || d > deg) return 0;
   else return termini[d];}
public Poly minus ()
 // EFFECTS: ritorna -this
 {Poly y = new Poly(deg);
 for (int i = 0; i < deg; i++)
  y.termini[i] = -termini[i];
 return y;}
public Poly sub (Poly q) throws
    NullPointerException
 // EFFECTS: q=null solleva NullPointerException
 // altrimenti ritorna this - q
 {return add(q.minus()); }
```

194

Prima implementazione di `Poly` 4

- più complesse
  - ma solo negli aspetti algoritmici
  - le implementazioni di `add` e `mul`
    - che non mostriamo
- se i polinomi sono sparsi
  - questa implementazione non è efficiente
    - arrays grandi e pieni di 0
  - un'implementazione alternativa in termini di `Vector` i cui elementi sono coppie (coefficiente, esponente)
    - esattamente il record type che abbiamo visto

```
class Pair {
int coeff; int exp;
Pair (int c, int n)
     { coeff = c; exp = n;}}
```

195

Seconda implementazione di `Poly`

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // cofficienti interi non modificabile
 // esempio: c0 + c1*x + c2*x^2 + ...
 private Vector termini; // la rappresentazione
 private int deg; // la rappresentazione
```

- gli oggetti contenuti in `termini` sono `Pair` che rappresentano i termini con coefficiente diverso da 0
- un esempio di operazione

```
public int coeff (int d)
 // EFFECTS: ritorna il coefficiente del
 // termine in this che ha come esponente d
 {for (int i = 0; i < termini.size(); i++)
   {Pair p = (Pair) termini.get(i);
   if (p.exp == d) return p.coeff;}
 return 0;}
```

- notare il casting

196

Metodi addizionali

- esistono vari metodi
  - definiti nella classe `Object`

  che possono essere ereditati

  quando ha senso

  o ridefiniti da qualunque classe
- alcuni esempi
  - equals
  - clone
  - toString

197

`equals`

- in `Object` verifica se due oggetti sono lo stesso oggetto
  - non se i due oggetti hanno lo stesso stato
  - va bene per i tipi modificabili (può essere ereditata)
    - dove lo stato è variabile
  - dovrebbe essere ridefinita per i tipi non modificabili
    - in termini di uguaglianza fra gli stati
- in `Object` c'è anche un metodo `hashCode` che produce, dato un oggetto, un valore da usare come chiave in una tabella Hash
  - stesso valore per oggetti equivalenti (secondo `equals`)
  - se un tipo non modificabile è usato come chiave, deve ridefinire anche `hashCode`

198

clone

- in `Object` genera una copia dell'oggetto
  - nuovo oggetto con lo stesso stato
    - copiando il frame delle variabili istanza
- questa implementazione non è sempre corretta
  - per esempio, in `IntSet` i campi `els` dei due oggetti conterrebbero esattamente lo stesso `Vector`
    - creando una situazione di condivisione (con trasmissione di modifiche) non desiderata
- il metodo viene ereditato solo se l'header della classe contiene la clausola `implements Cloneable`
- se non va bene quella di default si deve reimplementare

toString

- in `Object` genera una stringa contenente il tipo dell'oggetto ed il suo Hash code
- normalmente si vorrebbe ottenere una stringa composta da
  - tipo
  - valori dello stato
- se se ne ha bisogno, va ridefinita sempre

Esempi da provare

- specifica dei tipi utilizzati nello stato della semantica operazionale
  - Frame e Stacks modificabili
  - gli altri non modificabili
  - si supponga che gli identificatori, le liste di parametri ed il codice siano stringhe
  - si usino eccezioni per trattare i casi particolari

Progettazione dettagliata di un Tipo di Dato Astratto: l'ambiente di metodi

L'ambiente di metodi

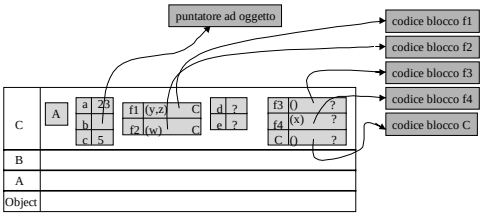
- Ambiente dei metodi

Menv = Id -> Mdescr

Mdescr = Idlist * Blocco * (Loc | Id)

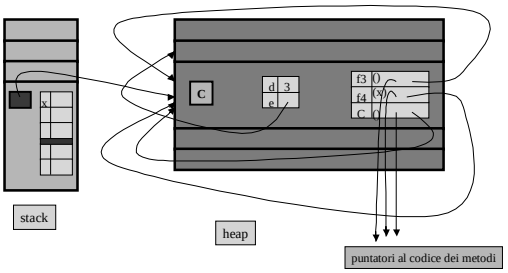
- memptyenv()
- mbind((μ:Menv), (i:Id), (m:Mdescr))
- mdefined ((μ:Menv), (i:Id))
- instantiate((μ:Menv), (l: Loc))
- applicazione

Dove era usato nello stato (1)



- parte statica (classi)

Dove era usato nello stato(2)



- parte dinamica (pila e heap)

L'ambiente di metodi 1

```
public class MethodEnv {
 // OVERVIEW: un MethodEnv è una
 // funzione parziale da identificatori
 // di metodi (stringhe) a descrizioni di
 // metodi. Non è modificabile.

 // costruttore
 public MethodEnv ()
 // EFFECTS: costruisce un nuovo
 // MethodEnv indefinito per tutti
 // gli identificatori
```


L'ambiente di metodi 2

```
public boolean defined (String s)
 // EFFECTS: se this è definita per
 // l'identificatore s ritorna true
 // altrimenti ritorna false
```


L'ambiente di metodi 3

```
public Mdescr apply (String s) throws Undefined
 // EFFECTS: se la funzione this è
 // definita per l'identificatore x
 // restituisce this(s) altrimenti
 // solleva l'eccezione (unchecked)
 // Undefined
```


L'ambiente di metodi 4

```
public MethodEnv bind (String s, Mdescr m)
 // EFFECTS: restituisce una funzione
 // diversa da this solo perché se
 // applicata ad s restituisce m
```


L'ambiente di metodi 5

```
public MethodEnv instantiate (ClassorObject o) throws
   NonObjectException
 // EFFECTS: restituisce una funzione
 // diversa da this solo perché per ogni
 // nome restituisce una descrizione di
 // metodo nuova contenente l'oggetto o
 // se o non e' un oggetto solleva
 // NonObjectException (unchecked)
```

Di che altri tipi abbiamo bisogno?

- le descrizioni di metodo
  - Mdescr = Idlist * Blocco * (Loc | Id)
  - `Mdescr`
- l'identificatore di classe o oggetto
  - (Loc | Id)
  - `ClassorObject`
- i due exception types (unchecked)
  - `NonObjectException`
  - `Undefined`

I tipi eccezione

- i due exception types (unchecked)
  - `NonObjectException`
  - `Undefined`

```
public class NonObjectException extends
  RuntimeException {
// costruttore
public NonObjectException (String s)
{super (); } }
```


I tipi eccezione

- i due exception types (unchecked)
  - `NonObjectException`
  - `Undefined`

```
public class Undefined extends RuntimeException {
// costruttore
public Undefined (String s)
{super (); } }
```


L'identificatore di classe o oggetto 1

- `ClassorObject`
- (Loc | Id)

```
public class ClassorObject {
// un ClassorObject è un oggetto che può
// essere un identificatore di classe
// (stringa), un identificatore di
// oggetto (Loc) o indefinito
```


L'identificatore di classe o oggetto 2

```
...
// 3 costruttori overloaded
public ClassorObject ()
// EFFECTS: costruisce un ClassorObject
// indefinito
public ClassorObject (String x)
// EFFECTS: costruisce un ClassorObject
// che contiene la classe x
public ClassorObject (Loc x)
// EFFECTS: costruisce un ClassorObject
// che contiene l'oggetto x
```


L'identificatore di classe o oggetto 3

```
public boolean isClass ()
// EFFECTS: se this è un identificatore
// di classe restituisce true altrimenti
// restituisce false
public boolean isUndefined ()
// EFFECTS: se this è indefinito
// restituisce true altrimenti
// restituisce false
public boolean isObject ()
// EFFECTS: se this è un oggetto
// restituisce true altrimenti
// restituisce false  }
```

Di che altri tipi abbiamo bisogno?

- le locazioni (identificatori di oggetti)
  - `Loc`

```
public class Loc {
 // Loc contiene un intero che permette
 // di indirizzare una heap: ha uno stato
 // interno che permette di generare
 // valori sempre diversi
```


Le locazioni

```
// costruttore
 public Loc ()
 // EFFECTS: costruisce una nuova Loc
 // MODIFIES: lo stato interno
// metodi
 public int key ()
 // restituisce il valore contenuto in
 // this   }
```


Le descrizioni di metodo

- le descrizioni di metodo
  - Mdescr = Idlist * Blocco * (Loc | Id)
  - `Mdescr`
- lo realizziamo direttamente con un record type
  - Idlist e Blocco sono stringhe
  - il terzo campo è un `ClassorObject`

Le descrizioni di metodo 1

```
public class Mdescr {
// OVERVIEW: un Mdescr è un record type
// che contiene tutta l'informazione
// associata ad un metodo
 String parameters;
 String body;
 ClassorObject belongsto;
```

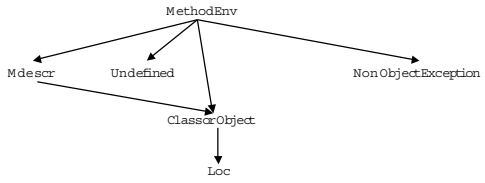

Le descrizioni di metodo 2

```
// costruttore
 public Mdescr (String x, String y, ClassorObject z)
 // EFFECTS: costruisce un record
 // contenente i tre argomenti
 {parameters = x; body = y;
  belongsto = z;}  }
```


Lo stato della progettazione

- classi specificate e implementate (eccezioni e record types)
  - `Mdescr`
  - `NonObjectException`
  - `Undefined`
- classi specificate
  - `ClassorObject`
  - `Loc`
  - `MethodEnv`

La relazione “chi usa chi” a livello di specifica



Come continuare

- le classi da implementare
  - `ClassorObject`
  - `Loc`
  - `MethodEnv`
- possono essere implementate in un ordine qualunque
  - `Loc`
  - `ClassorObject`
  - `MethodEnv`

Implementazione di `Loc` 1

```
public class Loc {
 // Loc contiene un intero che permette
 // di indirizzare una heap: ha uno stato
 // interno che permette di generare
 // valori sempre diversi
 private static int stato = 0;
 private int chiave;
```

- `chiave` è il valore “contenuto”
- `stato` è lo stato interno (static)

Implementazione di `Loc` 2

```
// costruttore
 public Loc ()
 // EFFECTS: costruisce una nuova Loc
 // MODIFIES: lo stato interno
 {
   chiave = stato;
   stato = stato + 1;
   return; }
```


Implementazione di `Loc` 3

```
// metodi
 public int key ()
 // restituisce il valore contenuto in
 // this
 { return chiave; } }
```


Implementazione di `ClassorObject` 1

```
public class ClassorObject {
 // un ClassorObject è un oggetto che può
 // essere un identificatore di classe
 // (stringa), un identificatore di
 // oggetto (Loc) o indefinito
private Object v;
```

- lo stato è una variabile di tipo `Object`

Implementazione di ClassorObject 2

```
public ClassorObject ()
 // EFFECTS: costruisce un ClassorObject
 // indefinito
{ }
 public ClassorObject (String x)
 // EFFECTS: costruisce un ClassorObject
 // che contiene la classe x
{ v = x; }
 public ClassorObject (Loc x)
 // EFFECTS: costruisce un ClassorObject
 // che contiene l'oggetto x
{ v = x; }
```


Implementazione di ClassorObject 3

```
public boolean isClass ()
 // EFFECTS: se this è un identificatore
 // di classe restituisce true altrimenti
 // restituisce false
{if (isUndefined()) return false;
String s;
try {s = (String) v;}
catch (Exception e) {return false; }
return true; }
```

- utilizziamo le eccezioni (mascherate) per fare un'analisi di casi sul tipo

Implementazione di ClassorObject 4

```
public boolean isUndefined ()
 // EFFECTS: se this è indefinito
 // restituisce true altrimenti
 // restituisce false
{ if (v == null) return true;
  else return false; }
```


Implementazione di ClassorObject 5

```
public boolean isObject ()
 // EFFECTS: se this è un oggetto
 // restituisce true altrimenti
 // restituisce false
{if (isUndefined()) return false;
Loc l;
try {l = (Loc) v;}
catch (Exception e) {return false;}
return true;} }
```


Implementazione di MethodEnv

```
public class MethodEnv {
 // OVERVIEW: un MethodEnv è una
 // funzione parziale da identificatori
 // di metodi (stringhe) a descrizioni di
 // metodi. Non è modificabile.
```

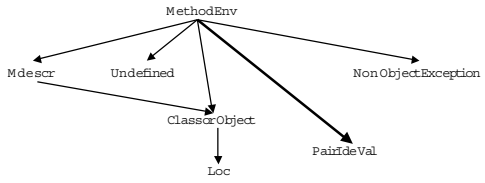
- decidiamo di rappresentare le funzioni con arrays i cui elementi sono coppie (String, Mdescr)
  - specifichiamo quindi per prima cosa un tipo record PairIdeVal
    - un po' più generale di quello che ci serve

Specifica ed implementazione di PairIdeVal

```
public class PairIdeVal {
 // OVERVIEW: un PairIdeVal è un record
 // type che contiene una associazione
 // tra nome e valore
 String nome;
 Object valore;
 // costruttore
 public PairIdeVal (String x, Object z)
  // EFFECTS: costruisce un record
  // contenente i due argomenti
  {nome = x; valore = z;} }
```

- un po' più generale di quello che ci serve
  - i valori sono Object e non Mdescr

La relazione “chi usa chi” a livello di specifica e di implementazione



Implementazione di MethodEnv 1

```
public class MethodEnv {
 // OVERVIEW: un MethodEnv è una
 // funzione parziale da identificatori
 // di metodi (stringhe) a descrizioni di
 // metodi. Non è modificabile.

  private PairIdeVal[] associazioni;
```


Implementazione di MethodEnv 2

```
// costruttore pubblico
public MethodEnv ()
 // EFFECTS: costruisce un nuovo
 // MethodEnv indefinito per tutti
 // gli identificatori
 {associazioni = new PairIdeVal[0];}
```


Implementazione di MethodEnv 3

```
// altri costruttori privati
private MethodEnv (PairIdeVal[] m)
 // EFFECTS: costruisce una nuova
 // funzione la cui rappresentazione
 // è m
 {associazioni =
    new PairIdeVal[m.length];
 if (m.length == 0) return ;
 for (int i = 0; i < m.length; i++) {
   PairIdeVal p = m[i];
   associazioni[i] = p;}   }
```


Implementazione di MethodEnv 4

```
// altri costruttori privati
private MethodEnv (int n)
 // EFFECTS: costruisce una nuova
 // funzione la cui rappresentazione
 // è un array vuoto lungo n
 {associazioni = new PairIdeVal[n];}
```


Implementazione di MethodEnv 5

```
// altri costruttori privati
private MethodEnv (PairIdeVal[] m, int n)
 // REQUIRES: n = m.length+1
 // EFFECTS: costruisce una nuova
 // funzione la cui rappresentazione
 // è un array uguale ad m più una
 // posizione vuota
  {associazioni = new PairIdeVal[n];
 if (m.length == 0) return ;
 for (int i = 0; i < m.length; i++) {
   PairIdeVal p = m[i];
   associazioni[i] = p;} }
```

Implementazione di MethodEnv 6

```
public boolean defined (String s)
 // EFFECTS: se this è definita per
 //l'identificatore s ritorna true
 // altrimenti ritorna false
{if (associazioni.length == 0)
    return false;
for (int i = 0; i < associazioni.length; i++) {
  PairIdeVal p = associazioni[i];
  if (s.equals(p.nome)) return true;}
return false;}
```


Implementazione di MethodEnv 7

```
public Mdescr apply (String s) throws Undefined
 // EFFECTS: se la funzione this è
 // definita per l'identificatore x
 // restituisce this(s) altrimenti
 // solleva l'eccezione (unchecked)
 // Undefined
{for (int i = 0; i < associazioni.length; i++) {
  PairIdeVal p = associazioni[i];
  if (s.equals(p.nome))
      return (Mdescr) p.valore;}
throw new Undefined("Methodenv.apply");}
```


Implementazione di MethodEnv 8

```
public MethodEnv bind (String s, Mdescr m)
 // EFFECTS: restituisce una funzione diversa da this
 // solo perché se applicata ad s restituisce m
{MethodEnv nuovo;
PairIdeVal newass = new PairIdeVal(s,m);
if (defined(s))
 {nuovo = new MethodEnv(associazioni);
 for (int i = 0; i < nuovo.associazioni.length; i++) {PairIdeVal p =
    nuovo.associazioni[i];
   if (s.equals(p.nome))
     nuovo.associazioni[i] = newass;}
   return nuovo;}
else {if (associazioni.length == 0)
  {nuovo = new MethodEnv(1);}
  else {nuovo = new MethodEnv(associazioni,
    associazioni.length + 1);}
nuovo.associazioni[associazioni.length] = newass;
return nuovo;}
```


Implementazione di MethodEnv 9

```
public MethodEnv instantiate (ClassorObject o) throws
    NonObjectException
 // EFFECTS: restituisce una funzione diversa da this
 // solo perché per ogni nome restituisce una
 // descrizione di metodo nuova contenente l'oggetto o
 // se o non e' un oggetto solleva NonObjectException
 // (unchecked)
{MethodEnv nuovo = new MethodEnv(associazioni);
if (o.isObject()) {
  for (int i = 0; i < associazioni.length; i++) {
    Mdescr vecchia = (Mdescr) associazioni[i].valore;
    Mdescr nuova = new
      Mdescr(vecchia.parameters,vecchia.body,o);
    PairIdeVal newass = new
      PairIdeVal(associazioni[i].nome,nuova);
    nuovo.associazioni[i] = newass;}
  return nuovo;}
else {throw new  NonObjectException("MethodEnv.instantiate");}}
```


Quello che dovete fare

- seguire la stessa metodologia per la specifica e l'implementazione del tipo di dato astratto (modificabile)

Frame = Id -> Val

Val = (Bool | Int | Loc)

- newframe()
- copy((φ:Frame))
- bind((φ:Frame) , (i:Id), (v:Val))
- update((φ:Frame) , (i:Id), (v:Val))
- defined((φ:Frame), (i:Id))

- cercando di riutilizzare al massimo i tipi definiti per MethodEnv

Astrazioni sui dati : Ragionare sui Tipi di Dato Astratti

Ragionare sui Tipi di Dato Astratti

- proprietà dell'astrazione
  - modificabilità
  - categorie di operazioni
  - dimostrare proprietà dell'astrazione
- dimostrare proprietà dell'implementazione
  - funzione di astrazione
  - invariante di rappresentazione
  - dimostrazione mediante induzione sui dati

Modificabilità 1

- i tipi non modificabili sono più sicuri
  - la condivisione di sottostrutture non crea problemi
- i tipi non modificabili sono spesso più inefficienti
  - la necessità di costruire spesso copie di oggetti può complicare la vita al garbage collector
- la scelta dovrebbe comunque tener conto delle caratteristiche dei concetti matematici o degli oggetti del mondo reale modellati dal tipo
  - gli interi non sono modificabili
  - gli insiemi sono modificabili
  - i conti correnti sono modificabili
  - ....

Modificabilità 2

- un tipo non modificabile può essere implementato utilizzando strutture modificabili
  - arrays, vectors, tipi record, tipi astratti modificabili

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // coffcienti interi non modificabile
 // esempio: c0 + c1*x + c2*x2 + ...
 private int[] termini; // la rappresentazione
 private int deg; // la rappresentazione
```

- attenzione comunque agli effetti laterali "nascosti"
  - un metodo può restituire la rappresentazione modificabile (esporre la *rep*)
  - un tipo non modificabile può contenere un tipo modificabile
    - che può essere restituito da un metodo (e poi modificato)

Categorie di operazioni 1

- creatori
  - creano oggetti del loro tipo "dal nulla"
    - sicuramente costruttori
    - non tutti i costruttori sono creatori
      - possono avere come argomenti oggetti del loro tipo

```
public IntSet ()
   // EFFECTS: inizializza this a vuoto
```

- produttori
  - prendono come argomenti oggetti del loro tipo e ne costruiscono altri
    - possono essere costruttori o metodi

```
public Poly sub (Poly q) throws
    NullPointerException
 // EFFECTS: q=null solleva NullPointerException
 // altrimenti ritorna this - q
```


Categorie di operazioni 2

- modificatori
  - modificano gli oggetti del loro tipo

```
public void insert (int x)
     // MODIFIES: this
     // EFFECTS: aggiunge x a this
```

- osservatori
  - prendono oggetti del loro tipo e restituiscono valori di altri tipi
    - per ottenere informazioni sugli oggetti

```
public boolean isIn (int x)
    // EFFECTS: se x appartiene a this ritorna
  // true, altrimenti false
public int coeff (int d)
 // EFFECTS: ritorna il coefficiente del
 // termine in this che ha come esponente d
```


Categorie di operazioni 3

- i modificatori
  - modificano gli oggetti del loro tipo
  - per tipi modificabili
- i produttori
  - prendono come argomenti oggetti del loro tipo e ne costruiscono altri
  - per tipi non modificabili
- svolgono funzioni simili

Categorie di operazioni 4

- quali e quante operazioni in una astrazione?
  - almeno un creatore
  - qualche produttore, se il tipo non è modificabile
  - qualche modificatore, se il tipo è modificabile
    - attraverso creatori e produttori (o modificatori) dovremmo essere in grado di generare tutti i valori astratti
  - qualche osservatore
- certe operazioni possono essere definite
  - nella classe (come "primitive")
  - fuori della classe (nuovi metodi)
- la scelta bilanciando efficienza dell'implementazione dei metodi e complessità della classe

Dimostrare proprietà dell'astrazione

- è spesso utile poter dimostrare proprietà delle astrazioni
  - anche per quelle procedurali
  - ma più interessante per le astrazioni sui dati
- per dimostrare la proprietà dobbiamo utilizzare le specifiche
- vediamo un esempio

Dimostrariamo una proprietà di IntSet 1

```
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)
    // MODIFIES: this
    // EFFECTS: aggiunge x a this
public void remove (int x)
    // MODIFIES: this
    // EFFECTS: toglie x da this
```

- vogliamo dimostrare che per ogni IntSet la sua size è >= 0
- basta convincerci che questo è vero per i costruttori ed i modificatori

Dimostrariamo una proprietà di IntSet 2

- per ogni IntSet la sua size è >= 0
- per il costruttore

```
public IntSet ()
    // EFFECTS: inizializza this a vuoto
```

- l'insieme vuoto ha cardinalità 0

Dimostrariamo una proprietà di IntSet 3

- per ogni IntSet la sua size è >= 0
- per ogni modificatore

```
public void insert (int x)
    // MODIFIES: this
    // EFFECTS: aggiunge x a this
```

- se la proprietà vale prima dell'inserimento, vale anche dopo perché l'inserimento può solo incrementare la cardinalità

Dimostrariamo una proprietà di IntSet 4

- per ogni IntSet la sua size è >= 0
- per ogni modificatore

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

- se la proprietà vale prima della rimozione, vale anche dopo perché la rimozione può ridurre la cardinalità solo se l'elemento era contenuto al momento della chiamata

Correttezza dell'implementazione

- se vogliamo dimostrare che le implementazioni dei metodi soddisfano le rispettive specifiche
  - non possiamo utilizzare la metodologia appena vista
- l'implementazione utilizza la rappresentazione
  - nel caso di `IntSet`
  - `private Vector els;`
- le specifiche esprimono proprietà dell'astrazione
  - nel caso di `IntSet`

```
public boolean isIn (int x)
   // EFFECTS: se x appartiene a this ritorna
  //true, altrimenti false
```

- è necessario mettere in relazione tra loro i due "insiemi di valori"

La funzione di astrazione 1

- la funzione di astrazione cattura l'intenzione del progettista nello scegliere una particolare rappresentazione
- la funzione di astrazione

  α : C --> A

  porta da uno stato concreto
  - lo stato di un oggetto della classe `C`

  a uno stato astratto
  - lo stato dell'oggetto astratto

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 private Vector els; //la rappresentazione
```

- α porta vettori in insiemi

La funzione di astrazione 2

- la funzione di astrazione è generalmente una funzione molti-a-uno

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 private Vector els; //la rappresentazione
```

  - piú stati concreti (vettori di interi) vengono portati nello stesso stato astratto (insieme)
  - α([1,2]) = {1,2}
  - α([2,1]) = {1,2}
- la funzione di astrazione deve sempre essere definita ed inserita come commento all'implementazione
  - perché è una parte importante delle decisioni relative all'implementazione

La funzione di astrazione 3

- la funzione di astrazione deve sempre essere definita ed inserita come commento all'implementazione
  - perché è una parte importante delle decisioni relative all'implementazione
- il problema è che non abbiamo una rappresentazione esplicita dei valori astratti
- diamo (nella OVERVIEW) la descrizione di un tipico stato astratto
- esempi
  - nella definizione della funzione di astrazione, useremo la notazione del linguaggio di programmazione

La funzione di astrazione di `IntSet`

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 // un tipico IntSet è {x1, ..., xn}
 private Vector els; //la rappresentazione
 //la funzione di astrazione
 // α(c) = { c.els.get(i).intValue() |
           0 <= i < c.els.size() }
```


La funzione di astrazione di `Poly`

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // coefficienti interi non modificabile
 // un tipico Poly: c0 + c1*x + c2*x^2 + ...
 private int[] termini; //la rappresentazione
 private int deg; //la rappresentazione
 //la funzione di astrazione
 // α(c) = c0 + c1*x + c2*x^2 + ... tale che
 // ci = c.termini[i] se 0 <= i < c.termini.length
 //  = 0 altrimenti
```

- notare che il valore di `deg` non ha nessuna influenza sulla funzione di astrazione
  - è una informazione derivabile dall'array `termini` che utilizziamo nello stato concreto per questioni di efficienza

La funzione di astrazione può essere implementata

- la funzione di astrazione deve sempre essere definita ed inserita come commento all'implementazione
- non avendo una rappresentazione esplicita dei valori astratti, possiamo rappresentarli come stringhe
- a questo punto, possiamo implementare la funzione di astrazione, che è esattamente il metodo `toString`
  - utile per stampare valori astratti

```
// α(c) = { c.els.get(i).intValue() |
            0 <= i < c.els.size() }

// α(c) = c0 + c1*x + c2*x^2 + ... tale che
// ci = c.termini[i] se 0 <= i < c.termini.size()
//    = 0 altrimenti
```


`toString` per `IntSet`

```
// α(c) = { c.els.get(i).intValue() |
            0 <= i < c.els.size() }

public String toString ()
 {String s = "{";
 for (int i = 0; i < els.size() - 1; i++) {
   s = s + els.get(i).toString() + ",";}
 if (els.size() > 0) {
   s = s + els.get(els.size() - 1).toString();}
 s = s + "}";
 return (s);}
```


Verso l'invariante di rappresentazione

- non tutti gli stati concreti "rappresentano" correttamente uno stato astratto

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 // un tipico IntSet è {x1, ..., xn}
 private Vector els; // la rappresentazione
 // la funzione di astrazione
 // α(c) = { c.els.get(i).intValue() |
            0 <= i < c.els.size() }
```

- il vettore `els` potrebbe contenere più occorrenze dello stesso elemento
  - questo sarebbe coerente con la funzione di astrazione
  - ma non rispecchierebbe la nostra scelta di progetto
    - riflessa nell'implementazione dei metodi

L'invariante di rappresentazione

- l'invariante di rappresentazione (rep invariant) è un predicato

  `I : C --> boolean`

  che è vero per gli stati concreti che sono rappresentazioni legittime di uno stato astratto
- l'invariante di rappresentazione, insieme alla funzione di astrazione, riflette le scelte relative alla rappresentazione
  - deve essere inserito nella documentazione della implementazione come commento
- la funzione di astrazione è definita solo per stati concreti che soddisfano l'invariante

L'invariante di rappresentazione di `IntSet`

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 // un tipico IntSet è {x1, ..., xn}
 private Vector els; // la rappresentazione
 // la funzione di astrazione:
 // α(c) = { c.els.get(i).intValue() |
            0 <= i < c.els.size() }
 // l'invariante di rappresentazione:
 // I(c) = c.els != null e
 // per ogni intero i, c.els.get(i) è un Integer
 // e  per tutti gli interi i,j, tali che
 //       0 <= i < j < c. els.size(),
 // c.els.get(i).intValue() !=
 //       c.els.get(j).intValue()
```

- il vettore non deve essere `null`
- gli elementi del vettore devono essere `Integer`
  - assunti soddisfatti in α
- tutti gli elementi sono distinti

Una diversa implementazione per `IntSet` 1

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 // un tipico IntSet è {x1, ..., xn}
 private boolean[100] els;
 private Vector altriels;
 private int dim;
```

- l'inserimento di un elemento `n` compreso tra 0 e 99 viene realizzato mettendo a true `els[n]`
- gli elementi maggiori di 99 sono inseriti nel vettore `altriels` gestito come nell'implementazione precedente
- `dim` contiene esplicitamente la cardinalità
  - che sarebbe complessa da calcolare a partire da `els`
- implementazione sensata solo se la maggior parte degli elementi sono compresi nell'intervallo 0-99

Una diversa implementazione per IntSet 2

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 // un tipico IntSet è {x1, ..., xn}
 private boolean[100] els;
 private Vector altriels;
 private int dim;
 // la funzione di astrazione:
 // α(c) = { c.altriels.get(i).intValue() |
 //    0 <= i < c.altriels.size() } +
 // {j | 0 <= j < 100 e c.els[j] }
```


Una diversa implementazione per IntSet 3

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 // un tipico IntSet è {x1, ..., xn}
 private boolean[100] els;
 private Vector altriels;
 private int dim;
 // l'invariante di rappresentazione:
 // I(c) = c.els != null e
 // c.altriels != null e
 // els.length = 100 e
 // per ogni intero i,
 //   c.altriels.get(i) è un Integer,
 //   c.altriels.get(i).intValue() non appartiene
 //        all'intervallo 0-99, e
 // per tutti gli interi i,j, tali che
 //        0 <= i < j < c.altriels.size(),
 //    c.altriels.get(i).intValue() !=
 //     c.altriels.get(j).intValue() e
 // c.dim = c.altriels.size() + conta(c.els, 0)
```


Una funzione ausiliaria nel rep invariant

```
where
conta(a,i) = if (i >= a.length) return 0;
   else if (a[i]) return (1 + conta(a, i-1))
     else return (conta(a, i-1))
```


L'invariante di rappresentazione di Poly

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // cofficienti interi non modificabile
 // un tipico Poly: c0 + c1*x + c2*x^2 + ...
 private int[] termini; // la rappresentazione
 private int deg; // la rappresentazione
 // la funzione di astrazione
 // α(c) = c0 + c1*x + c2*x^2 + ... tale che
 // ci = c.termini[i] se 0 <= i < c.termini.size()
 //    = 0 altrimenti
 // l'invariante di rappresentazione:
 // I(c) = c.termini != null e
 //  c.termini.length >= 1 e
 //  c.deg = c.termini.length-1 e
 //  c.deg > 0 ==> c.termini[deg] != 0
```


L'invariante di rappresentazione può essere implementato 1

- il metodo `repOk` che verifica l'invariante dovrebbe essere fornito da ogni astrazione sui dati
  - pubblico perché deve poter essere chiamato da fuori della sua classe
- ha sempre la seguente specifica

```
public boolean repOk ()
 // EFFECTS: ritorna true se il rep invariant
 // vale per this, altrimenti ritorna false
```


`repOK`

- può essere usato da programmi di test per verificare se una implementazione preserva l'invariante
- può essere usato dentro le implementazioni di costruttori e metodi
  - creatori, modificatori e produttori dovrebbero chiamarlo prima di ritornare per assicurarsi che per l'oggetto costruito o modificato vale l'invariante
    - per esempio, dovrebbero chiamarlo `insert` e `remove` di `IntSet`, `add`, `mul`, `minus` di `Poly`
- se l'invariante non vale si dovrebbe sollevare `FailureException`

repOK per Poly

```
public class Poly {
 private int[] termini; //la rappresentazione
 private int deg; //la rappresentazione
 // I(c) = c.termini != null e
 // c.termini.length >= 1 e
 // c.deg = c.termini.length-1 e
 // c.deg > 0 ==> c.termini[deg] != 0


public boolean repOk() {
if (termini == null || deg != termini.length - 1 ||termini.length == 0)
    return false;
if (deg == 0) return true;
return termini[deg] != 0; }
```


repOK per IntSet

```
public class IntSet {
 private Vector els; //la rappresentazione
 // I(c) = c.els != null e
 // per ogni intero i, c.els.get(i) è un Integer
 // e per tutti gli interi i,j, tali che
 //       0 <= i < j < c. els.size(),
 // c.els.get(i).intValue() !=
 //       c.els.get(j).intValue()


public boolean repOk() {
    if (els == null) return false;
    for (int i = 0; i < els.size(); i++)  {
          Object x = els.get(i);
          if (! (x instanceof Integer)) return false;
          for (int j = i + 1; j < els.size(); j++)
          if (x.equals (els.get(j))) return false; }
    return true; }
```


Correttezza di una implementazione

- invece di "eseguire" `repOk` (controllo dinamico), possiamo dimostrare formalmente che, ogniqualvolta un oggetto del nuovo tipo è manipolato all'esterno della classe, esso soddisfa l'invariante
  - induzione sul tipo di dato
- dobbiamo poi dimostrare, per ogni metodo, che l'implementazione soddisfa la specifica
  - usando la funzione di rappresentazione

Soddisfacimento del rep invariant

- per prima cosa, dimostriamo che l'invariante vale per gli oggetti restituiti dai costruttori
- in modo induttivo, dimostriamo che vale per tutti i metodi (produttori e modificatori)
  - assumiamo che l'invariante valga per `this` e per tutti gli argomenti del tipo
  - dimostriamo che vale quando il metodo ritorna
    - per `this`
    - per tutti gli argomenti del tipo
    - per gli oggetti del tipo ritornati
- induzione sul numero di invocazioni di metodi usati per produrre il valore corrente dell'oggetto
  - la base dell'induzione riguarda i costruttori

Correttezza di IntSet 1

```
public class IntSet {
 private Vector els; //la rappresentazione
 // I(c) = c.els != null e
 // per ogni intero i, c.els.get(i) è un Integer
 // e per tutti gli interi i,j, tali che
 //       0 <= i < j < c. els.size(),
 // c.els.get(i).intValue() !=
 //       c.els.get(j).intValue()

 public IntSet ()
 {els = new Vector();}
```

- il costruttore soddisfa l'invariante perché restituisce un `Vector` vuoto

Correttezza di IntSet 2

```
public class IntSet {
 private Vector els; //la rappresentazione
 // I(c) = c.els != null e
 // per ogni intero i, c.els.get(i) è un Integer
 // e per tutti gli interi i,j, tali che
 //       0 <= i < j < c. els.size(),
 // c.els.get(i).intValue() !=
 //       c.els.get(j).intValue()
 public void insert (int x)
   {Integer y = new Integer(x);
   if (getIndex(y) < 0) els.add(y); }
 private int getIndex (Integer x)
     // EFFECTS: se x occorre in this ritorna la
   // posizione in cui si trova, altrimenti -1
```

- il metodo `insert` soddisfa l'invariante perché aggiunge `x` a `this` solo se `x` non è già in `this`

Correttezza di IntSet 3

```
public class IntSet {
 private Vector els; // la rappresentazione
 // I(c) = c.els != null e
 // per ogni intero i, c.els.get(i) è un Integer
 // e per tutti gli interi i,j, tali che
 //       0 <= i < j < c. els.size(),
 // c.els.get(i).intValue() !=
 //       c.els.get(j).intValue()
public void remove (int x)
  {int i = getIndex(new Integer(x));
  if (i < 0) return;
  els.set(i, els.lastElement());
  els.remove(els.size() - 1);}
```

- il metodo `remove` soddisfa l'invariante perché rimuove `x` da `this` solo se `x` è in `this`

Correttezza di Poly 1

```
public class Poly {
 private int[] termini; // la rappresentazione
 private int deg; // la rappresentazione
 // I(c) = c.termini != null e
 //  c.termini.length >= 1 e
 //  c.deg = c.termini.length-1 e
 //  c.deg > 0 ==> c.termini[deg] !=  0
public Poly ()
  {termini = new int[1]; deg = 0; }
```

- il primo costruttore soddisfa l'invariante perché restituisce un `Array` di un elemento e `deg = 0`

Correttezza di Poly 2

```
public class Poly {
 private int[] termini; // la rappresentazione
 private int deg; // la rappresentazione
 // I(c) = c.termini != null e
 //  c.termini.length >= 1 e
 //  c.deg = c.termini.length-1 e
 //  c.deg > 0 ==> c.termini[deg] !=  0
public Poly (int c, int n) throws NegativeExponentExc
 if (n < 0) throw new NegativeExponentExc ("Poly(int,int) constructor");
 if (c == 0)
   {termini = new int[1]; deg = 0; return; }
 termini = new int[n+1];
 for (int i = 0; i < n; i++) termini[i] = 0;
 termini[n] = c; deg = n; }
```

- il secondo costruttore soddisfa l'invariante perché testa esplicitamente il caso `c=0`

Correttezza di Poly 3

```
public class Poly {
 private int[] termini; // la rappresentazione
 private int deg; // la rappresentazione
 // I(c) = c.termini != null e
 //  c.termini.length >= 1 e
 //  c.deg = c.termini.length-1 e
 //  c.deg > 0 ==> c.termini[deg] !=  0
public Poly sub (Poly q) throws
    NullPointerException
  {return add(q.minus()); }
```

- il metodo `sub` soddisfa l'invariante perché
  - lo soddisfano `q` e `this`
  - lo soddisfano `add` e `minus`

Le implementazioni dei metodi soddisfano la specifica

- si ragiona usando la funzione di astrazione
  - un metodo alla volta
- ciò è possibile solo perché abbiamo già dimostrato che il rep invariant è soddisfatto da tutte le operazioni
  - il rep invariant cattura le assunzioni comuni fra le varie operazioni
  - permette di trattarle separatamente

Correttezza di IntSet 1

```
public class IntSet {
 private Vector els; // la rappresentazione
 // la funzione di astrazione
 // α(c) = { c.els.get(i).intValue() |
          0 <= i < c.els.size() }
public IntSet ()
    // EFFECTS: inizializza this a vuoto
 {els = new Vector();}
```

- l'astrazione di un vettore vuoto è proprio l'insieme vuoto

Correttezza di IntSet 2

```
public class IntSet {
 private Vector els; // la rappresentazione
 // la funzione di astrazione
 // α(c) = { c.els.get(i).intValue() |
           0 <= i < c.els.size() }
public int size ()
 // EFFECTS: ritorna la cardinalità di this
 {return els.size(); }
```

- il numero di elementi del vettore è la cardinalità dell'insieme perché
  - la funzione di astrazione mappa gli elementi del vettore in quelli dell'insieme
  - il rep invariant garantisce che non ci sono elementi duplicati in els
    - senza dover andare a guardare come è fatta insert

Correttezza di IntSet 3

```
public class IntSet {
 private Vector els; // la rappresentazione
 // la funzione di astrazione
 // α(c) = { c.els.get(i).intValue() |
           0 <= i < c.els.size() }
public void remove (int x)
    // MODIFIES: this
    // EFFECTS: toglie x da this
 {int i = getIndex(new Integer(x));
 if (i < 0) return;
 els.set(i, els.lastElement());
 els.remove(els.size() - 1);}
```

- se x non occorre nel vettore non fa niente
  - corretto perché in base alla funzione di astrazione x non appartiene all'insieme
- se x occorre nel vettore lo rimuove
  - e quindi in base alla funzione di astrazione x non appartiene all'insieme modificato

Astrazione sul controllo: gli iteratori

Gli iteratori

- perché vogliamo iterarare "in modo astratto"
- iteratori e generatori in Java
  - specifica
  - utilizzazione
  - implementazione
  - rep invariant e funzione di astrazione
  - un esempio

Perché vogliamo iterarare "in modo astratto"

- problema: iterare su tipi di dato arbitrari
- esempio: calcolare la somma di tutti gli elementi di un IntSet

```
public static int setSum (IntSet s) throws NullPointerException
    // EFFECTS: se s è null solleva
  // NullPointerException altrimenti
    //ritorna la somma degli elementi di s
```


Soluzione insoddisfacente 1

```
public static int setSum (IntSet s) throws NullPointerException {
  // EFFECTS: se s è null solleva
  // NullPointerException altrimenti
    //ritorna la somma degli elementi di s
    int[] a = new int [s.size()];
    int sum = 0;
    for (int i = 0; i < a.length; i++)
     {a[i] = s.choose();
      sum = sum + a[i];
      s.remove(a[i]); }
  //risistema s
    for (int i = 0; i < a.length; i++)
    s.insert(a[i]);
    return sum;}
```

- ad ogni iterazione vengono chiamate due operazioni (choose e remove)
- gli elementi rimossi vanno reinseriti

Soluzione insoddisfacente 2

- potremmo realizzare `setSum` come metodo della classe `IntSet`
  - in modo più efficiente
    - accedendo la rappresentazione
  - non è direttamente collegata al concetto di `IntSet`
  - quante altre operazioni simili dovremmo mettere in `IntSet`?
    - trovare il massimo elemento.....

Soluzione insoddisfacente 3

```
public int[] members ()
 // EFFECTS: restituisce un array contenente gli
 // elementi di this, ciascuno esattamente una volta,
 // in un ordine arbitrario
public static int setSum (IntSet s) {
    int[] a = s.members();
    int sum = 0;
    for (int i = 0; i < a.length; i++) sum = sum + a[i];
    return sum;}
```

- inefficiente
  - due strutture dati
  - non sempre vogliamo generare tutti gli elementi della collezione
    - massimo elemento

Altre soluzioni insoddisfacenti

- dotiamo `IntSet` di una operazione che ritorna la rappresentazione
  - distruggiamo l'astrazione
- ridefiniamo l'astrazione `IntSet` in modo da avere una nozione di indiciamento
  - è un'astrazione molto più complessa e non direttamente legata alla nozione di insieme

Di cosa abbiamo bisogno?

- un meccanismo generale di iterazione
  - facile da usare
  - efficiente
  - che preservi l'astrazione
- per ogni i prodotto da g esegui a su i
- g è un generatore che produce in modo incrementale (uno alla volta) tutti gli elementi i della collezione corrispondente all'oggetto
- l'azione a da compiere sugli elementi è separata dalla generazione degli elementi stessi

Iteratori e ordine superiore

- per ogni i prodotto da g esegui a su i
- cose di questo genere si realizzano molto facilmente con la normale astrazione procedurale in quei linguaggi (tipicamente funzionali) in cui le procedure sono "cittadini di prima classe", cioè valori come tutti gli altri
  - possono essere passate come parametri ad altre procedure
  - il generatore è una procedura che ha come parametro la procedura che codifica l'azione da eseguire sugli elementi della collezione

Iteratori in Java

- per ogni i prodotto da g esegui a su i
- i generatori sono oggetti di tipo `Iterator`
  - il tipo `Iterator` è definito dalla seguente interfaccia Java (`java.util`package)

```
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)
}
```

  - possiamo definire metodi che restituiscono generatori

Come si usano i generatori 1

- per ogni i prodotto da g esegui a su i
- il metodo primesLT100

```
public static Iterator primesLT100 ()
// EFFECTS: restituisce un generatore,
// che genera incrementalmente tutti i
// numeri primi (Integer) minori di 100
```

- può essere utilizzato per realizzare un'iterazione astratta

```
// ciclo controllato da hasnext
Iterator g = primesLT100 ();
while (g.hasNext())
    {int x = (Integer) g.next()).intValue();
 // usa x }
```


Come si usano i generatori 2

```
public static Iterator primesLT100 ()
// EFFECTS: restituisce un generatore,
// che genera incrementalmente tutti i
// numeri primi (Integer) minori di 100
```

- può essere utilizzato per realizzare un'iterazione astratta

```
// ciclo controllato da exception
Iterator g = primesLT100();
try {while (true) {int x =
    ((Integer) g.next()).intValue();
   // uso di x
   }
catch (NoSuchElementException e) { };
```


Specifica dei metodi che restituiscono generatori

- spesso chiamati *iteratori*
  - da non confondere con il tipo Iterator che restituiscono
- possono essere procedure stand alone
  - come primesLT100
- più interessante quando sono metodi di una classe che definisce una astrazione sui dati
  - vediamo degli esempi su IntSet e Poly

Specifica di un iteratore per Poly

```
public class Poly {
// come prima più
 public Iterator terms ()
 // EFFECTS: ritorna un generatore che produrrà gli
 // esponenti dei termini diversi da 0 in this (come
 // Integers) fino al grado del polinomio, in ordine
 // crescente
                                    }
```

- un tipo di dato può avere anche più iteratori

Specifica di un iteratore per IntSet

```
public class IntSet {
// come prima più
 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                    }
```

- la clausola REQUIRES impone condizioni sul codice che utilizza il generatore
  - per questo è messa alla fine
  - tipica degli iteratori su tipi di dato modificabili

Specifica di un iteratore stand alone

```
public class Num {
// come prima più
 public static Iterator allPrimes ()
 // EFFECTS: ritorna un generatore che produrrà tutti
 // i numeri primi (come Integers) ciascuno una
 // sola volta, in ordine arbitrario
                                    }
```

- il limite al numero di iterazioni deve essere imposto dall'esterno
  - il generatore può produrre infiniti elementi

Utilizzazione degli iteratori 1

```
public Iterator terms ()
 // EFFECTS: ritorna un generatore che produrrà gli
 // esponenti dei termini diversi da 0 in this (come
 // Integers) fino al grado del polinomio, in ordine
 // crescente
public class Comp {
 public static Poly diff (Poly p) throws NullPointerException
  // EFFECTS: se p è null solleva NullPointerException
  // altrimenti ritorna il poly ottenuto differenziando
  // p
     {Poly q = new Poly();
     Iterator g = p.terms();
     while (g.hasNext()) {
     int exp = ((Integer) g.next()).intValue();
     if (exp == 0) continue; // ignora il termine 0
   q = q.add (new Poly(exp*p.coeff(exp), exp-1));
   return q;}}
```

- implementazione di diff esterna alla classe `Poly`

Utilizzazione degli iteratori 2

```
public static Iterator allPrimes ()
 // EFFECTS: ritorna un generatore che produrrà tutti
 // i numeri primi (come Integers) ciascuno una
 // sola volta, in ordine arbitrario


public static void printPrimes (int m) {
 // MODIFIES: System.out
 // EFFECTS: stampa tutti i numeri primi minori o uguali a m
 // su System. out
    Iterator g =      Num.allPrimes();
    while (true) {
    Integer p =      (Integer) g.next();
    if (p.intValue() > m) return; // forza la terminazione
    System.out.println("The next prime is: " + p.toString()); }
    }
```


Utilizzazione dei generatori

```
public static int max (Iterator g) throws EmptyException,
    NullPointerException {
// REQUIRES: il generatore g genera (contiene) solo Integers
// MODIFIES: g
// EFFECTS: se g è null solleva NullPointerException; se g è
// vuoto solleva EmptyException, altrimenti consuma tutti gli
// elementi di g e restituisce il massimo intero in g
try {int m = (Integer) g.next().intValue();
    while (g.hasNext())
     {int x = (Integer) g.next().intValue();
     if (m < x) m = x; } return m;}
catch (NoSuchElementException e)
{throw new EmptyException("Comp.max"); } }
```

- gli iteratori possono essere passati come argomento a procedure che cosí astraggono da dove provengono gli argomenti su cui lavorano
  - prodotti da `elements` di `IntSet`, `primesLT100`, ...

Implementazione degli iteratori e dei generatori

- i generatori sono oggetti che hanno come tipo un sottotipo di `Iterator`
  - istanze di una classe γ che "implementa" l'interfaccia `Iterator`
- un iteratore α è un metodo (stand alone o associato ad un tipo astratto) che ritorna il generatore istanza di γ
  - γ deve essere contenuta nello stesso modulo che contiene α
    - dall'esterno del modulo si deve poter vedere solo l'iteratore α (con la sua specifica)
    - non la classe γ che definisce il generatore
- γ deve avere una visibilità limitata al package che contiene α
  - oppure può essere contenuta nella classe che contiene α
    - come inner class privata
- dall'esterno i generatori sono visti come oggetti di tipo `Iterator`
  - perché il sottotipo γ non è visibile

Classi nidificate

- una classe γ dichiarata all'interno di una classe α può essere
  - static (di proprietà della classe α)
  - di istanza (di proprietà degli oggetti istanze di α)
- se γ è static come sempre non può accedere direttamente le variabili di istanza ed i metodi di istanza di α
  - le classi che definiscono i generatori si possono quasi sempre definire come inner classes statiche

Classi nidificate: semantica

- la presenza di classi nidificate richiede la presenza di un ambiente di classi
  - all'interno delle descrizioni di classi
  - all'interno degli oggetti (per classi interne non static)
  - vanno modificate di conseguenza anche tutte le regole che accedono i nomi

Implementazione degli iteratori 1

```
public class Poly {
 private int[] termini;
 private int deg;
 public Iterator terms () {return new PolyGen(this); }
 // EFFECTS: ritorna un generatore che produrrà gli
 // esponenti dei termini diversi da 0 in this (come
 //Integers) fino al grado del polinomio, in ordine crescente
 private static class PolyGen implements Iterator {
 // inner class (classe annidata)
    private Poly p; // il Poly su cui si itera
    private int n; // il prossimo termine da considerare
    PolyGen (Poly it) {
    // REQUIRES: it != null
      p = it; if (p.termini[0] == 0) n = 1; else n = 0; }
    public boolean hasNext () {return n <= p.deg; }
    public Object next () throws NoSuchElementException {
  for (int e = n; e <= p.deg; e++)
     if (p.termini[e] != 0){n = e + 1; return new Integer(e); }
    throw new NoSuchElementException("Poly.terms"); } } }
```


Implementazione degli iteratori 2

```
public class Poly {
 private int[] termini;
 private int deg;
 public Iterator terms () {return new PolyGen(); }
 // EFFECTS: ritorna un generatore che produrrà gli
 // esponenti dei termini diversi da 0 in this (come
 //Integers) fino al grado del polinomio, in ordine crescente
 private class PolyGen implements Iterator {
 // inner class (classe annidata)
    private int n; // il prossimo termine da considerare
    PolyGen () {
    // REQUIRES: it != null
      if (termini[0] == 0) n = 1; else n = 0; }
    public boolean hasNext () {return n <= deg; }
    public Object next () throws NoSuchElementException {
  for (int e = n; e <= deg; e++)
     if (termini[e] != 0) {n = e + 1; return new Integer(e); }
    throw new NoSuchElementException("Poly.terms"); } } }
```


Implementazione degli iteratori 3

```
public class Num {
 public static Iterator allPrimes (){return new PrimesGen();}
 // EFFECTS: ritorna un generatore che produrrà tutti
 //i numeri primi (come Integers) ciascuno una
 //sola volta, in ordine arbitrario
 private static class PrimeGen implements Iterator {
 // inner class (classe annidata)
    private Vector ps; // primi già dati
    private int p; // prossimo candidato alla generazione
    PrimesGen () {p = 2; ps = new Vector(); }
    public boolean hasNext () {return true }
    public Object next () {
     if (p == 2) { p = 3; return 2;}
   for (int n = p; true; n = n + 2)
     for (int i = 0; i < ps.size(); i++){
      int e1 = ((Integer) ps.get(i)).intValue();
        if (n%e1 == 0) break; // non è primo
      if (e1*e1 > n)
       {ps.add(new Integer(n); p = n + 2; return n;}}} }}
```


Classi nidificate e generatori

- le classi i cui oggetti sono generatori definiscono comunque dei tipi astratti
  - sottotipi di `Iterator`
- in quanto tali devono essere dotati di
  - una funzione di astrazione
  - un invariante di rappresentazione

Funzione di astrazione per i generatori

- dobbiamo sapere cosa sono gli stati astratti
- per tutti i generatori, lo stato astratto è
  - la sequenza di elementi che devono ancora essere generati
  - la funzione di astrazione mappa la rappresentazione su tale sequenza

Funzione di astrazione 1

```
public class Poly {
 private int[] termini;
 private int deg;
 public Iterator terms () {return new PolyGen(this); }
 private static class PolyGen implements Iterator {
 // inner class (classe annidata)
    private Poly p; // il Poly su cui si itera
    private int n; // il prossimo termine da considerare
 // la funzione di astrazione
 // α(c) = [ x1, x2, ...] tale che
 // ogni xi è un Integer, e
 // gli xi sono tutti e soli gli indici i>=n
 // per cui c.p.termini[i] != 0, e
 // xi > xj per tutti gli i > j >= 1
```

- notare che si usano le rep sia di `Poly` che di `Polygen`

Invariante di rappresentazione 1

```
public class Poly {
 private int[] termini;
 private int deg;
 public Iterator terms () {return new PolyGen(this); }
 private static class PolyGen implements Iterator {
  // inner class (classe annidata)
     private Poly p; // il Poly su cui si itera
     private int n; // il prossimo termine da considerare
  // l'invariante di rappresentazione:
  // i(c) = c.p != null e
  // (0 <= c.n <= c.p.deg)
```

- notare che si usano le rep sia di `Poly` che di `Polygen`

Funzione di astrazione 2

```
public class Num {
 public static Iterator allPrimes (){return new PrimesGen();}
 private static class PrimeGen implements Iterator {
  // inner class (classe annidata)
     private Vector ps; // primi già dati
     private int p; // prossimo candidato alla generazione
  // la funzione di astrazione
  // α(c) = [p1, p2, ...] tale che
  // ogni pi è un Integer, è primo, ed è >= c.p e
  // tutti i numeri primi >= n occorrono nella
  //     sequenza, e
  // pi > pj per tutti gli i > j >= 1
```


Invariante di rappresentazione 2

```
public class Num {
 public static Iterator allPrimes (){return new PrimesGen();}
 private static class PrimeGen implements Iterator {
  // inner class (classe annidata)
     private Vector ps; // primi già dati
     private int p; // prossimo candidato alla generazione
  // l'invariante di rappresentazione:
  // i(c) = c.ps != null e
  // tutti gli elementi di c.ps sono primi e
  //     ordinati in modo crescente, e
  // contengono tutti i primi < c.p e > 2
```


Conclusioni sugli iteratori

- in molti tipi di dato astratti (collezioni) gli iteratori sono un componente essenziale
  - supportano l'astrazione via specifica
  - portano a programmi efficienti in tempo e spazio
  - sono facili da usare
  - non distruggono la collezione
  - ce ne possono essere più d'uno
- se il tipo di dato astratto è modificabile ci dovrebbe sempre essere il vincolo sulla non modificabilità del dato durante l'uso dell'iteratore
  - altrimenti è molto difficile specificarne il comportamento previsto
  - in alcuni casi può essere utile combinare generazioni e modifiche

Generare e modificare

- programma che esegue tasks in attesa su una coda di tasks

```
Iterator g = q.allTasks();
while (g.hasNext()) {
    Task t = (Task) g.next();
    // esecuzione di t
    // se t genera un nuovo task nt, viene messo
    // in coda facendo q.enq(nt)
    }
```

- casi come questo sono molto rari

Un esempio con iteratore: le liste ordinate di interi

Un nuovo esempio completo: le liste ordinate

- `OrderedIntList`
- lista ordinata di interi
  - modificabile
  - essenziale la presenza di un iteratore
    - altrimenti sarebbe complesso accedere gli elementi

Specifica di `OrderedIntList` 1

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 // costruttore
 public OrderedIntList ()
    // EFFECTS: inizializza this alla lista vuota
 // metodi
 public void addEl (int el) throws
    DuplicateException
  // MODIFIES: this
  // EFFECTS: aggiunge el a this, se el non occorre in
  // this, altrimenti solleva DuplicateException
 public void remEl (int el) throws
    NotFoundException
  // MODIFIES: this
  // EFFECTS: toglie el da this, se el occorre in
  // this, altrimenti solleva NotFoundException
```


Specifica di `OrderedIntList` 2

```
 public boolean isIn (int el)
  // EFFECTS: se el appartiene a this ritorna
  // true, altrimenti false
 public boolean isEmpty ()
  // EFFECTS: se this è vuoto ritorna true, altrimenti
  // false
 public int least () throws EmptyException
  // EFFECTS: se this è vuoto solleva EmptyException
  // altrimenti ritorna l'elemento minimo in this
 public Iterator smallToBig ()
  // EFFECTS: ritorna un generatore che produrrà gli
  // elementi di this (come Integers), in ordine
  // crescente
  // REQUIRES: this non deve essere modificato finché
  // il generatore è in uso
 public boolean repOk ()
 public String toString ()
}
```


Specifica di `OrderedIntList` 3

```
public class OrderedIntList {
 public OrderedIntList ()
 public void addEl (int el) throws
    DuplicateException
 public void remEl (int el) throws
    NotFoundException
 public boolean isIn (int el)
 public boolean isEmpty ()
 public int least () throws EmptyException
 public Iterator smallToBig ()
 public boolean repOk ()
 public String toString ()}
```

- senza l'iteratore, non ci sarebbe nessuna operazione per accedere gli elementi
- `DuplicateException` e `NotFoundException` checked

Implementazione di `OrderedIntList` 1

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
```

- la rep contiene
  - una variabile `boolean` che ci dice se la lista è vuota
  - la variabile intera che contiene l'eventuale valore dell'elemento
  - due (puntatori a) `OrderedIntList`s che contengono la lista di quelli minori e quelli maggiori, rispettivamente
- implementazione ricorsiva
- lista doppia
  - si può percorrere da ogni elemento in avanti o all'indietro

Implementazione di `OrderedIntList` 2

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // la funzione di astrazione (ricorsiva!)
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 // l'invariante di rappresentazione (ricorsivo!)
 // I(c) = c.vuota oppure
 // (c.prima != null e c.dopo != null e
 // I(c.prima) e I(c.dopo) e
 // (!c.prima.isEmpty() -> c.prima.max() < c.val) e
 // (!c.dopo.isEmpty() -> c.dopo.least() >= c.val) )
```

- l'invariante utilizza metodi esistenti
  - `isEmpty` e `least`
- \+ `max`

Implementazione di OrderedIntList 3

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
//tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // costruttore
 public OrderedIntList ()
    // EFFECTS: inizializza this alla lista vuota
  { vuota = true; }
```

- il costruttore inizializza solo la variabile vuota

Implementazione di OrderedIntList 3.1

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 // I(c) = c.vuota oppure
 // (c.prima != null e c.dopo != null e
 // I(c.prima) e I(c.dopo) e
 // (!c.prima.isEmpty() -> c.prima.max() < c.val) e
 // (!c.dopo.isEmpty() -> c.dopo.least() >= c.val) )
 public OrderedIntList ()
    // EFFECTS: inizializza this alla lista vuota
  { vuota = true; }
```

- l'implementazione del costruttore
  - soddisfa l'invariante (c.vuota = true)
  - verifica la propria specifica (α(c.vuota) = [])

Implementazione di OrderedIntList 4

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
//tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 public void addEl (int el) throws DuplicateException
  // MODIFIES: this
  // EFFECTS: aggiunge el a this, se el non occorre in
  //this, altrimenti solleva DuplicateException
  {if (vuota) {
    prima = new OrderedIntList();
    dopo = new OrderedIntList(); val = el;
    vuota = false; return; }
   if (el == val) throw new
       DuplicateException("OrderedIntList.addEl");
   if (e1 < val) prima.addEl(e1);
     else dopo.addEl(el); }
```

- propaga automaticamente l'eventuale eccezione sollevata dalle chiamate ricorsive

Implementazione di OrderedIntList 4.1

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 //I(c) = c.vuota oppure
 // (c.prima != null e c.dopo != null e
 //I(c.prima) e I(c.dopo) e
 // (!c.prima.isEmpty() -> c.prima.max() < c.val) e
 // (!c.dopo.isEmpty() -> c.dopo.least() >= c.val) )
 public void addEl (int el) throws DuplicateException
   {if (vuota) {
    prima = new OrderedIntList();
    dopo = new OrderedIntList(); val = el;
    vuota = false; return; }
   ...
```

prima != null e dopo != null (calcolati dal costruttore)

I(prima) e I(dopo) (calcolati dal costruttore)

le implicazioni sono vere perché la premessa è falsa

Implementazione di OrderedIntList 4.2

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 //I(c) = c.vuota oppure
 // (c.prima != null e c.dopo != null e
 //I(c.prima) e I(c.dopo) e
 // (!c.prima.isEmpty() -> c.prima.max() < c.val) e
 // (!c.dopo.isEmpty() -> c.dopo.least() >= c.val) )
 public void addEl (int el) throws DuplicateException
   ...
   if (e1 < val) prima.addEl(e1);
     else dopo.addEl(el); }
```

- this non è vuoto

prima != null e dopo != null

I(prima) e I(dopo) (calcolati da una chiamata ricorsiva)

- ramo then: il nuovo massimo di prima è (induttivamente) minore di val
- ramo else: il nuovo minimo di dopo è (induttivamente) maggiore di val

Implementazione di OrderedIntList 4.3

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public void addEl (int el) throws DuplicateException
  // MODIFIES: this
  // EFFECTS: aggiunge el a this, se el non occorre in
  //this, altrimenti solleva DuplicateException
  {if (vuota) {
    prima = new OrderedIntList();
    dopo = new OrderedIntList(); val = el;
    vuota = false; return; }
   ...
```

- $\alpha(c_{pre}) = []$
- $\alpha(c.prima) = []$
- $\alpha(c.dopo) = []$
- $[c.val] = [e1]$
- $\alpha(c) = [e1]$

Implementazione di OrderedIntList 4.4

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public void addEl (int el) throws DuplicateException
  // MODIFIES: this
  // EFFECTS: aggiunge el a this, se el non occorre in
  // this, altrimenti solleva DuplicateException
  ...
  if (el == val) throw new
      DuplicateException("OrderedIntList.addEl");
```

- se ci sono elementi duplicati solleva l'eccezione, eventualmente propagando eccezioni sollevate dalle chiamate ricorsive (vedi dopo)

Implementazione di OrderedIntList 4.5

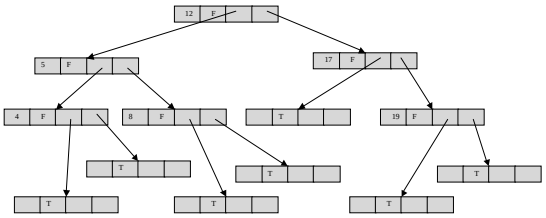
```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public void addEl (int el) throws DuplicateException
  // MODIFIES: this
  // EFFECTS: aggiunge el a this, se el non occorre in
  // this, altrimenti solleva DuplicateException
  ...
  if (e1 < val) prima.addEl(e1);
    else dopo.addEl(el); }
```

- $\alpha(c_{pre}) = \alpha(c.prima_{pre}) + [c.val] + \alpha(c.dopo_{pre})$
- se `e1 < val` la chiamata ricorsiva solleva l'eccezione oppure produce
- $\alpha(c.prima) = aggiunge\ e1\ a\ prima_{pre}$
- $\alpha(c.dopo) = \alpha(c.dopo_{pre})$
- $\alpha(c) = aggiunge\ e1\ a\ c_{pre}$

Come è fatta una OrderedIntList

- vediamo la lista prodotta dalla sequenza di comandi

```
OrderedIntList ls = new OrderedIntList();
ls.addEl(12); ls.addEl(5); ls.addEl(17);
ls.addEl(4);ls.addEl(8); ls.addEl(19);
```



Implementazione di OrderedIntList 5

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 public void remEl (int el) throws NotFoundException
  // MODIFIES: this
  // EFFECTS: toglie el da this, se el occorre in
  // this, altrimenti solleva NotFoundException
  {if (vuota) throw new
      NotFoundException("OrderedIntList.remEl");
  if (el == val)
   try { val = dopo.least(); dopo.remEl(val); }
   catch (EmptyException e)
    { vuota = prima.vuota; val = prima.val;
     dopo = prima.dopo; prima = prima.prima; return;}
  else if (e1 < val) prima.remEl(e1);
    else dopo.remEl(el); }
```


Implementazione di OrderedIntList 6

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 public boolean isIn (int el)
  // EFFECTS: se el appartiene a this ritorna
  // true, altrimenti false
  {if (vuota) return false;
  if (el == val) return true;
  if (e1 < val) return prima.isIn(el); else return
      dopo.isIn(el); }
 public boolean isEmpty ()
  // EFFECTS: se this è vuoto ritorna true, altrimenti false
  {return vuota; }
```


Implementazione di OrderedIntList 6.1

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public boolean isIn (int el)
  // EFFECTS: se el appartiene a this ritorna
  // true, altrimenti false
  {if (vuota) return false;
  if (el == val) return true;
  if (e1 < val) return prima.isIn(el); else return
      dopo.isIn(el); }
 public boolean isEmpty ()
  // EFFECTS: se this è vuoto ritorna true, altrimenti false
  {return vuota; }
```

- dimostrazioni di correttezza ovvie

Implementazione di OrderedIntList 7

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
//tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 public int least () throws EmptyException
  // EFFECTS: se this è vuoto solleva EmptyException
  // altrimenti ritorna l'elemento minimo in this
 {if (vuota) throw new
    EmptyException("OrderedIntList.least");
  try { return prima.least(); }
  catch (EmptyException e) {return val;} }
```


Implementazione di OrderedIntList 7.1

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 // α(c) = se c.vuota allora [], altrimenti
 // α(c.prima) + [c.val] + α(c.dopo)
 public int least () throws EmptyException
  // EFFECTS: se this è vuoto solleva EmptyException
  // altrimenti ritorna l'elemento minimo in this
 {if (vuota) throw new
    EmptyException("OrderedIntList.least");
  try { return prima.least(); }
  catch (EmptyException e) {return val;} }
```

- dimostrazione di correttezza ovvia

Implementazione di OrderedIntList 8

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
//tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 public Iterator smallToBig ()
  // EFFECTS: ritorna un generatore che produrrà gli
  // elementi di this (come Integers), in ordine
  // crescente
  // REQUIRES: this non deve essere modificato finché
  // il generatore è in uso
 {return new OLGen(this, count());}

 private int count () {
  if (vuota) return 0;
  return 1 + prima.count() + dopo.count(); }
```

- al generatore viene passato il numero di elementi della lista
  - calcolato dal metodo privato count

Implementazione del generatore di OrderedIntList 1

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
//tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo
  // la funzione di astrazione (ricorsiva!)
  // α(c) = se c.cnt = 0 allora [], altrimenti
  // se il numero di elementi di α(c.figlio) è c.cnt
  // allora α(c.figlio),
  // altrimenti α(c.figlio) + [Integer(c.me.val)] +
  //    α(OLGen(c.dopo))
```


Implementazione del generatore di OrderedIntList 2

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
//tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo
  // l'invariante di rappresentazione
  // I(c) = c.cnt = 0 oppure
  // (c.cnt > 0 e c.me != null e c.figlio != null e
  // (c.cnt = c.figlio.cnt + 1 oppure
  // c.cnt = c.figlio.cnt + c.me.dopo.count() + 1))
```


Implementazione del generatore di OrderedIntList 3

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
//tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo

  OLGen (OrderedIntList o, int n) {
   // REQUIRES: o != null
   cnt = n;
   if (cnt > 0) { me = o;
    figlio = new OLGen(o.prima, o.prima.count()); }
```

- anche il costruttore è ricorsivo

Implementazione del generatore di `OrderedIntList` 3.1

```
public class OrderedIntList {
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo
// I(c) = c.cnt = 0 oppure
 // (c.cnt > 0 e c.me != null e c.figlio != null e
 // (c.cnt = c.figlio.cnt + 1 oppure
 // c.cnt = c.figlio.cnt + c.me.dopo.count() + 1))
  OLGen (OrderedIntList o, int n) {
   // REQUIRES: o != null
   cnt = n;
   if (cnt > 0) { me = o;
    figlio = new OLGen(o.prima, o.prima.count()); }
```

- *cnt = 0 oppure*
- *cnt > 0 e me != null (clausola REQUIRES) e*
- *figlio != null (chiamata ricorsiva) e*
- *disgiunzione sui count (proprietà di count)*

Implementazione del generatore di `OrderedIntList` 4

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo

  public boolean hasNext () {
   return cnt > 0; }
```

- si noti l'uso di `cnt` per rendere efficiente anche questo metodo

Implementazione del generatore di `OrderedIntList` 5

```
public class OrderedIntList {
// OVERVIEW: una OrderedIntList è una lista
// modificabile di interi ordinata
// tipico elemento: [x1, ..., xn], xi<xj se i<j
 private boolean vuota;
 private OrderedIntList prima, dopo;
 private int val;
 private static class OLGen implements Iterator {
  private int cnt; // numero di elementi ancora da generare
  private OLGen figlio; // sottogeneratore corrente
  private OrderedIntList me; // il mio nodo
  public Object next () throws NoSuchElementException {
   if (cnt == 0) throw new
     NoSuchElementException("OrderedIntList.smallToBig");
   cnt --;
   try { return new Integer(figlio.next()); }
   catch (NoSuchElementException e) {}
   // se arriva qui ha finito tutti quelli prima
   figlio = new OLGen(me.dopo, cnt);
   return new Integer(me.val); }
```


Le s-espressioni

Un nuovo esempio completo: le s-espressioni

- `Sexpr`
- alberi binari (possibilmente "vuoti") che hanno sulle foglie atomi (stringhe)
- sono la struttura dati base del linguaggio LISP
  - modificabile
  - la nostra versione è dotata di un iteratore

Specifica di `Sexpr` 1

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
// costruttori
 public Sexpr ()
    // EFFECTS: inizializza this alla Sexpr vuota
 public Sexpr (String s)
    // EFFECTS: inizializza this alla foglia contenente s
// metodi
 public Sexpr cons (Sexpr s) throws
   NullPointerException
 // EFFECTS: costruisce un nuovo albero binario che ha
 // this come sottoalbero sinistro ed s come
 // sottoalbero destro. Se this o s sono indefiniti,
 // solleva NullPointerException
 public void rplaca (Sexpr s) throws
   NotANodeException
 // MODIFIES: this
 // EFFECTS: rimpiazza in this il sottoalbero sinistro
 // con s. Se this non è un nodo binario solleva
 // NotANodeException (checked)
```

Specifica di Sexpr 2

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 public void rplacd (Sexpr s) throws
    NotANodeException
  // MODIFIES: this
  // EFFECTS: rimpiazza in this il sottoalbero destro
  // con s. Se this non è un nodo binario solleva
  // NotANodeException (checked)
 public Sexpr car () throws NotANodeException
  // EFFECTS: ritorna il sottoalbero sinistro di this.
  // Se this non è un nodo binario solleva
  // NotANodeException (checked)
 public Sexpr cdr () throws NotANodeException
  // EFFECTS: ritorna il sottoalbero destro di this.
  // Se this non è un nodo binario solleva
  // NotANodeException (checked)
```


Specifica di Sexpr 3

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 public boolean nullS () throws NullPointerException
  // EFFECTS: ritorna true se this è l'albero vuoto,
  // altrimenti ritorna false. Se this è indefinito solleva
  // NullPointerException
 public boolean atom () throws NullPointerException
  // EFFECTS: ritorna false se this è un albero binario,
  // altrimenti ritorna true. Se this è indefinito solleva
  // NullPointerException
 public Iterator leaves ()
  // EFFECTS: ritorna un generatore che produrrà le foglie
  // nella frontiera di this (come Strings), da sinistra a
  // destra
  // REQUIRES: this non deve essere modificato finché
  // il generatore è in uso

}
```


Implementazione di Sexpr 1

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
```

- la rep contiene
  - una variabile boolean che ci dice se l'albero è vuoto oppure consiste di una sola foglia
  - la variabile stringa che contiene l'eventuale stringa associata alla foglia
  - due (puntatori a) Sexpr che contengono i sottoalberi sinistro e destro, rispettivamente
- implementazione ricorsiva

Implementazione di Sexpr 2

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 // la funzione di astrazione (ricorsiva!)
 // α(c) =
 //   se c.foglia && c.stringa = "", Sexpr vuota
 //   se c.foglia && c.stringa = s, foglia s
 //   altrimenti è l'albero che ha come sottoalberi
 //   sinistro e destro α(c.sinistro) e α(c.destro)
 // l'invariante di rappresentazione (ricorsivo!)
 // I(c) = c.foglia oppure
 // (!c.foglia e c.sinistro != null e c.destro != null
 // e I(c.sinistro) e I(c.destro))
```


Implementazione di Sexpr 3

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
// costruttori
 public Sexpr ()
    // EFFECTS: inizializza this alla Sexpr vuota
  {foglia = true; stringa = ""; }
 public Sexpr (String s)
    // EFFECTS: inizializza this alla foglia contenente s
  {foglia = true; stringa = s; }
```


Implementazione di Sexpr 3.1

```
public class Sexpr {
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 // α(c) =
 //   se c.foglia && c.stringa = "", Sexpr vuota
 //   se c.foglia && c.stringa = s, foglia s
 //   altrimenti è l'albero che ha come sottoalberi
 //   sinistro e destro α(c.sinistro) e α(c.destro)
 // I(c) = c.foglia oppure
 // (!c.foglia e c.sinistro != null e c.destro != null
 // e I(c.sinistro) e I(c.destro))
 public Sexpr ()
    // EFFECTS: inizializza this alla Sexpr vuota
  {foglia = true; stringa = ""; }
```

- l'invariante è soddisfatto (foglia è true)
- la specifica è soddisfatta

α(c) = Sexpr vuota, perché foglia && stringa = ""

Implementazione di `Sexpr` 3.2

```
public class Sexpr {
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 // α(c) =
 //   se c.foglia && c.stringa = "", Sexpr vuota
 //   se c.foglia && c.stringa = s, foglia s
 //   altrimenti è l'albero che ha come sottoalberi
 //   sinistro e destro α(c.sinistro) e α(c.destro)
 // I(c) = c.foglia oppure
 // (!c.foglia e c.sinistro != null e c.destro != null
 // e I(c.sinistro) e I(c.destro))
public Sexpr (String s)
    // EFFECTS: inizializza this alla foglia contenente s
 {foglia = true; stringa = s; }
```

- l'invariante è soddisfatto (`foglia` è `true`)
- la specifica è soddisfatta

α(c) = foglia s, perché foglia && stringa = s

Implementazione di `Sexpr` 4

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 public Sexpr cons (Sexpr s) throws
    NullPointerException
 // EFFECTS: costruisce un nuovo albero binario che ha
 // this come sottoalbero sinistro ed s come
 // sottoalbero destro. Se this o s sono indefiniti,
 // solleva NullPointerException
 {if (this == null || s == null) throw new
      NullPointerException ("Sexpr.cons");
  Sexpr nuovo = new Sexpr();
  nuovo.sinistro = this; nuovo.destro = s;
  nuovo.foglia = false; return nuovo; }
```


Implementazione di `Sexpr` 4.1

```
public class Sexpr {
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 // I(c) = c.foglia oppure
 // (!c.foglia e c.sinistro != null e c.destro != null
 // e I(c.sinistro) e I(c.destro))
 public Sexpr cons (Sexpr s) throws
    NullPointerException
 {if (this == null || s == null) throw new
      NullPointerException ("Sexpr.cons");
  Sexpr nuovo = new Sexpr();
  nuovo.sinistro = this; nuovo.destro = s;
  nuovo.foglia = false; return nuovo; }
```

- *!nuovo.foglia*
- *nuovo.sinistro != null e nuovo.destro != null*
- *I(nuovo.sinistro) e I(nuovo.destro), perché per induzione I(this) e I(s)*

Implementazione di `Sexpr` 4.2

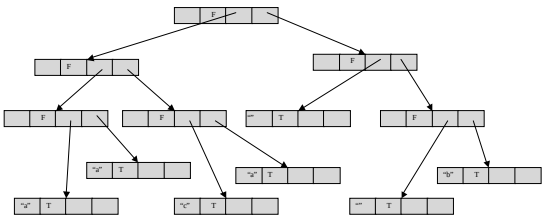
```
public class Sexpr {
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 // α(c) =
 //   se c.foglia && c.stringa = "", Sexpr vuota
 //   se c.foglia && c.stringa = s, foglia s
 //   altrimenti è l'albero che ha come sottoalberi
 //   sinistro e destro α(c.sinistro) e α(c.destro)
 public Sexpr cons (Sexpr s) throws
    NullPointerException
 // EFFECTS: costruisce un nuovo albero binario che ha
 // this come sottoalbero sinistro ed s come
 // sottoalbero destro. Se this o s sono indefiniti,
 // solleva NullPointerException
 {if (this == null || s == null) throw new
      NullPointerException ("Sexpr.cons");
  Sexpr nuovo = new Sexpr();
  nuovo.sinistro = this; nuovo.destro = s;
  nuovo.foglia = false; return nuovo; }
```

- correttezza ovvia

Come è fatta una `Sexpr`

- vediamo l'albero prodotto dalla espressione

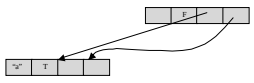
```
(((((new Sexpr("a")).cons(new Sexpr("a"))).
  cons((new Sexpr("c")).cons(new Sexpr("a")))).
   cons((new Sexpr()).cons((new Sexpr()).cons(new Sexpr("b"))))))
```



Come è fatta una `Sexpr` 2

- sottoalberi possono essere condivisi

```
Sexpr s1 = new Sexpr("a");
Sexpr s2 = s1.cons(s1);
```



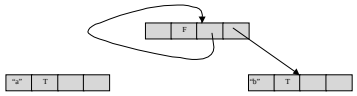
Implementazione di Sexpr 5

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 public void rplaca (Sexpr s) throws
    NotANodeException
 // MODIFIES: this
 // EFFECTS: rimpiazza in this il sottoalbero sinistro
 // con s. Se this non è un nodo binario solleva
 // NotANodeException (checked)
 {if (foglia) throw new
        NotANodeException("Sexpr.rplaca");
   sinistro = s; return; }
```


Come è fatta una Sexpr 3

- usando le operazioni che modificano (rplaca e rplacd) si possono costruire strutture cicliche

```
Sexpr s1 = new Sexpr("a"); Sexpr s2 = new Sexpr("b");
Sexpr s3 = s1.cons(s2); s3.rplaca(s3);
```



Implementazione di Sexpr 5.1

```
public class Sexpr {
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 // I(c) = c.foglia oppure
 // (!c.foglia e c.sinistro != null e c.destro != null
 // e I(c.sinistro) e I(c.destro))
 public void rplaca (Sexpr s) throws
    NotANodeException
   {if (foglia) throw new
        NotANodeException("Sexpr.rplaca");
   sinistro = s; return; }
```

- l'invariante è soddisfatto
  - foglia è false
  - destro non è modificato
  - sinistro soddisfa l'invariante per induzione

Implementazione di Sexpr 5.2

```
public class Sexpr {
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 // α(c) =
 //  se c.foglia && c.stringa = "", Sexpr vuota
 //  se c.foglia && c.stringa = s, foglia s
 //  altrimenti è l'albero che ha come sottoalberi
 //  sinistro e destro α(c.sinistro) e α(c.destro)
 public void rplaca (Sexpr s) throws
    NotANodeException
 // MODIFIES: this
 // EFFECTS: rimpiazza in this il sottoalbero sinistro
 // con s. Se this non è un nodo binario solleva
 // NotANodeException (checked)
 {if (foglia) throw new
        NotANodeException("Sexpr.rplaca");
   sinistro = s; return; }
```

- soddisfa chiaramente la specifica

Implementazione di Sexpr 6

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 public void rplacd (Sexpr s) throws
    NotANodeException
 // MODIFIES: this
 // EFFECTS: rimpiazza in this il sottoalbero destro
 // con s. Se this non è un nodo binario solleva
 // NotANodeException (checked)
 {if (foglia) throw new
        NotANodeException("Sexpr.rplacd");
   destro = s; return; }
```


Implementazione di Sexpr 7

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 public Sexpr car () throws NotANodeException
 // EFFECTS: ritorna il sottoalbero sinistro di this.
 // Se this non è un nodo binario solleva
 // NotANodeException (checked)
 {if (foglia) throw new
        NotANodeException("Sexpr.car");
  return sinistro; }
 public Sexpr cdr () throws NotANodeException
 // EFFECTS: ritorna il sottoalbero destro di this.
 // Se this non è un nodo binario solleva
 // NotANodeException (checked)
 {if (foglia) throw new
        NotANodeException("Sexpr.cdr");
  return destro; }
```

Implementazione di Sexpr 8

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 public boolean nullS () throws NullPointerException
  // EFFECTS: ritorna true se this è l'albero vuoto,
  // altrimenti ritorna false. Se this è indefinito solleva
  // NullPointerException
  {if (this == null) throw new NullPointerException();
     if (! foglia) return false;
     if (stringa == "") return true; return false; }
 public boolean atom () throws NullPointerException
  // EFFECTS: ritorna false se this è un albero binario,
  // altrimenti ritorna true. Se this è indefinito solleva
  // NullPointerException
  {if (this == null) throw new NullPointerException();
     return foglia;}
```


Implementazione di Sexpr 9

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 public Iterator leaves ()
  // REQUIRES: this non deve essere ciclico
  // EFFECTS: ritorna un generatore che produrrà le foglie
  // nella frontiera di this (come Strings), da sinistra a
  // destra
  // REQUIRES: this non deve essere modificato finché
  // il generatore è in uso
  {return new LeavesGen(this,numerofoglie());}
 private int numerofoglie () {
     if (foglia) {if (stringa == "") {return 0; }
     else {return 1; } }
     try {return (car().numerofoglie() +
     cdr().numerofoglie()); }
     catch (NotANodeException e) {return 0; } }
```

- il try & catch solo per evitare di dichiarare l'eccezione!

Implementazione del generatore di Sexpr 1

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 private static class LeavesGen implements Iterator {
    private LeavesGen figlio; // sottogeneratore corrente
    private Sexpr io; //il mio albero
    private int quanti; //numero di elementi ancora da generare
    //la funzione di astrazione (ricorsiva!)
 // α(c) = se c.quanti = 0 allora [],
 // se c.quanti = 1 allora [c.io.stringa] ,
 // altrimenti α(c.figlio) + α(LeavesGen(c.destra))
```


Implementazione del generatore di Sexpr 2

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 private static class LeavesGen implements Iterator {
    private LeavesGen figlio; // sottogeneratore corrente
    private Sexpr io; //il mio albero
    private int quanti; //numero di elementi ancora da generare
    //l'invariante di rappresentazione
 // I(c) = (c.quanti = 0 e c.io.foglia e
 // c.io.stringa = "") oppure
 // (c.quanti = 1 e c.io.foglia e c.io.stringa != "")
 // oppure
 // (c.quanti > 0 e c.io != null e c.figlio != null e
 // c.quanti = c.figlio.quanti +
 //   c.io.destra.numerofoglie())
```


Esercizio per casa

- implementare `repOK` e `toString` per il generatore e provarli su qualche esempio per vedere se sono giusti

Implementazione del generatore di Sexpr 3

```
public class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 private static class LeavesGen implements Iterator {
    private LeavesGen figlio; // sottogeneratore corrente
    private Sexpr io; //il mio albero
    private int quanti; //numero di elementi ancora da generare
    LeavesGen (Sexpr s,int n) {
     //REQUIRES: s != null
     quanti = n;
     if (quanti > 0)
    {io = s; if (io.foglia) return;
       try {figlio = new LeavesGen(io.car(),
        io.car().numerofoglie()); }
        catch (NotANodeException e) {}
        return; }
     return;}
```

- il try & catch solo per evitare di dichiarare l'eccezione!

Implementazione del generatore di `Sexpr` 4

```
public class Sexpr {
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 private static class LeavesGen implements Iterator {
    private LeavesGen figlio; // sottogeneratore corrente
    private Sexpr io; // il mio albero
    private int quanti; // numero di elementi ancora da generare
    public boolean hasNext() { return quanti > 0;}
    public Object next() throws NoSuchElementException {
     if (quanti == 0) throw new
      NoSuchElementException("Sexpr.leaves");
     quanti--; if (io.foglia) {return io.stringa;}
     try {return figlio.next(); }
     catch (NoSuchElementException e) {}
     try {figlio = new LeavesGen(io.cdr(),
     io.cdr().numerofoglie()); return figlio.next(); }
     catch (NotANodeException e) {
      throw new NoSuchElementException("Sexpr.leaves");}
```


Le gerarchie di tipi

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?

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

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
  - astraiamo diversi sottotipi a quello che hanno in comune
    - la specifica del loro supertipo

Sommario

- 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)

Tipo apparente e tipo effettivo

- supertipo τ, sottotipo σ
- un oggetto di tipo σ
  - può essere assegnato ad una variabile di tipo τ
  - 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

Tipo apparente e tipo effettivo: un esempio 1

- 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();
```


Tipo apparente e tipo effettivo: un esempio 2

```
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)

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

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

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

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

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

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

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`

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

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

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)
  // MODIFIES: this
  // EFFECTS: aggiunge x a this
 public void remove (int x)
  // MODIFIES: this
  // 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 `MaxIntSet` 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

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`

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`

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

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`
  - non trattato nella nostra semantica operazionale
- 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

`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

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.repOk ()) 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; } }
```

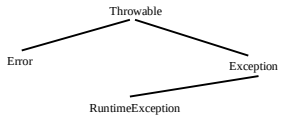

Cosa succede se il supertipo fa vedere la rappresentazione?

- 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

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;}}
```


Classi astratte come supertipi

- implementazione parziale di un tipo
- può avere variabili di istanza e uno o pi° ú 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

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

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(SortedIntSet) e in cui gli elementi sono
 // accessibili in modo ordinato
 public SortedIntSet ()
  // EFFECTS: inizializza this al SortedIntSet 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

`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

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
}
```


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`

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
}
```


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

Interfacce

- contiene 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`

Le gerarchie di tipi: implementazioni multiple e principio di sostituzione

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

Implementazioni multiple

- il tipo superiore della gerarchia
  - un'interfaccia
  - una classe astratta
- definisce una famiglia di tipi tale per cui
  - tutti i membri hanno esattamente gli stessi metodi e la stessa semantica
    - per esempio, ci potrebbero essere implementazioni sparse e dense dei polinomi, realizzate con sottoclassi
      - che forniscono l'implementazione di tutti i metodi astratti, in accordo con le specifiche del supertipo
      - in più hanno i costruttori
    - un programma potrebbe voler usare tutte e due le implementazioni
      - scegliendo ogni volta la più adeguata
  - gli oggetti dei sottotipi vengono dall'esterno tutti visti come oggetti dell'unico supertipo
  - dall'esterno si vedono solo i costruttori dei sottotipi

IntList

- il supertipo è una classe astratta
- usiamo i sottotipi per implementare i due casi della definizione ricorsiva
  - lista vuota
  - lista non vuota
  - la classe astratta ha alcuni metodi non astratti
    - comuni alle due sottoclassi
    - definiti in termini dei metodi astratti
  - la classe astratta non ha variabili di istanza e quindi nemmeno costruttori

Specifica del supertipo IntList

```
public abstract class IntList {
 // OVERVIEW: un IntList è una lista non modificabile di Integers.
 // Elemento tipico [x1,...,xn]
 public abstract Integer first () throws EmptyException;
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  //ritorna il primo elemento di this
 public abstract IntList rest () throws EmptyException;
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  //ritorna la lista ottenuta da this togliendo il primo elemento
 public abstract Iterator elements ();
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 //this (come Integers) nell'ordine che hanno in this
 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 del supertipo IntList

```
public abstract class IntList {
 // OVERVIEW: un IntList è una lista non modificabile di Integers.
 // Elemento tipico [x1,...,xn]
 // metodi astratti
 public abstract Integer first () throws EmptyException;
 public abstract IntList rest () throws EmptyException;
 public abstract Iterator elements ();
 public abstract IntList addEl (Integer x);
 public abstract int size ();
 public abstract boolean repOk ();
 // metodi concreti
 public String toString () {....};
 public boolean equals (IntList o) {
  // confronta gli elementi usando elements
  }
}
```

- toString e equals sono implementate
  - utilizzando il generatore elements

Implementazione del sottotipo 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 Iterator elements () { return new EmptyGen(); }
 public IntList addEl (Integer x)
 {return new FullIntList(x);}
 public int size () {....}
 public boolean repOk () {....}
 static private class EmptyGen implements Iterator {
     EmptyGen () {}
     public boolean hasNext () { return false; }
     public Object next () throws NoSuchElementException {
             throw new NoSuchElementException("IntList.elements"); }    }
}
```


Implementazione del sottotipo FullIntList

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


Poly

- il supertipo è una classe astratta
- la specifica della classe astratta è quella solita
- usiamo i sottotipi per realizzare due diverse implementazioni
  - DensePoly
  - SparsePoly
  - la classe astratta ha alcuni metodi non astratti
    - comuni alle due sottoclassi
    - definiti in termini dei metodi astratti
  - la classe astratta non ha variabili di istanza e quindi nemmeno costruttori

Implementazione del supertipo Poly

```
public abstract class Poly {
 protected int deg; //il grado
 protected Poly (int n) {deg = n; }
 public abstract int coeff (int d);
 public abstract Poly add (Poly q) throws NullPointerException;
 public abstract Poly mul (Poly q) throws NullPointerException;
 public abstract Poly minus ();
 public abstract Iterator terms ();
 public abstract boolean repOk ();
 public int degree () {return deg;}
 public Poly sub (Poly p) {return add(p.minus());}
 public String toString (){...}
 public boolean equals (Poly p)
  {if (p == null || deg != p.deg) return false;
     Iterator tg = terms(); Iterator pg = p.terms();
     while (tg.hasNext())
    {int tx = ((Integer) tg.next()).intValue( );
      int px = ((Integer) pg.next()).intValue( );
      if (tx != px || coeff (tx) != p. coeff (px) return false); } return true; }
}
```

le sottoclassi di `Poly`

- nelle operazioni si decide quale delle due rappresentazioni conviene usare
  - l'add di `DensePoly` lascia a `SparsePoly` la gestione del caso in cui i due oggetti sono diversi
- possono essere necessarie conversioni fra le due rappresentazioni
  - ed altri problemi simili

Implementazione del sottotipo `DensePoly` 1

```
public class DensePoly extends Poly {
 private int[] trms; // coefficienti fino a deg
 public DensePoly () { super(0); trms = new int[1]; }
 public DensePoly (int c, int n) throws NegExpException {...  }
 private DensePoly (int n) { super (n); trms = new int [n+1] ; }
 ...
 public Poly add (Poly q) throws NullPointerException
 {if (q instanceof SparsePoly) return q. add (this) ;
  DensePoly la, sm;
  if (deg > q.deg) {la = this; sm = (DensePoly) q; }
     else { la = (DensePoly) q; sm = this; }
  int newdeg = la.deg;
  if (sm.deg == la.deg)
     for(int k = sm.deg; k > 0; k--) if (sm.trms[k] + la.trms[k] != 0) break; else newdeg--;
     DensePoly r = new DensePoly(newdeg);
  int i;
  for (i = 0; i <= sm.deg && i <= newdeg; i++)
    r.trms[i] = sm.trms[i] + la.trms[i];
  for (int j = i; j <= newdeg; j ++) r. trms [j] = la. trms [j];
  return r; }
}
```


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 `Poly` può lavorare correttamente su oggetti del tipo `DensePoly`
- 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
    - la specifica del loro supertipo

Principio di sostituzione

- devono essere supportate
  - la regola della segnatura
    - gli oggetti del sottotipo devono avere tutti i metodi del supertipo
    - le segnature dei metodi del sottotipo devono essere compatibili con le segnature dei corrispondenti metodi del supertipo
  - la regola dei metodi
    - le chiamate dei metodi del sottotipo devono comportarsi come le chiamate dei corrispondenti metodi del supertipo
  - la regola delle proprietà
    - il sottotipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del supertipo
- tutte le regole riguardano solo le specifiche!

Regola della segnatura

- se una chiamata è type-correct per il supertipo lo è anche per il sottotipo
  - garantita dal compilatore Java
    - che permette solo che i metodi del sottotipo sollevino meno eccezioni di quelli del supertipo
    - potrebbe essere più liberale
      - il tipo ritornato dal metodo del sottotipo potrebbe essere un sottotipo
  - le altre due regole non possono essere garantite dal compilatore Java
    - hanno a che fare con la specifica della semantica! ...

Regola dei metodi 1

- si può ragionare sulle chiamate dei metodi usando la specifica del supertipo anche se viene eseguito il codice del sottotipo
- garantito che va bene se i metodi del sottotipo hanno esattamente le stesse specifiche di quelli del supertipo
- come possono essere diverse?
  - se la specifica nel supertipo è nondeterministica (comportamento sottospecificato) il sottotipo può avere una specifica più forte che risolve (in parte) il nondeterminismo

Regola dei metodi 2

- il metodo `elements` di `IntSet` non assume un ordine degli elementi dell'insieme

```
public class IntSet {
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 //this (come Integers) ciascuno una sola volta, in ordine arbitrario
```

- il metodo `elements` di `SortedIntSet` assume l'ordine crescente

```
public class SortedIntSet {
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 //this (come Integers) ciascuno una sola volta, in ordine crescente
```


Regola dei metodi 3

- in generale un sottotipo può indebolire le precondizioni e rafforzare le post condizioni
- per avere compatibilità tra le specifiche del supertipo e quelle del sottotipo devono essere soddisfatte le regole
  - regola delle precondizione
    - pre_{super} ==> pre_{sub}
  - regola delle postcondizione
    - pre_{super} && $post_{sub}$ ==> $post_{super}$

Regola dei metodi 4

- indebolire la precondizione
  - pre_{super} ==> pre_{sub}

  ha senso, perché il codice che utilizza il metodo è scritto per usare il supertipo
  - ne verifica la precondizione
  - verifica anche la precondizione del metodo del sottotipo
- esempio: un metodo in `IntSet`

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

potrebbe essere ridefinito in un sottotipo

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


Regola dei metodi 5

- rafforzare la post condizione
  - pre_{super} && $post_{sub}$ ==> $post_{super}$

  ha senso, perché il codice che utilizza il metodo è scritto per usare il supertipo
  - assume come effetti quelli specificati nel supertipo
  - gli effetti del metodo del sottotipo includono comunque quelli del supertipo (se la chiamata soddisfa la precondizione più forte)
- esempio: un metodo in `IntSet`

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

potrebbe essere ridefinito in un sottotipo

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


Regola dei metodi 6

```
public class IntSet {
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 //this (come Integers) ciascuno una sola volta, in ordine arbitrario

public class SortedIntSet extends IntSet {
 public Iterator elements ()
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 //this (come Integers) ciascuno una sola volta, in ordine crescente
```

- entrambi i metodi hanno precondizione `true`
- la postcondizione del metodo del sottotipo
  - gli elementi sono generati in ordine crescente

  implica la postcondizione del metodo del supertipo

Regola dei metodi: violazioni 1

- consideriamo `insert` in `IntSet`

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

- supponiamo di definire un sottotipo di `IntSet` con la seguente specifica di `insert`

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

Regola dei metodi: violazioni 2

- consideriamo `addEl` in `OrderedIntList`

```
public class OrderedIntList throws DuplicateException {
 public void addEl (int x)
  // MODIFIES: this
  // EFFECTS: aggiunge x a this se non c'è già
```

- supponiamo di definire un sottotipo di `OrderedIntList` con la seguente specifica di `addEl`

```
public void addEl (int x)
  // MODIFIES: this
  // EFFECTS: aggiunge x a this se x è pari, altrimenti non fa nulla
```

- non è un problema la differenza della segnatura
- ma c'è un problema con la regola dei metodi
  - se l'elemento c'è già, i due metodi hanno comportamento diverso
  - e quello del sottotipo fa "meno cose"

Regola delle proprietà 1

- il ragionamento sulle proprietà degli oggetti basato sul supertipo è ancora valido quando gli oggetti appartengono al sottotipo
- 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 supertipo
- proprietà invarianti
  - un `FatSet` non è mai vuoto
- proprietà di evoluzione
  - il grado di un `Poly` non cambia

Regola delle proprietà 2

- per mostrare che un sottotipo soddisfa la regola delle proprietà dobbiamo mostrare che preserva le proprietà del supertipo
- per le proprietà invarianti
  - bisogna provare che creatori e produttori del sottotipo stabiliscono l'invariante (solita induzione sul tipo)
  - che tutti i metodi (anche quelli nuovi, inclusi i costruttori) del sottotipo preservano l'invariante
- per le proprietà di evoluzione
  - bisogna mostrare che ogni metodo del sottotipo 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 è un insieme modificabile di interi la
// cui dimensione è sempre almeno 1
```

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

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

  e abbia un costruttore che crea un insieme con almeno un elemento
- possiamo provare che gli oggetti `FatSet` hanno dimensione maggiore di zero

Regola delle proprietà: una proprietà invariante 2

- consideriamo il sottotipo `ThinSet` che ha tutti i metodi di `FatSet` con identiche specifiche e in aggiunta il metodo

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

- `ThinSet` non è un sottotipo legale di `FatSet`
  - perché il suo extra metodo può svuotare l'oggetto
  - l'invariante del supertipo non sarebbe conservato

Regola delle proprietà: una proprietà di evoluzione (non modificabilità)

- `Poly`, `DensePoly` e `SparsePoly`
  - se un oggetto `Poly` ha un grado x ciascun metodo di `Poly` lo lascia immutato
  - lo stesso con i due sottotipi
- tipo `SimpleSet` che ha i due soli metodi `insert` e `isIn`
  - oggetti `SimpleSet` possono solo crescere in dimensione
  - `IntSet` non può essere un sottotipo di `SimpleSet` perché il metodo `remove` non conserva la proprietà

`equals` e sottotipi

- quando vogliamo che l'uguaglianza confronti la struttura (tipi non modificabili) i sottotipi possono rendere il tutto più complesso
  - possono avere più struttura
  - possono essere modificabili

Supertipi incompleti

- convenzioni sui nomi dei metodi dei sottotipi
- niente di semantico nelle specifiche
- il codice utilizzatore non è scritto in termini loro
- esempio:
  - tipi collezione, in cui vogliamo che metodi simili abbiano nomi simili
  - i sottotipi possono essere modificabili o non modificabili
  - un metodo del supertipo (modificatore)

```
public void put (Object x) throws NotSupportedException
// MODIFIES: this
// EFFECTS: se this è modificabile, aggiunge x a this, altrimenti
// solleva NotSupportedException
```

- serve solo per standardizzare i nomi dei metodi

Snippets

- definiscono solo pochi metodi
- sono interfacce
- servono a forzare il fatto che tutti i sottotipi abbiano quei metodi

Progettazione gerarchica delle s-espressioni, utilizzando l'ereditarietà

Due implementazioni alternative delle s-espressioni

- alberi binari (possibilmente "vuoti", `nil`) che hanno sulle foglie atomi (stringhe)
- la definizione ricorsiva del tipo come verrebbe scritta in ML
  - `type sexpr = Nil |`
    `Atom of string |`
    `Cons of sexpr * sexpr`
- vogliamo dare due implementazioni "alternative"

Due implementazioni

- `type sexpr = Nil |`
  `Atom of string |`
  `Cons of sexpr * sexpr`
- vogliamo dare due implementazioni "alternative"
- una è quella che abbiamo visto
  - ogni oggetto ha 4 campi
    - solo alcuni di questi sono utilizzati nei vari casi della definizione ricorsiva
- la seconda implementazione utilizza i sottotipi per realizzare i diversi casi
- alcune operazioni possono essere comuni alle due implementazioni
  - `Sexpr` è una classe astratta e non un interfaccia

Specifica e implementazione della classe astratta Sexpr 1

```
public abstract class Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
// scompaiono i costruttori
// metodi astratti
 public abstract Sexpr cons (Sexpr s) throws
    NullPointerException;
  // EFFECTS: costruisce un nuovo albero binario che ha
  // this come sottoalbero sinistro ed s come
  // sottoalbero destro. Se this o s sono indefiniti,
  // solleva NullPointerException
 public abstract void rplaca (Sexpr s) throws
    NotANodeException;
  // MODIFIES: this
  // EFFECTS: rimpiazza in this il sottoalbero sinistro
  // con s. Se this non è un nodo binario solleva
  // NotANodeException (checked)
 public abstract void rplacd (Sexpr s) throws
    NotANodeException;
  // MODIFIES: this
  // EFFECTS: rimpiazza in this il sottoalbero destro
  // con s. Se this non è un nodo binario solleva
  // NotANodeException (checked)
```


Specifica e implementazione della classe astratta Sexpr 2

```
public abstract Sexpr car () throws NotANodeException;
// EFFECTS: ritorna il sottoalbero sinistro di this.
// Se this non è un nodo binario solleva
// NotANodeException (checked)
public abstract Sexpr cdr () throws NotANodeException;
// EFFECTS: ritorna il sottoalbero destro di this.
// Se this non è un nodo binario solleva
// NotANodeException (checked)
public abstract String getatom () throws NotAnAtomException;
// EFFECTS: Se this non è una foglia solleva
// NotAnAtomException (checked). Altrimenti ritorna la stringa
// contenuta nella foglia
public abstract boolean nullS() throws NullPointerException;
// EFFECTS: ritorna true se this è l'albero vuoto,
// altrimenti ritorna false. Se this è indefinito solleva
// NullPointerException
public abstract boolean atom () throws NullPointerException;
// EFFECTS: ritorna false se this è un albero binario,
// altrimenti ritorna true. Se this è indefinito solleva
// NullPointerException
public abstract String toString ();
```


Specifica e implementazione della classe astratta Sexpr 3

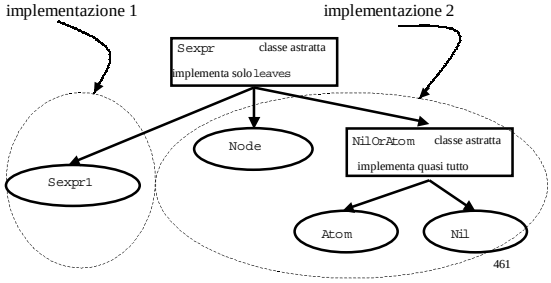
```
// Metodi concreti
 public Iterator leaves ()
  // REQUIRES: this non deve essere ciclico
  // EFFECTS: ritorna un generatore che produrrà le foglie
  // nella frontiera di this (come Strings), da sinistra a
  // destra
  // REQUIRES: this non deve essere modificato finché
  // il generatore è in uso
  {return new LeavesGen(this,numerofoglie());}

  private int numerofoglie () {
  // riscritta senza usare la rappresentazione
     if (nullS()) return 0;
     if (atom()) return 1;
     try {return (car().numerofoglie() +
     cdr().numerofoglie()); }
     catch (NotANodeException e) {return 0; } }
```


Specifica e implementazione della classe astratta Sexpr 4

```
// classe interna concreta (implementazione che non utilizza
// la rep di Sexpr
 private static class LeavesGen implements Iterator {
    private LeavesGen figlio; // sottogeneratore corrente
    private Sexpr io; // il mio albero
    private int quanti; // numero di elementi ancora da generare
    LeavesGen (Sexpr s,int n) {
     //REQUIRES: s != null
     quanti = n;
     if (quanti > 0)
    {io = s; if (io.atom()) return;
       try {figlio = new LeavesGen(io.car(),
       io.car().numerofoglie()); }
        catch (NotANodeException e) {}
        return; }
      return;}
 public boolean hasNext() { return quanti > 0;}
 public Object next () throws NoSuchElementException {
    if (quanti == 0) throw new
      NoSuchElementException("Sexpr.leaves");
    quanti--; if (io.atom())
  try {return io.getatom();} catch (NotAnAtomException e) {}
 try {return figlio.next();} catch (NoSuchElementException
     e) {}
    try {figlio = new LeavesGen(io.cdr(),
    io.cdr().numerofoglie()); return figlio.next(); }
     catch (NotANodeException e) {
       throw new NoSuchElementException("Sexpr.leaves");}
```


Struttura della gerarchia

```
type sexpr = Nil |
    Atom of string  |
    Cons of sexpr * sexpr
```



Implementazione di Sexpr1 1

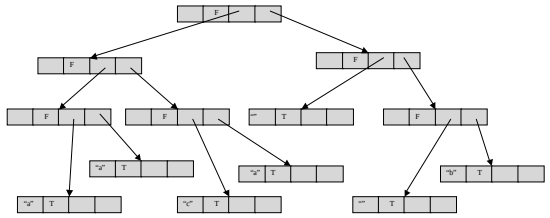
```
public class Sexpr1 extends Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
 public Sexpr1 ()
     // EFFECTS: inizializza this alla Sexpr vuota
 {foglia = true; stringa = ""; }
 public Sexpr1 (String s)
     // EFFECTS: inizializza this alla foglia contenente s
 {foglia = true; stringa = s; }
 public Sexpr cons (Sexpr s) throws NullPointerException
 {if (this == null || s == null) throw new
     NullPointerException ("Sexpr1.cons");
  Sexpr1 nuovo = new Sexpr1();
  nuovo.sinistro = this; nuovo.destro = s;
  nuovo.foglia = false; return (Sexpr) nuovo; }
 public void rplaca (Sexpr s) throws NotANodeException
   {if (foglia) throw new
        NotANodeException("Sexpr.rplaca");
   sinistro = s; return; }
.....
```

Implementazione di `Sexpr1` 2

```
public class Sexpr1 extends Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private boolean foglia;
 private Sexpr sinistro, destro;
 private String stringa;
...
 public String getatom () throws NotAnAtomException
  { if (!foglia || stringa == "") throw new
      NotAnAtomException("Sexpr1.getatom");
    return stringa; }
 public String toString() {
  if (foglia) {if (stringa == "") return "nil"; else return stringa; }
  return "(" + sinistro.toString() + "." + destro.toString() + ")"; }
```

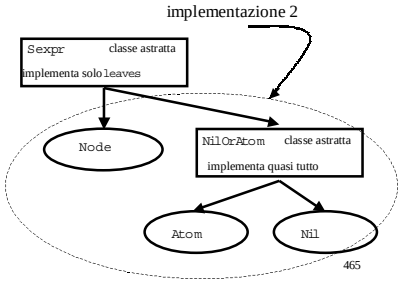

Come è fatta una `Sexpr1`

```
((((new Sexpr1("a")).cons(new Sexpr1("a"))).
  cons((new Sexpr1("c")).cons(new Sexpr1("a")))).
   cons((new Sexpr1()).cons((new Sexpr1()).cons(new Sexpr1("b"))))))
```



L'implementazione 2

```
type sexpr = Nil |
    Atom of string  |
    Cons of sexpr * sexpr
```



La classe `Node`

- implementa il caso ricorsivo della definizione
- notare che le specifiche di alcune operazioni astratte sono ridate perché diverse
  - definiscono un solo caso e quindi possono sollevare meno eccezioni oppure .....
- vediamo comunque insieme specifiche e implementazione

Implementazione di `Node` 1

```
public class Node extends Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private Sexpr sinistro, destro;
 public Node () {}
     // EFFECTS: inizializza this ad un nodo indefinito
 public Sexpr cons (Sexpr s) throws NullPointerException
 {if (this == null || s == null) throw new
     NullPointerException ("Node.cons");
  Node nuovo = new Node();
  nuovo.sinistro = this; nuovo.destro = s;
  return (Sexpr) nuovo; }
 public void rplaca (Sexpr s)
  // MODIFIES: this
  // EFFECTS: rimpiazza in this il sottoalbero sinistro con s.
  // Non solleva NotANodeException
    {if (this == null || s == null) throw new
      NullPointerException ("Node.rplaca");
    sinistro = s; return; }
 public void rplacd (Sexpr s)
  // MODIFIES: this
  // EFFECTS: rimpiazza in this il sottoalbero destro con s.
  // Non solleva NotANodeException
    {if (this == null || s == null) throw new
      NullPointerException ("Node.rplacd");
    destro = s; return; }
```


Implementazione di `Node` 2

```
public class Node extends Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private Sexpr sinistro, destro;
 public Sexpr car ()
  // EFFECTS: restituisce il sottoalbero sinistro di this.
  // Non solleva NotANodeException
    {if (this == null) throw new
      NullPointerException ("Node.car");
    return sinistro; }
 public Sexpr cdr ()
  // EFFECTS: restituisce il sottoalbero destro di this.
  // Non solleva NotANodeException
    {if (this == null) throw new
      NullPointerException ("Node.cdr");
    return destro; }
 public String getatom () throws NotAnAtomException
  // EFFECTS: Solleva NotAnAtomException
    {if (this == null) throw new
      NullPointerException ("Node.getatom");
    throw new NotAnAtomException("Node.getatom"); }
```

Implementazione di `Node` 3

```
public class Node extends Sexpr {
// OVERVIEW: una Sexpr è un albero binario modificabile
// che ha sulle foglie atomi (stringhe)
 private Sexpr sinistro, destro;
 public boolean nullS ()
  // EFFECTS: restituisce false
    {if this == null) throw new
      NullPointerException ("Node.nullS");
    return false; }
 public boolean atom ()
  // EFFECTS: restituisce false
    {if this == null) throw new
      NullPointerException ("Node.atom");
    return false; }
 public String toString ()
 {return "(" + sinistro.toString() + "." + destro.toString() + ")";} }
```


`Nil` e `Atom`

- implementano quasi tutte le operazioni nello stesso modo
  - sono diverse solo su `NullS`, `getatom` e `toString`
- facciamo una classe astratta `NilOrAtom` in cui implementiamo le operazioni comuni
  - i metodi non definiti continuano a restare astratti come nella classe `Sexpr` e verranno implementati nelle due classi concrete `Nil` e `Atom`

La classe astratta `NilOrAtom` 1

```
public abstract class NilOrAtom extends Sexpr {
// OVERVIEW: un NilOrAtom è un albero vuoto o una foglia
 public Sexpr cons (Sexpr s) throws NullPointerException
 {if this == null || s == null) throw new
     NullPointerException ("Node.cons");
  Node nuovo = new Node();
  nuovo.sinistro = this; nuovo.destro = s;
  return (Sexpr) nuovo; }
 public void rplaca (Sexpr s) throws NotANodeException
  // EFFECTS: Solleva NotANodeException
    {if this == null || s == null) throw new
      NullPointerException ("NilOrAtom.rplaca");
    throw new NotANodeException ("NilOrAtom.rplaca");; }
 public void rplacd (Sexpr s) throws NotANodeException
  // EFFECTS: Solleva NotANodeException
    {if this == null || s == null) throw new
      NullPointerException ("NilOrAtom.rplacd") ;
    throw new NotANodeException ("NilOrAtom.rplacd");}
 public Sexpr car () throws NotANodeException
  // EFFECTS: Solleva NotANodeException
    {if this == null) throw new
      NullPointerException ("NilOrAtom.car") ;
    throw new NotANodeException ("NilOrAtom.car"); }
```


La classe astratta `NilOrAtom` 2

```
public abstract class NilOrAtom extends Sexpr {
// OVERVIEW: un NilOrAtom è un albero vuoto o una foglia
 public Sexpr cdr () throws NotANodeException
  // EFFECTS: Solleva NotANodeException
    {if this == null) throw new
      NullPointerException ("NilOrAtom.cdr") ;
    throw new NotANodeException ("NilOrAtom.cdr");}
 public boolean atom ()
 // EFFECTS: Ritorna true
   {if this == null) throw new
     NullPointerException ("NilOrAtom.atom") ;
    return true ;} }
```


La classe `Nil`

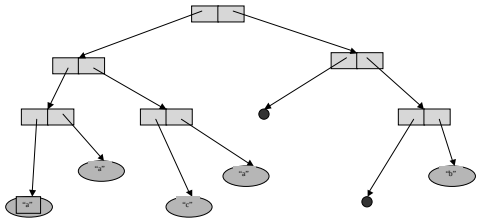
```
public class Nil extends NilOrAtom {
// OVERVIEW: un Nil è un albero vuoto
 public Nil () {}
 // EFFECTS: Crea un albero vuoto
 public boolean nullS ()
 // EFFECTS: Ritorna true
   {if this == null) throw new
     NullPointerException ("Nil.nullS") ;
    return true ;}
 public String getatom () throws NotAnAtomException
 // EFFECTS: Solleva NotAnAtomException
   {if this == null) throw new
     NullPointerException ("Nil.getatom") ;
    throw new NotAnAtomException ("Nil.getatom");}
 public String toString ()
   {return "nil" ;} }
```


La classe `Atom`

```
public class Atom extends NilOrAtom {
// OVERVIEW: un Atom è una foglia
 String atomo;
 public Atom (String s) { atomo = s; }
  // EFFECTS: Crea us foglia contenente s
 public boolean nullS ()
 // EFFECTS: Ritorna false
   {if this == null) throw new
     NullPointerException ("Atom.nullS") ;
    return false ;}
 public String getatom ()
 // EFFECTS: Restituisce la stringa contenuta nella foglia. Non solleva
 // NotAnAtomException
   {if this == null) throw new
     NullPointerException ("Atom.getatom") ;
    return atomo ;}
 public String toString ()
   {return atomo;} }
```

Come è fatta una `Sexpr` (seconda implementazione)

```
(((((new Atom("a")).cons(new Atom("a"))).
  cons((new Atom("c")).cons(new Atom("a")))).
   cons((new Nil()).cons((new Nil()).cons(new Atom("b"))))))
```



`Sexpr` (mescoliamo le implementazioni)

- possiamo usare insieme (e mescolare) le 4 sottoclassi concrete di `Sexpr`
  - nil e gli atomi si possono indifferentemente costruire con i costruttori di `Nil`, `Atom` e `Sexpr1`
  - la scelta fra il `cons` di `Sexpr1` e quelli (tutti uguali) di `Node`, `Atom` o `Nil` è guidata dal tipo dell'oggetto

```
(((((new Sexpr1("a")).cons(new Atom("a"))).
  cons((new Atom("c")).cons(new Atom("a")))).
   cons((new Sexpr1()).cons((new Nil()).cons(new Atom("b"))))))
```


Confrontiamo le specifiche per il supertipo ed i sottotipi: `rplaca`

```
public abstract class Sexpr {
 public abstract void rplaca (Sexpr s) throws NotANodeException;
  // MODIFIES: this
  // EFFECTS: rimpiazza in this il sottoalbero sinistro
  // con s. Se this non è un nodo binario solleva
  // NotANodeException (checked)

public class Node extends Sexpr {
 public void rplaca (Sexpr s)
  // MODIFIES: this
  // EFFECTS: rimpiazza in this il sottoalbero sinistro con s.
  // Non solleva NotANodeException

public abstract class NilOrAtom extends Sexpr {
 public void rplaca (Sexpr s) throws NotANodeException
  // EFFECTS: Solleva NotANodeException
```

- ok, perché
  - in `Node`, this è sempre un nodo binario
  - in `NilOrAtom`, this non è mai un nodo binario

Confrontiamo le specifiche per il supertipo ed i sottotipi: `car`

```
public abstract class Sexpr {
 public abstract Sexpr car () throws NotANodeException;
  // EFFECTS: ritorna il sottoalbero sinistro di this.
  // Se this non è un nodo binario solleva
  // NotANodeException (checked)

public class Node extends Sexpr {
 public Sexpr car ()
  // EFFECTS: restituisce il sottoalbero sinistro di this.
  // Non solleva NotANodeException

public abstract class NilOrAtom extends Sexpr {
 public Sexpr car () throws NotANodeException
  // EFFECTS: solleva NotANodeException
```

- ok, perché
  - in `Node`, this è sempre un nodo binario
  - in `NilOrAtom`, this non è mai un nodo binario

Confrontiamo le specifiche per il supertipo ed i sottotipi: `getatom`

```
public abstract class Sexpr {
 public abstract String getatom () throws NotAnAtomException;
  // EFFECTS: Se this non è una foglia solleva
  // NotAnAtomException (checked). Altrimenti ritorna la stringa
  // contenuta nella foglia

public class Node extends Sexpr {
 public String getatom () throws NotAnAtomException
  // EFFECTS: Solleva NotAnAtomException

public class Nil extends NilOrAtom {
 public String getatom () throws NotAnAtomException
  // EFFECTS: Solleva NotAnAtomException

public class Atom extends NilOrAtom {
 public String getatom ()
  // EFFECTS: Restituisce la stringa contenuta nella foglia. Non solleva
  // NotAnAtomException
```

- ok, perché
  - in Node e in Nil, this non è mai una foglia
  - in `Atom`, this è sempre una foglia

Confrontiamo le specifiche per il supertipo ed i sottotipi: `nullS`

```
public abstract class Sexpr {
 public abstract boolean nullS() throws NullPointerException;
  // EFFECTS: ritorna true se this è l'albero vuoto,
  // altrimenti ritorna false. Se this è indefinito solleva
  // NullPointerException

public class Node extends Sexpr {
 public boolean nullS ()
  // EFFECTS: restituisce false

public class Nil extends NilOrAtom {
 public boolean nullS ()
  // EFFECTS: Ritorna true

public class Atom extends NilOrAtom {
 public boolean nullS ()
  // EFFECTS: Ritorna false
```

- ok, perché
  - in Node e in Atom, this non è mai un albero vuoto
  - in `Nil`, this è sempre un albero vuoto

Astrazioni polimorfe

Perché il polimorfismo

- non vogliamo definire versioni differenti dell'astrazione corrispondente ad una collezione di elementi
  - quando cambia il tipo degli elementi
  - insiemi di stringhe, insiemi di interi, insiemi di caratteri, etc.
- possiamo usare astrazioni polimorfe
  - che funzionano per diversi tipi
- un'astrazione di dato può essere polimorfa
  - rispetto al tipo degli elementi contenuti nei suoi oggetti
  - l'astrazione `Vector` è polimorfa rispetto al tipo dei suoi elementi
- una procedura o un iteratore possono essere polimorfi
  - rispetto ai tipi di uno o piú dei loro argomenti
  - il metodo per rimuovere un elemento di tipo arbitrario da un vettore

Polimorfismo "vero"

- esistono tipi "parametrici"
  - parametri di tipo "tipo"
  - `t set`, `t stack`,
  - il parametro `t` può essere istanziato ad un tipo qualunque
    - producendo una versione del tipo come `int set` o `int stack stack`
- in Java non si può fare così
  - i tipi sono classi
  - le classi non hanno parametri
    - tanto meno parametri di tipo classe
- il polimorfismo si realizza con le gerarchie

Polimorfismo in Java

- espresso attraverso la gerarchia di tipi
- gli argomenti e le variabili di istanza
  - rispetto ai quali si vuole essere polimorfi
- vengono dichiarati appartenere ad un supertipo (tipo apparente!)
- i valori effettivi potranno appartenere ad un qualunque sottotipo
- il polimorfismo di Java è molto più debole di quello offerto da linguaggi in cui esistono davvero "i tipi polimorfi" (ML)
  - poco supporto da parte del compilatore
  - possibili eccezioni a tempo di esecuzione (casting)

Scelta del supertipo in una astrazione polimorfa

- molto spesso è `Object`
  - come nel caso di `Vector`
  - i metodi dell'astrazione polimorfa devono poter essere definiti utilizzando soltanto i metodi di `Object`
- talvolta è necessario utilizzare altri metodi
  - il supertipo è definito da una apposita interface
    - che prevede tali metodi
    - che definisce i reali vincoli sul tipo degli elementi
- nell'approccio più comune (element subtype)
  - gli elementi sono sottotipi di tale interface
- in un approccio alternativo (related subtype)
  - bisogna definire un sottotipo dell'interface per ogni tipo potenziale di elementi

Sommario

- astrazioni di dati polimorfe come collezioni di `Object`
  - `Set`: specifica e implementazione
  - problemi relativi all'uguaglianza e contenitori
- utilizzazione delle astrazioni di dati polimorfe, casting
- interfacce nell'approccio element subtype
  - `Comparable` e `OrderedList`
- l' approccio related subtype
  - `Adder` e `SumSet`
- la combinazione dei due approcci

Astrazioni di dati polimorfe come collezioni di `Object`: `Set`

- astraiamo in `IntSet` dal tipo degli elementi
- specifica simile a quella di `IntSet`
  - i metodi accettano oggetti come argomenti e restituiscono oggetti
- l'overview ci dice che
  - per confrontare oggetti i metodi usano il metodo `equals`
  - l'oggetto `null` non è mai contenuto in `this`

La specifica di `Set`

```
public class Set {
 // OVERVIEW: un Set è un insieme modificabile di Objects, con un numero qualunque
 // di elementi. null non può mai essere elemento di un Set. Si usa equals per
 // determinare l'uguaglianza degli elementi
 // costruttori
 public Set ()
  // EFFECTS: inizializza this all'insieme vuoto
 // metodi
 public void insert (Object x) throws NullPointerException
  // MODIFIES: this
  // EFFECTS: se x è null solleva NullPointerException, altrimenti
  // aggiunge x agli elementi di this
 public void remove (Object x)
  // MODIFIES: this
  // EFFECTS: se x è in this lo rimuove, altrimenti non fa nulla
 public boolean isIn (Object x)
 // EFFECTS: ritorna true se x appartiene a this, altrimenti ritorna false
 public boolean subset (Set s)
 // EFFECTS: ritorna true se tutti gli elementi di this appartengono a s,
 // altrimenti ritorna false
 // specifica di size e elements
}
```


Implementazione di `Set`

```
public class Set {
 private Vector els;
 public Set () { els = new Vector(); }
 private Set (Vector x) { els = x; }
 public void insert (Object x) throws NullPointerException {
  if (getIndex(x) < 0) els.add(x); }
 private int getIndex (Object x) {
     for (int i = 0; i < els.size(); i++)
          if (x.equals(els.get(i))) return i;
     return -1; }
 public boolean subset (Set s) {
     if (s == null) return false;
  for (int i = 0; i < els.size(); i++);
   if (!s.isIn(els.get(i))) return false;
  return true; }
 public Object clone () { return new Set((Vector) els.clone()); }
}
```


Implementazione di `Set`: commenti

- il metodo `insert` memorizza nell'insieme l'oggetto e non un `clone` dell'oggetto
  - indicato nella specifica
  - x, cioè l'oggetto, è aggiunto all'insieme
- il metodo `clone` non clona gli elementi dell'insieme ma clona il vettore `els`
  - l'insieme clonato condivide gli oggetti con l'insieme che viene clonato

Funzione di astrazione ed invariante di rappresentazione

- ancora molto simili a quelle di `IntSet`
- la funzione di astrazione produce ora gli oggetti in `c.els` invece degli interi

$//\alpha(c) = \{ c.els.get(i) \mid 0 <= i < c.els.size() \}$

- il rep invariant include la condizione che l'insieme non contenga `null` e dice anche che l'uguaglianza degli elementi è controllata dal metodo `equals`

```
//I(c) = c.els != null e
//per ogni intero i, tale che 0<=i<c.els.size())
//          c.els.get(i) non è null,
//e  per tutti gli interi i,j, tali che
//      0 <= i < j < c. els.size(),
//! c.els.get(i).equals(c.els.get(j))
```


Uguaglianza 1

- una collezione come `Set` determina se un elemento è membro della collezione usando il metodo `equals`
  - il contenuto di un oggetto del tipo della collezione dipende da come è implementato `equals` per gli elementi della collezione
- esempio: insiemi di `Vector`
  - il metodo `equals` per `Vector` restituisce `true` se i due vettori hanno lo stesso stato
  - può essere complesso fare in modo che vettori distinti vengano comunque visti come elementi distinti dell'insieme

Uguaglianza 2

```
Set s = new Set();
Vector x = new Vector();
Vector y = new Vector();
s.insert(x);
s.insert (y); // y non viene aggiunto ad s perché risulta esserci già
x.add(new Integer(3));
if (s.isIn(y)) // non ci arriva!
```

- poiché `y` ha lo stesso stato di `x` quando è inserito in `s` non è aggiunto a `s`
- quando lo stato di `x` cambia, `y` non piú uguale a `x` e la chiamata a `isIn` restituisce `false`

Contenitori

- quando vogliamo distinguere oggetti distinti con lo stesso stato, possiamo avvolgere gli oggetti in `Container`s
- un `Container` è non modificabile
- due `Container`s sono uguali se contengono esattamente lo stesso oggetto
- anche `Container` è polimorfo

La classe `Container`

```
public class Container {
// OVERVIEW: un Container contiene un singolo oggetto; due
// Containers sono uguali se contengono lo stesso oggetto; i
// Containers non sono modificabili
 private Object el;
 //costruttore
 public Container (Object x)
 // EFFECTS: fa in modo che this contenga x
 { el = x; }
 // metodi
 public Object get ( )
 // EFFECTS: ritorna l'oggetto contenuto in this
 { return el; }
 public boolean equals (Object x)
 { if (! x instanceOf Container) return false;
   return (el == ((Container) x.el)); } }
```


Uguaglianza 3

- avvolgendo i vettori nei contenitori, possiamo inserirli nell'insieme anche quando hanno lo stesso stato

```
Set s = new Set ();
Vector x = new Vector( );
Vector y = new Vector( );
s.insert(new Container(x));
s.insert(new Container(y));
x.add(new Integer(3));
if (s.isIn(new Container(y))) // arriva qui
```

- `s` contiene due elementi, uno per `x` e l'altro per `y`
- anche se `x` viene modificato, continuiamo a trovare `y` nell'insieme
  - notare che ora passiamo oggetti di tipo `Container` come argomenti ai metodi di `Set`

Utilizzazione delle astrazioni polimorfe

- nella collezione possono essere messi solo oggetti
  - i valori primitivi devono essere avviluppati nel loro corrispondente tipo oggetto
- osservatori che restituiscono elementi della collezione restituiscono `Object`
  - occorrerà usare il casting al valore primitivo

```
Set s = new Set();
s.insert(new Integer(3));
...
Iterator g = s.elements();
while (g.hasNext()) {
   int i = ((Integer) g.next()).intValue();
... }
```


Utilizzazione delle astrazioni polimorfe: compilazione e casting

- tre diversi modi di fare insiemi (omogenei) di interi
  - la classe `IntSet` in Java
    - i metodi prendono come argomenti e ritornano solo interi
    - il tutto è controllato staticamente dal compilatore
  - inserendo `Integer`s nella classe `Set` in Java
    - i metodi devono fare il casting e controllare che la collezione sia omogenea
    - il compilatore non può aiutare
    - gli "errori di tipo" si rilevano come Eccezioni di Cast a tempo di esecuzione
  - istanziando il tipo parametrico `'t set` a `int set` in ML
    - il compilatore tratta realmente il tipo parametrico e le sue istanze
    - è in gradi di rilevare staticamente errori di tipo come se avessi `IntSet`

Interfacce nell'approccio element subtype

- il tipo `Set` e molte altre astrazioni di dati polimorfe applicano ai loro parametri solo metodi di `Object`
- alcune astrazioni richiedono metodi aggiuntivi
  - supponiamo di voler definire un tipo `OrderedList`
    - versione polimorfa di `OrderedIntList`
  - abbiamo bisogno di ordinare gli elementi
    - `Object` non ha associata nessuna relazione di ordinamento
  - ci serve un supertipo i cui sottotipi abbiano tutti un metodo per il confronto (relazione di ordinamento totale)
    - esiste
    - si chiama `Comparable`
    - è definito in `java.util`

L'interfaccia `Comparable`

```
public interface Comparable {
// OVERVIEW: i sottotipi di Comparable forniscono un metodo
// per determinare la relazione di ordinamento fra i loro
// oggetti; l'ordinamento deve essere totale e, ovviamente,
// transitivo e simmetrico; infine
// x.compareTo(y) == 0 implica x.equals(y)
public int compareTo(Object x) throws ClassCastException, NullPointerException;
// EFFECTS: se x è null, lancia NullPointerException;
// se this e x non sono confrontabili, solleva ClassCastException;
// altrimenti, se this è minore di x ritorna -1;
// se this = x ritorna 0; se this è maggiore di x, ritorna 1
}
```


Sottotipi di `Comparable` ed eccezioni

- nell'implementazione di `CompareTo` in tutte le classi che implementano `Comparable`, bisogna analizzare un po' di casi eccezionali
  - l'argomento è `null`
  - l'argomento ha un tipo che non è un sottotipo di `Comparable`
  - l'argomento ha un tipo che è un sottotipo di `Comparable`, ma il tipo di `this` e quello dell'argomento sono incompatibili tra loro
    - sia `Integer` che `String` sono sottotipi di `Comparable`
    - `x.compareTo(s)`, con `x` `Integer` e `s` `String` non ha senso
- in tutti questi casi, salvo il primo, `compareTo` deve sollevare `ClassCastException`
- altra situazioni in cui "errori di tipo" non possono essere scoperti dal compilatore e diventano Eccezioni a run time

La classe `OrderedList`

- `Comparable` è un supertipo che si assume definito prima dei sottotipi che lo implementano (elementi di `OrderedList`)
- specifica e implementazione simili a quelle di `OrderedIntList`
  - argomenti e risultati sono `Comparable` invece che `int`
  - il confronto è fatto usando `compareTo`
- `OrderedList` assicura che gli elementi della lista siano omogenei
  - necessario, perché `compareTo` solleva un'eccezione se gli oggetti non sono confrontabili
- il tipo degli elementi nella lista è determinato dall'inserimento del primo elemento
  - se la lista diventa vuota il tipo può cambiare con l'aggiunta di un nuovo elemento
- il metodo `addEl` assicura che il primo elemento sia comparabile rigettando il tentativo di aggiungere alla lista `null`

Specifica e implementazione di `OrderedList` 1

```
public class OrderedList {
// OVERVIEW: `e una lista modificabile ordinata di oggetti di tipo Comparable
// Oggetto tipico [x1, . . . , xn] con xi < xj se i < j
// Il confronto fra gli elementi è effettuato con il loro metodo compareTo
 private boolean empty;
 private OrderedList left, right;
 private Comparable val;
 // costruttore
 public OrderedList ( )
  // EFFECTS: inizializza this alla lista ordinata vuota
  { empty = true; }
```


Specifica e implementazione di `OrderedList` 2

```
public class OrderedList {
// OVERVIEW: `e una lista modificabile ordinata di oggetti di tipo Comparable
// Oggetto tipico [x1, . . . , xn] con xi < xj se i < j
// Il confronto fra gli elementi è effettuato con il loro metodo compareTo
 private boolean empty;
 private OrderedList left, right;
 private Comparable val;
// metodi
 public void addEl (Comparable el) throws NullPointerException,
     DuplicateException, ClassCastException
     // MODIFIES: this
     // EFFECTS: se el è in this, solleva DuplicateException; se el è null
     // solleva NullPointerException; se el non è confrontabile con gli altri
     // elementi in this solleva ClassCastException; altrimenti, aggiunge el a
   // this
  {if (val == null) throw new NullPointerException("OrderedList.addEl");
  if (empty) { left = new OrderedList(); right = new OrderedList();
     val = el; empty = false; return; }
  int n = el.compareTo(val);
  if (n = 0) throw new DuplicateException("OrderedList.addEl");
  if (n < 0) left.addEl(el); else right.addEl(el) ; }
```

Specifica e implementazione di `OrderedList` 3

```
public class OrderedList {
// OVERVIEW: `e una lista modificabile ordinata di oggetti di tipo Comparable
// Oggetto tipico [x1, . . . , xn] con xi < xj se i < j
// Il confronto fra gli elementi `e effettuato con il loro metodo compareTo
 private boolean empty;
 private OrderedList left, right;
 private Comparable val;
// metodi
 public void remEl (Comparable el) throws NotFoundException
// MODIFIES: this
// EFFECTS: se el non `e in this, solleva NotFoundException;
// altrimenti, rimuove el da this

 public boolean isIn (Comparable el)
// EFFECTS: se el `e in this ritorna true altrimenti ritorna false
}
```


Interfacce nell'approccio related subtype

- nell'approccio element subtype
  - definiamo l'interfaccia che definisce le proprietà del tipo polimorfo
  - realizziamo gli oggetti come istanze di sottotipi di tale interfaccia
    - i tipi vanno progettati "a priori"
- talvolta un tipo polimorfo collezione è definito dopo che già esistono i tipi per gli elementi desiderati
  - abbiamo bisogno di un diverso modo per accedere i metodi usati nella collezione
- nell'approccio related subtype
  - definiamo un'interfaccia i cui oggetti hanno i metodi richiesti
  - gli oggetti non sono istanze di sottotipi dell'interfaccia
  - i tipi degli oggetti possono essere definiti prima dell'interfaccia
  - per ogni tipo di elementi "preesistente", definiamo un opportuno sottotipo dell'interfaccia "a posteriori"

Interfacce nell'approccio related subtype

- nell'approccio related subtype
  - definiamo un'interfaccia i cui oggetti hanno i metodi richiesti
  - gli oggetti non sono istanze di sottotipi dell'interfaccia
  - i tipi degli oggetti possono essere definiti prima dell'interfaccia
  - per ogni tipo di elementi "preesistente", definiamo un opportuno sottotipo dell'interfaccia "a posteriori"
- esempio
  - supponiamo di voler definire un insieme (polimorfo) che mantiene l'informazione sulla somma degli elementi
    - per far questo il tipo polimorfo (operazioni `insert` e `remove`) deve poter accedere i metodi che il tipo degli elementi deve avere per sommare e sottrarre valori
  - il primo passo è la definizione di una interfaccia `Adder`, che ha due operazioni per sommare e sottrarre

L'interfaccia `Adder`

```
public interface Adder {
// OVERVIEW: tutti i sottotipi di Adder forniscono metodi per
// sommare e sottrarre gli elementi di un "tipo collegato"
 public Object add (Object x, Object y) throws NullPointerException,
    ClassCastException;
 // EFFECTS: se uno tra x o y è null, solleva
 // NullPointerException; se x e y non sono sommabili solleva
 // ClassCastException; altrimenti ritorna la somma di x e y
 public Object sub (Object x, Object y) throws NullPointerException,
    ClassCastException;
 // EFFECTS: se uno tra x o y è null, solleva
 // NullPointerException; se x e y non sono sommabili solleva
 // ClassCastException; altrimenti ritorna la differenza tra x e y
 public Object zero ();
 // EFFECTS: ritorna l'oggetto che rappresenta lo zero per il
 //tipo collegato
}
```


L'approccio related subtype

- supponiamo di voler definire un insieme (polimorfo) che mantiene l'informazione sulla somma degli elementi
  - abbiamo definito un'interfaccia `Adder` i cui oggetti hanno i metodi richiesti
  - gli oggetti dell'insieme non sono istanze di sottotipi dell'interfaccia
  - ci interessa mettere nell'insieme oggetti di tipo `Poly`
  - definiamo un sottotipo di `Adder` collegato a `Poly`
    - che ha le operazioni per sommare e sottrarre `Poly`s
    - del sottotipo non occorre dare la specifica perché è un sottotipo di `Adder`

Il sottotipo di `Adder` collegato a `Poly`

```
public class PolyAdder implements Adder    {
 private Poly z;       // il Poly zero
 public PolyAdder () {z = new Poly(); }
 public Object add (Object x, Object y) throws NullPointerException,
    ClassCastException {
  if (x == null || y == null)
    throw new NullPointerException ("PolyAdder.add");
  return ((Poly) x).add((Poly) y); }
public Object sub (Object x, Object y) throws NullPointerException,
    ClassCastException {
  if (x == null || y == null)
    throw new NullPointerException ("PolyAdder.sub");
  return ((Poly) x).sub((Poly) y); }
public Object zero () { return z; }
}
```

- abbiamo messo lo zero nella rep
  - potevamo generarne uno ogni volta che ci serviva lo zero

Relazione tra PolyAdder e Poly

- i metodi sono diversi (in questo caso solo nella segnatura)
  - Object add (Object x, Object y)
    - Poly add (Poly x)
  - Object sub (Object x, Object y)
    - Poly sub (Poly x)
- si può definire un IntegerAdder che aggiunge e sottrae Integer
  - anche se gli Integers non hanno nessun metodo aritmetico

Il tipo SumSet

- insieme di oggetti che tiene traccia della somma degli oggetti che sono attualmente in esso contenuti
  - gli oggetti devono essere "sommabili"
  - esempio di uso dell'interfaccia Adder
  - il tipo degli elementi dell'insieme è determinato quando viene creato l'insieme mediante l'oggetto Adder che è un argomento del costruttore

Specifica e implementazione di SumSet 1

```
public class SumSet {
// OVERVIEW: un SumSet è un insieme modificabile di oggetti che mantiene la
    somma degli elementi nell'insieme. La somma è calcolata usando un oggetto
    Adder.
 private Vector els; // gli elementi
 private Object s; // la somma degli elementi
 private Adder a; // l'oggetto usato per fare i conti
// costruttore
 public SumSet (Adder p) throws NullPointerException
// EFFECTS: this diventa l'insieme vuoto di elementi del tipo
// collegato a p, valore iniziale della somma = p.zero()
  { els =new Vector(); a = p; s = p.zero (); }
```


Specifica e implementazione di SumSet 2

```
 private Vector els; // gli elementi
 private Object s; // la somma degli elementi
 private Adder a; // l'oggetto usato per fare i conti
 public void insert (Object x) throws NullPointerException,
    ClassCastException
// MODIFIES: this
// EFFECTS: se x è null solleva NullPointerException, se x
// non è sommabile agli altri elementi di this, solleva
// ClassCastException; altrimenti aggiunge x a this e
// ricalcola la somma
 { Object z = a.add(s, x); int i = getIndex(x);
  if (i < 0) { els.add(x); s = z; } }
public Object sum = ()
// EFFECTS: ritorna la somma degli elementi di this
 {return s; } }
```


Utilizzazione di SumSet

```
Adder a = new PolyAdder ();
SumSet s = new Sumset (a);
s.insert(new Poly(3, 7));
s.insert(new Poly(4, 8));
Poly p = (Poly) s.sum ();
```

- SumSet è scomodo da usare perché dobbiamo definire un sottotipo di Adder per ogni tipo di elementi
- può essere utile combinare l'approccio "related subtype" con quello "element subtype"
  - per esempio combinare Adder con un tipo come Comparable

La combinazione dei due approcci

```
public interface Adder {
 public Object add (Object x, Object y) throws NullPointerException, ClassCastException;
 public Object sub (Object x, Object y) throws NullPointerException, ClassCastException;
 public Object zero ();}
public interface Addable {
 public Object add (Object x) throws NullPointerException, ClassCastException;
 public Object sub (Object x) throws NullPointerException, ClassCastException;
 public Object zero ();}
```

- gli elementi di SumSet possono essere
  - sottotipi di Addable
    - definiti dopo Addable
    - forzati a implementare anche le operazioni di Addable
  - tipi per cui abbiamo definito un tipo collegato sottotipo di Adder
- due costruttori corrispondenti in SumSet

La combinazione dei due approcci nelle collezioni di `java.util`

- alcuni dei tipi polimorfi lì definiti usano insieme le due interfacce

```
public interface Comparator {
public int compare (Object x, Object y) throws ClassCastException,
    NullPointerException;
// EFFECTS: se x o y è null, lancia NullPointerException;
// se x e y non sono confrontabili, solleva ClassCastException;
// altrimenti, se x è minore di y ritorna -1;
// se x = y ritorna 0; se x è maggiore di y, ritorna 1
}

public interface Comparable {
public int compareTo (Object x) throws ClassCastException, NullPointerException;}
```


Procedure polimorfe

- stesse tecniche anche per rendere polimorfa l'astrazione procedurale
- si astrae dal tipo dei parametri formali
- per l'implementazione, stesse possibilità dell'astrazione sui dati
  - utilizza solo i metodi che tutti gli oggetti hanno, cioè quelli definiti da `Object`
  - usa un'interfaccia
    - supertipo del tipo dei parametri (element subtype)
    - supertipo di un tipo collegato al tipo dei parametri (related subtype)

Due `sort` polimorfi

- due metodi (specifiche solo!) `sort` che ordinano vettori
  - la prima funziona se gli elementi del vettore appartengono a sottotipi di `Comparable`
  - la seconda prende come argomento un `Comparator`

```
public static sort (Vector v) throws ClassCastException
 // MODIFIES: v
 // EFFECTS: se v non è null, lo ordina in modo crescente usando
 //il metodo compareTo di Comparable; se alcuni elementi di v sono
 //null o non confrontabili solleva ClassCastException
public static sort (Vector v, Comparator c) throws ClassCastException
 // MODIFIES: v
 // EFFECTS: se v non è null, lo ordina in modo crescente usando
 //il metodo compare di c; se alcuni elementi di v sono
 //null o non confrontabili solleva ClassCastException
```


Gerarchie e polimorfismo: liste

Generalizzare le liste di interi

- `List`
- lista di oggetti
  - non modificabile
- vorremo poi definire un sottotipo
  - versione ordinata

`List`

- classe astratta
- usate i sottotipi per implementare i due casi della definizione ricorsiva
  - lista vuota
  - lista non vuota

Specifica del supertipo `List`

```
public abstract class List {
 // OVERVIEW: un List è una lista non modificabile di Objects.
 // Elemento tipico [x1,...,xn]
 public abstract Object first () throws EmptyException;
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  //ritorna il primo elemento di this
 public abstract List rest () throws EmptyException;
  // EFFECTS: se this è vuoto solleva EmptyException, altrimenti
  //ritorna la lista ottenuta da this togliendo il primo elemento
 public abstract Iterator elements ();
 // EFFECTS: ritorna un generatore che produrrà tutti gli elementi di
 //this (come Objects) nell'ordine che hanno in this
 public abstract List addEl (Object x);
  // EFFECTS: restituisce la lista ottenuta aggiungendo x all'inizio di this
 public abstract List remEl (Object x);
  // EFFECTS: restituisce la lista ottenuta rimuovendo x da this
 public abstract int size ();
  // EFFECTS: ritorna il numero di elementi di this
 public abstract boolean repOk ();
 public String toString ();
 public boolean equals (List o);
}
```


Implementazione del supertipo `List`

```
public abstract class List {
 // OVERVIEW: un List è una lista non modificabile di Objects.
 // Elemento tipico [x1,...,xn]
 // metodi astratti
 public abstract Object first () throws EmptyException;
 public abstract List rest () throws EmptyException;
 public abstract Iterator elements ();
 public abstract List addEl (Object x);
 public abstract List remEl (Object x);
 public abstract int size ();
 public abstract boolean repOk ();
 // metodi concreti
 public String toString () {....}
 public boolean equals (List o) {.... }
}
```

- implementare `toString` e `equals`
  - utilizzando il generatore `elements`

Implementazione del sottotipo `EmptyList`

```
public class EmptyList extends List {
 public EmptyList () {}
 public Integer first () throws EmptyException {....}
 public List rest () throws EmptyException {....}
 public Iterator elements () { return new EmptyGen(); }
 public List addEl (Object x) {....}
 public List remEl (Object x) {....}
 public int size () {....}
 public boolean repOk () {....}
 static private class EmptyGen implements Iterator {
     EmptyGen () {}
     public boolean hasNext () { return false; }
     public Object next () throws NoSuchElementException {
            throw new NoSuchElementException("List.elements"); }    }
}
```


Implementazione del sottotipo `FullList`

```
public class FullList extends List {
 private int sz;
 private Object val;
 private List next;
 public FullList (Object x)
 {sz = 1; val = x; next = new EmptyList ( ); }
 public Integer first () throws EmptyException {....}
 public List rest () throws EmptyException {....}
 public Iterator elements () {....}
 public List addEl (Object x) {....}
 public List remEl (Object x) {....}
 public int size () {....}
 public boolean repOk () {....}}
```


Il sottotipo `OrderedList`

```
public class OrderedList extends? List {
 // OVERVIEW: una OrderedList è un sottotipo di List, che ha una operazione in più
 // per inserire un elemento che tiene conto dell'ordine
 public OrderedList ();
 // EFFECTS: restituisce la lista ottenuta inserendo x in this
 public OrderedList addEl (Comparable x);
  // EFFECTS: restituisce la lista ottenuta inserendo x in this in modo che il
 //risultato sia una lista ordinata. Solleva leeccezioni che deve
}
```

- rifarlo anche con sottotipi di `Comparator`

Elementi di programmazione concorrente in Java: i threads

Cosa si e cosa no

- non vedremo
  - perché la concorrenza
  - semantica della concorrenza
    - dimostrazione di proprietà di programmi concorrenti
- vedremo
  - la particolare versione di concorrenza di Java
  - le relazioni con il componente sequenziale
    - oggetti, gerarchie

529

Sommario

- multithreading
- threads in Java
- sincronizzazione
- comunicazione
- un esempio
- astrazione, oggetti, concorrenza

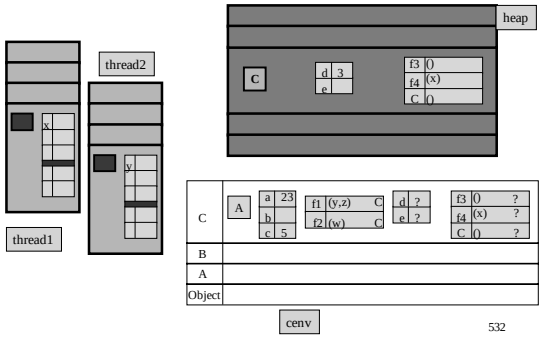
530

Threads

- attraverso i threads è possibile in Java eseguire diversi tasks in modo concorrente (multithreading)
- un thread è essenzialmente un flusso di controllo
- threads diversi all'interno della stessa applicazione (programma) condividono la maggior parte dello stato
  - sono condivisi l'ambiente delle classi e la heap
  - ogni thread ha un proprio stack delle attivazioni
  - per quanto riguarda le variabili
    - sono condivise le variabili statiche (classi) e le variabili di istanza (heap)
    - non sono condivise le variabili locali dei metodi (stack)

531

Multithreading e stato



532

Switch di contesto

- in generale, quando diversi processi (flussi di esecuzione) condividono un unico processor, il sistema operativo deve ogni tanto sospendere l'esecuzione di un processo e riattivarne un altro
- si realizza con una sequenza di eventi chiamata switch di contesto
  - bisogna salvare una notevole quantità di informazione relativa al processo sospeso e ripristinare una simile quantità di informazione per il processo da riattivare
  - uno switch di contesto tra due processi può richiedere l'esecuzione di migliaia di istruzioni

533

Threads e switch di contesto

- lo switch di contesto tra threads di un programma Java viene effettuato dalla JVM (Java Virtual Machine)
  - i threads condividono una gran parte dello stato
  - lo switch di contesto fra due threads richiede tipicamente l'esecuzione di meno di 100 istruzioni

534

La classe Thread

- la classe Thread nel package java.lang ha le operazioni per creare e controllare threads in programmi Java
  - l'esecuzione di codice Java avviene sempre sotto il controllo di un oggetto Thread
- un oggetto di tipo Thread deve essere per prima cosa associato al metodo che vogliamo lui esegua
  - Java fornisce due modi per associare un metodo ad un Thread (vedi dopo)

Specifica (parziale) della classe Thread 1

```
public class java.lang.Thread extends java.lang.Object
    implements java.lang.Runnable {
// OVERVIEW: un Thread è un oggetto che ha il controllo
// dell'esecuzione di un thread
// costruttori
 public Thread()
  // EFFECTS: crea un nuovo Thread con un nome di default, che invoca
  // il proprio metodo run(), quando si chiama start() ; serve solo se
  // l'oggetto appartiene ad una sottoclasse di Thread che ridefinisce
  // run()
 public Thread(Runnable t)
  // EFFECTS: crea un nuovo Thread con un nome di default, che invoca
  // il metodo run() di t, quando si chiama start()
// metodi della classe
 public static Thread currentThread()
  // EFFECTS: restituisce l'oggetto di tipo Thread che
  // controlla il thread attualmente in esecuzione
 public static void sleep(long n) throws InterruptedException
  // EFFECTS: fa dormire il thread attualmente in esecuzione
  // (che mantiene i suoi locks), e non ritorna prima che
```


Specifica (parziale) della classe Thread 2

```
public class java.lang.Thread extends java.lang.Object
    implements java.lang.Runnable {
// OVERVIEW: un Thread è un oggetto che ha il controllo
// dell'esecuzione di un thread
// metodi di istanza final
 public final void stop () throws ThreadDeath
  // EFFECTS: causa la terminazione di this, sollevando
  // l'eccezione ThreadDeath; se this era sospeso viene
  // riesumato; se dormiva viene svegliato; se non era neanche
  // iniziato, l'eccezione viene sollevata appena si fa lo
  // start();
  // REQUIRES: l'eccezione può essere catturata e gestita ma
  // deve comunque essere rimbalzata al metodo chiamante per
  // far terminare correttamente il thread
 public final void suspend ()
  // EFFECTS: this viene sospeso; se lo è già non fa niente
 public final void resume ()
  // EFFECTS: this viene riesumato; se non era sospeso non fa
  // nulla
```


Specifica (parziale) della classe Thread 3

```
public class java.lang.Thread extends java.lang.Object
    implements java.lang.Runnable {
// OVERVIEW: un Thread è un oggetto che ha il controllo
// dell'esecuzione di un thread
// metodi di istanza su cui si può fare l'overriding
 public void start ()
  // EFFECTS: fa in modo che this possa essere schedulato per
  // l'esecuzione; il codice da eseguire è il metodo run()
  // dell'oggetto Runnable specificato durante la creazione;
  // se questo non esiste è il metodo run() di this
  // REQUIRES: può essere eseguito una sola volta
 public void run ()
  // EFFECTS: non fa niente; deve essere ridefinito in una
  // sottoclasse di Thread oppure in una classe che implementa
  // Runnable
}
```


Creazione di threads: stile 1

- definiamo una sottoclasse di Thread che ridefinisce il metodo run()
  - il metodo contiene il codice che vogliamo eseguire nel thread
  - la sottoclasse non ha bisogno di avere un costruttore
    - all'atto della creazione di un nuovo oggetto si chiama automaticamente il costruttore Thread()
  - dopo aver creato l'oggetto della sottoclasse, il codice parte invocando il metodo start()

Un esempio di thread stupido 1

- cosa fa il metodo run() che contiene il codice che vogliamo eseguire nel thread
  - visualizza il thread corrente
  - stampa nell'ordine i numeri da 0 a 4, con un intervallo di 1 secondo
    - l'attesa viene realizzata con il metodo statico sleep() che deve essere incluso in un try perché può sollevare l'eccezione InterruptedException se interrotto da un altro thread
  - visualizza il messaggio "Fine run"

Un esempio di thread stupido 2

```
public void run(){
  System.out.println ("Thread run" +
        Thread.currentThread ());
  for (int i = 0; i < 5; i++) {
    System.out.println (i);
    try {Thread.currentThread ().sleep (1000); }
    catch (InterruptedException e) {  } }
  System.out.println ("Fine run");}
```


Creazione di threads stile 1: esempio il thread

- la sottoclasse di Thread

```
public class MioThread extends Thread {
 public void run(){
  System.out.println ("Thread run" +
        Thread.currentThread ());
  for (int i = 0; i < 5; i++) {
    System.out.println (i);
    try {Thread.currentThread ().sleep (1000); }
    catch (InterruptedException e) {  } }
  System.out.println ("Fine run");} }
```


Creazione di threads stile 1: esempio il programma "principale"

- visualizza il thread corrente
- crea e manda in esecuzione il thread
- fa dormire il thread corrente per 2 secondi
- visualizza il messaggio "Fine main"
- termina

```
public class ProvaThread {
 public static void main (String argv[]) {
  System.out.println ("Thread corrente: " +
     Thread.currentThread ());
  MioThread t = new MioThread ();
  t.start ();
  try { Thread.sleep (2000); }
  catch (InterruptedException e) { }
  System.out.println ("Fine main"); } }
```


Creazione di threads stile 1: esempio il risultato

```
Thread corrente: Thread[main,5,system]
Thread run: Thread(Thread-0,5,system]
0
1
Fine main
2
3
4
Fine run
```


Creazione di threads: stile 2

- definiamo una classe c che implementa l'interfaccia Runnable
  - nella classe deve essere definito il metodo run()
  - non è necessario che siano definiti costruttori
- dopo aver creato un oggetto di tipo c, creiamo un nuovo oggetto di tipo Thread passando come argomento al costruttore Thread(Runnable t) l'oggetto di tipo c
- il thread (codice del metodo run() di c) viene attivato eseguendo il metodo start() dell'oggetto di tipo Thread

Creazione di threads stile 2: esempio

```
public class ProvaThread implements Runnable {
 public static void main (String argv[]) {
  System.out.println ("Thread corrente: " +
     Thread.currentThread ());
  ProvaThread pt = new ProvaThread ();
  Thread t = new Thread(pt);
  t.start ();
  try { Thread.sleep (2000); }
  catch (InterruptedException e) { }
  System.out.println ("Fine main"); }
 public void run(){
  System.out.println ("Thread run" +
        Thread.currentThread ());
  for (int i = 0; i < 5; i++) {
    System.out.println (i);
    try {Thread.currentThread ().sleep (1000); }
    catch (InterruptedException e) {  } }
  System.out.println ("Fine run");} }
```

Sincronizzazione 1

- con il multithreading parti di uno stesso programma girano in modo concorrente
  - per lo più in modo indipendente
  - a volte è necessario che certe operazioni vengano eseguite in sequenza
    - quando due o più thread accedono contemporaneamente a variabili correlate oppure a una stessa risorsa del sistema, come un file, una stampante o una connessione di rete, i risultati possono essere imprevedibili
    - occorrono strumenti che permettano di eseguire certe sezioni di codice a non piú di un thread alla volta (sincronizzazione)

Sincronizzazione 2

- Java fornisce il meccanismo di sincronizzazione dei *mutex* (contrazione di mutual exclusion)
- un mutex è una risorsa del sistema che può essere posseduta da un solo thread alla volta
- ogni istanza di qualsiasi oggetto ha associato un mutex
- quando un thread esegue un metodo che è stato dichiarato sincronizzato mediante il modificatore `synchronized`
  - entra in possesso del mutex associato all'istanza
  - nessun altro metodo sincronizzato può essere eseguito su quell'istanza fintanto che il thread non ha terminato l'esecuzione del metodo

Sincronizzazione: esempio 1

```
public class ProvaThread2 implements Runnable {
 public static void main (String argv[]) {
   ProvaThread2 pt = new ProvaThread2 ();
   Thread t = new Thread(pt);
   t.start ();
   pt.m2(); }
 public void run(){ m1();}
 synchronized void m1 () {
   ... }
 void m2 () {
   ... } }
```

- due metodi, `m1` e `m2`, vengono invocati contemporaneamente da due threads su uno stesso oggetto `pt`
  - `m1` è dichiarato `synchronized` mentre `m2` no
  - il mutex associato a `pt` viene acquisito all'ingresso del metodo `m1`
  - non blocca l'esecuzione di `m2` in quanto esso non tenta di acquisire il mutex

Sincronizzazione: esempio 2

```
public class ProvaThread2 implements Runnable {
 public static void main (String argv[]) {
   ProvaThread2 pt = new ProvaThread2 ();
   Thread t = new Thread(pt);
   t.start ();
   pt.m2(); }
 public void run(){ m1();}
 synchronized void m1 () {
   for (char c = 'A'; c < 'F'; c++) {
     System.out.println (c);
     try { Thread.sleep (1000); }
     catch (InterruptedException e) { } } }
 void m2 () {
   for (char c = '1'; c < '6'; c++) {
     System.out.println (c);
      try {Thread.sleep (1000); }
     catch (InterruptedException e) { } } }
```


Sincronizzazione esempio: risultati

```
1
A
2
B
3
C
4
D
5
E
```


Sincronizzazione: esempio 3

- se si dichiara synchronized anche il metodo `m2`, si hanno due threads che tentano di acquisire lo stesso mutex
  - i due metodi vengono eseguiti in sequenza, producendo il risultato

```
1
2
3
4
5
A
B
C
D
E
```

Sincronizzazione: esempio 4

- il mutex è associato all'istanza
  - se due threads invocano lo stesso metodo sincronizzato su due istanze diverse, essi vengono eseguiti contemporaneamente

```
public class ProvaThread3 implements Runnable {
 public static void main (String argv[]) {
   ProvaThread3 pt = new ProvaThread3 ();
   ProvaThread3 pt2 = new ProvaThread3 ();
   Thread t = new Thread(pt);
   t.start ();
   pt2.m1(); }
 public void run(){ m1();}
 synchronized void m1 () {
   for (char c = 'A'; c < 'F'; c++) {
     System.out.println (c);
     try { Thread.sleep (1000); }
     catch (InterruptedException e) { } } } }
```


Sincronizzazione esempio: risultati

```
A
A
B
B
C
C
D
D
E
E
```


Sincronizzazione di metodi statici

- anche i metodi statici possono essere dichiarati sincronizzati
  - poiché essi non sono legati ad alcuna istanza, viene acquisito il mutex associato all'istanza della classe `Class` che descrive la classe
- se invochiamo due metodi statici sincronizzati di una stessa classe da due threads diversi
  - essi verranno eseguiti in sequenza
- se invochiamo un metodo statico e un metodo di istanza, entrambi sincronizzati, di una stessa classe
  - essi verranno eseguiti contemporaneamente

Sincronizzazione con metodi statici: esempio 1

```
public class ProvaThread4 implements Runnable {
 public static void main (String argv[]) {
   ProvaThread4 pt = new ProvaThread4 ();
   Thread t = new Thread(pt);
   t.start ();
   m2(); }
 public void run(){ m1();}
 synchronized void m1 () {
   for (char c = 'A'; c < 'F'; c++) {
     System.out.println (c);
     try { Thread.sleep (1000); }
     catch (InterruptedException e) { } } }
 static synchronized void m2 () {
   for (char c = '1'; c < '6'; c++) {
     System.out.println (c);
      try {Thread.sleep (1000); }
     catch (InterruptedException e) { } } }
```


Sincronizzazione con metodi statici: esempio 2

- il risultato

```
1
A
2
B
3
C
4
D
5
E
```


Sincronizzazione implicita

- se una classe non ha metodi sincronizzati ma si desidera evitare l'accesso contemporaneo a uno o più metodi
  - è possibile acquisire il mutex di una determinata istanza racchiudendo le invocazioni dei metodi da sincronizzare in un blocco sincronizzato
- struttura dei blocchi sincronizzati

```
synchronized (istanza) {
comando1;
...
comandon;}
```

- la gestione di programmi multithread è semplificata poiché il programmatore non ha la preoccupazione di rilasciare il mutex ogni volta che un metodo termina normalmente o a causa di una eccezione, in quanto questa operazione viene eseguita automaticamente

Sincronizzazione implicita: esempio

```
public class ProvaThread5 implements Runnable {
 public static void main (String argv[]) {
   ProvaThread5 pt = new ProvaThread5 ();
   Thread t = new Thread(pt);
   t.start ();
   synchronized (pt) { pt.m2();} }
 public void run(){ m1();}
 synchronized void m1 () {
   for (char c = 'A'; c < 'F'; c++) {
     System.out.println (c);
     try { Thread.sleep (1000); }
     catch (InterruptedException e) { } } }
 void m2 () {
   for (char c = '1'; c < '6'; c++) {
     System.out.println (c);
      try {Thread.sleep (1000); }
     catch (InterruptedException e) { } } }
```

- sequenzializza l'esecuzione dei due metodi anche se `m2` non è sincronizzato

Comunicazione fra threads

- la sincronizzazione permette di evitare l'esecuzione contemporanea di parti di codice delicate
  - evitando comportamenti imprevedibili
- il multithreading può essere sfruttato al meglio solo se i vari threads possono comunicare per cooperare al raggiungimento di un fine comune
  - esempio classico: la relazione produttore-consumatore
    - il thread consumatore deve attendere che i dati da utilizzare vengano prodotti
    - il thread produttore deve essere sicuro che il consumatore sia pronto a ricevere per evitare perdita di dati
- Java fornisce metodi della classe `Object`
  - disponibili in istanze di qualunque classe
  - invocabili solo da metodi sincronizzati

Metodi di `Object` per la comunicazione fra threads 1

- `public final void wait()`
  - il thread che invoca questo metodo rilascia il mutex associato all'istanza e viene sospeso fintanto che non viene risvegliato da un altro thread che acquisisce lo stesso mutex e invoca il metodo `notify` o `notifyAll`, oppure viene interrotto con il metodo `interrupt` della classe `Thread`
- `public final void wait (long millis)`
  - si comporta analogamente al precedente, ma se dopo un'attesa corrispondente al numero di millisecondi specificato in `millis` non è stato risvegliato, esso si risveglia
- `public final void wait (long millis, int nanos)`
  - si comporta analogamente al precedente, ma permette di specificare l'attesa con una risoluzione temporale a livello di nanosecondi

Metodi di `Object` per la comunicazione fra threads 2

- `public final void notify ()`
  - risveglia il primo thread che ha invocato `wait` sull'istanza
  - poiché il metodo che invoca `notify` deve aver acquisito il mutex, il thread risvegliato deve
    - attenderne il rilascio
    - competere per la sua acquisizione come un qualsiasi altro thread
- `public final void notifyAll ()`
  - risveglia tutti i threads che hanno invocato `wait` sull'istanza
  - i threads risvegliati competono per l'acquisizione del mutex e se ne esiste uno con priorità piú alta, esso viene subito eseguito

Un esempio di comunicazione fra threads

- la classe `Monitor` definisce oggetti che permettono la comunicazione fra un thread produttore ed un thread consumatore
- gli oggetti di tipo `Monitor` possono
  - ricevere una sequenza di stringhe dal thread produttore tramite il metodo `send`
  - ricevere un segnale di fine messaggi dal produttore tramite il metodo `finemessaggi`
  - inviare le stringhe nello stesso ordine al thread consumatore tramite il metodo `receive`
  - inviare un segnale di fine comunicazione al consumatore tramite il metodo `finecomunicazione`
  - tutti i metodi di `Monitor` sono sincronizzati

Specifica della classe `Monitor`

```
class Monitor {
// OVERVIEW: un Monitor è un oggetto che può contenere un messaggio (stringa) e
// che permette di trasferire una sequenza di messaggi in modo sincrono da un
// thread produttore ad un thread consumatore
synchronized void send (String msg)
// EFFECTS: se this è vuoto, riceve msg e diventa pieno; altrimenti il thread
// viene sospeso finché this non diventa vuoto
synchronized String receive ()
// EFFECTS: se this è pieno, restituisce l'ultimo messaggio ricevuto e diventa
// vuoto; altrimenti il thread viene sospeso finché this non diventa pieno
synchronized void finemessaggi ()
// EFFECTS: this chiude la comunicazione con il produttore
// REQUIRES: il thread produttore non può invocare altri metodi dopo questo
synchronized boolean finecomunicazione ()
// EFFECTS: restituisce true se this è vuoto ed ha chiuso la comunicazione con il
// produttore
}
```

Implementazione della classe `Monitor` 1

```
class Monitor {
 // OVERVIEW: un Monitor è un oggetto che può contenere un messaggio (stringa) e
 // che permette di trasferire una sequenza di messaggi in modo sincrono da un
 // thread produttore ad un thread consumatore
 private boolean pieno = false;
 private boolean stop = false;
 private String buffer;
 synchronized void send (String msg) {
 // EFFECTS: se this è vuoto, scrive msg e diventa pieno; altrimenti il thread
 // viene sospeso finché this non diventa vuoto
      if (pieno) try {wait (); } catch (InterruptedException e) { }
      pieno = true;
      notify ();
      buffer = msg; }
 synchronized void finemessaggi () {
 // EFFECTS: this chiude la comunicazione con il produttore
 // REQUIRES: il thread produttore non può invocare altri metodi dopo questo
      stop = true; }
```


Il metodo `send`

```
synchronized void send (String msg) {
 // EFFECTS: se this è vuoto, scrive msg e diventa pieno; altrimenti il thread
 // viene sospeso finché this non diventa vuoto
      if (pieno) try {wait (); } catch (InterruptedException e) { }
      pieno = true;
      notify ();
      buffer = msg; }
```

- quando il thread produttore lo invoca, il metodo `send` verifica il valore della variabile istanza `pieno`
  - se `pieno` è false
    - memorizza il messaggio nella variabile `buffer`
    - aggiorna la variabile `pieno`
    - avverte il thread consumatore che c'è un nuovo dato
  - se `pieno` è true
    - il thread si mette in attesa fintanto che il consumatore non segnala che l'area di comunicazione è disponibile

Implementazione della classe `Monitor` 2

```
class Monitor {
 // OVERVIEW: un Monitor è un oggetto che può contenere un messaggio (stringa) e
 // che permette di trasferire una sequenza di messaggi in modo sincrono da un
 // thread produttore ad un thread consumatore
 private boolean pieno = false;
 private boolean stop = false;
 private String buffer;
 synchronized String receive () {
 // EFFECTS: se this è pieno, restituisce l'ultimo messaggio ricevuto e diventa
 // vuoto; altrimenti il thread viene sospeso finché this non diventa pieno
   if (!pieno) try {wait (); } catch (InterruptedException e) { }
   pieno = false;
   notify ();
   return buffer; }
 synchronized boolean finecomunicazione () {
 // EFFECTS: restituisce true se this è vuoto ed ha chiuso la comunicazione con il
 // produttore
   return stop & !pieno ; }
}
```


Il metodo `receive`

```
synchronized String receive () {
 // EFFECTS: se this è pieno, restituisce l'ultimo messaggio ricevuto e diventa
 // vuoto; altrimenti il thread viene sospeso finché this non diventa pieno
  if (!pieno) try {wait (); } catch (InterruptedException e) { }
  pieno = false;
  notify ();
  return buffer; }
```

- quando il thread consumatore lo invoca, il metodo `receive` verifica il valore della variabile istanza `pieno`
  - se `pieno` è true
    - aggiorna la variabile `pieno`
    - avverte il thread produttore che il buffer è di nuovo disponibile
    - restituisce il messaggio contenuto nel `buffer`
  - se `pieno` è false
    - il thread si mette in attesa fintanto che il produttore non segnala che un nuovo messaggio è disponibile

Un thread consumatore

- la classe `Consumatore` fa partire un thread che si occupa di visualizzare i dati (stringhe) prodotti da un thread produttore
  - il costruttore
    - riceve e memorizza in una variabile di istanza l'oggetto di tipo `Monitor` che si occupa di sincronizzare le operazioni tra produttore e consumatore
    - crea un nuovo thread
  - il metodo `run`
    - esegue un ciclo all'interno del quale acquisisce un messaggio dal monitor e lo stampa, finché la comunicazione non viene fatta terminare dal produttore

Il thread consumatore

```
class Consumatore implements java.lang.Runnable       {
  Monitor monitor;
  Consumatore (Monitor m) {
    monitor = m;
    Thread t = new Thread (this);
    t.start (); }
  public void run () {
     while (! monitor.finecomunicazione() )
        System.out.println (monitor.receive ());
     return; } }
```

Un thread produttore

- la classe `Produttore` fa partire un thread che genera una sequenza finita di messaggi (stringhe)
  - il costruttore
    - riceve e memorizza in una variabile di istanza l'oggetto di tipo `Monitor` che si occupa di sincronizzare le operazioni tra produttore e consumatore
    - crea il nuovo thread
  - il metodo `run`
    - manda al `Monitor` uno dopo l'altro le stringhe contenute in un array e poi segnala la fine della comunicazione

Il thread produttore

```
class Produttore implements java.lang.Runnable          {
  Monitor monitor;
  Produttore (Monitor m) {
    monitor = m;
    Thread t = new Thread (this);
    t.start ();}
  public void run () {
    String messaggi [] = {"Esempio", "di", "comunicazione", "fra", "thread"};
    for (int i = 0; i < messaggi.length; i++)
        monitor.send(messaggi[i]);
    monitor.finemessaggi();
    return;  } }
```


Come parte il tutto

```
public class Provaprodcons {
  public static void main (String argv []) {
    Monitor monitor = new Monitor();
    Consumatore c = new Consumatore(monitor);
    Produttore p = new Produttore(monitor);}  }
```

- si creano i due threads ed il monitor per farli comunicare
  - c'è anche il thread del main che ritorna dopo aver fatto partire gli altri
- la sincronizzazione e la comunicazione sono completamente contenute nella classe `Monitor`

Come si sposa la concorrenza con l'astrazione via specifica

- incapsulando sincronizzazione e comunicazione in classi come `Monitor` possiamo
  - specificare astrazioni sui dati orientate alla gestione della concorrenza
    - con invarianti di rappresentazione
  - dimostrare che la loro implementazione soddisfa la specifica
    - ma non è sempre facile capire cos'è la funzione di astrazione
  - dimostrare proprietà dei programmi che li usano (inclusi i threads) usando solo le loro specifiche
    - quasi come se non ci fosse concorrenza
- in Java si possono fare programmi concorrenti in molti altri modi

Come si sposa la concorrenza con il polimorfismo

- è immediato realizzare monitors parametrici rispetto al tipo dei messaggi scambiati
  - sia usando messaggi di tipo `Object`
  - che usando sottotipi di interfacce opportune

Come si sposa la concorrenza con le gerarchie di tipo e l'ereditarietà

- molto male (inheritance anomaly)
  - è molto difficile riuscire ad ereditare metodi sincronizzati
  - è difficile applicare il principio di sostituzione

Testing e debugging

Validazione

- vogliamo assicurarci che un programma funziona come vorremmo e scoprire perché non lo fa, se questo è il caso
- validazione
  - un processo che ha l'obiettivo di accrescere la nostra fiducia nel fatto che un programma abbia il comportamento che ci aspettiamo
  - è effettuata di solito con una combinazione di testing e di verifica (ragionamento formale o informale)
- debugging
  - un processo che ha l'obiettivo di capire perché un programma non funziona
    - identificare l'origine degli errori
- defensive programming
  - uno stile di definizione dei programmi che facilita i processi di validazione e di debugging

Sommario

- verifica e testing
- generazione dei casi di test
  - Black-Box testing
  - Glass-Box testing
- testing di astrazioni
  - procedure
  - iteratori
  - astrazioni sui dati
  - astrazioni polimorfe
  - gerarchie di tipi
  - astrazioni sui dati sincronizzate
- strumenti per il testing
- testing di unità e di integrazione
- debugging
- defensive programming

Validazione

- perché?
  - il risultato più desiderabile sarebbe la garanzia assoluta che tutti gli utenti del programma saranno sempre soddisfatti del suo comportamento
    - non è ottenibile
  - il miglior risultato che possiamo sperare di raggiungere è la garanzia che il programma soddisfi la sua specifica
- come?
  - possiamo cercare di argomentare (dimostrare) che il programma funziona per tutti i possibili input (verifica)
    - ragionamento sul testo del programma
    - la verifica formale è troppo pesante senza l'aiuto di una macchina e gli strumenti a disposizione oggi (di tipo generale) sono ancora insoddisfacenti
    - perciò la maggior parte della verifica è ancora fatta in modo informale ed è un processo difficile
  - in alternativa alla verifica, possiamo ricorrere al testing

Validazione via testing

- possiamo facilmente convincerci che un programma funziona su un insieme di input eseguendolo su ciascun elemento dell'insieme e controllando i risultati
  - se l'insieme di input possibili è piccolo un testing esaustivo è possibile
  - per la maggior parte dei programmi l'insieme dei casi possibili è così grande che un testing esaustivo è impossibile
  - un insieme ben scelto di casi di test può accrescere la nostra fiducia che il programma funziona come specificato o rivelare la maggior parte degli errori
- gli aspetti tecnici fondamentali sono
  - la scelta dei casi di test
  - l'organizzazione del processo di testing

Testing

- eseguire un programma su un insieme di casi di test
- confrontare i risultati ottenuti con quelli attesi
- scopo
  - rivelare l'esistenza di errori
  - il testing non indica dove sono localizzati gli errori
    - questa informazione si ottiene con il debugging
  - nel testing esaminiamo la relazione tra gli inputs e gli outputs
    - nel debugging prestiamo attenzione anche agli stati intermedi della computazione
- la chiave per il successo del testing è la scelta di dati di test appropriati

Dati di test

- come già osservato, il testing esaustivo è impossibile per quasi tutti i programmi
- si deve trovare un insieme ragionevolmente piccolo di test che consenta di approssimare l'informazione che avremmo ottenuto con il testing esaustivo
- esempio
  - il nostro programma prende come argomento un intero e fa due cose diverse a seconda che l'argomento sia pari o dispari
  - si ottiene una buona approssimazione del testing esaustivo analizzando il comportamento del programma sull'insieme di dati di test

  { un intero pari qualunque, un intero dispari qualunque, 0 }

Black-Box testing

- i casi di test sono generati considerando la sola specifica
  - senza considerare la struttura interna del modulo sotto test
- vantaggi
  - il testing non è influenzato dall'implementazione del componente
    - il programmatore ha erroneamente ed implicitamente assunto che il programma non sarebbe stato mai chiamato con un certo insieme di valori di input
    - di conseguenza non ha incluso il codice per trattare tale insieme di valori
    - se i dati di test fossero generati guardando l'implementazione, non si genererebbero mai dati di quell'insieme
  - robustezza rispetto a cambiamenti dell'implementazione
    - i dati non devono essere cambiati anche se sono stati fatti cambiamenti al programma sotto test
  - i risultati di un test possono essere interpretati da persone che non conoscono i dettagli interni dei programmi

Testing dei cammini nella specifica

- un buon modo di generare dati di test è di esplorare cammini alternativi attraverso la specifica
  - i cammini si possono trovare mediante le clausole REQUIRES e EFFECTS
- inoltre vanno testati
  - i casi limite
  - gli errori di aliasing

Testing dei cammini nella specifica 1

- esempio di cammino nella clausola REQUIRES
  - specifica di un metodo stand alone

```
static float sqrt (float x, float epsilon)
// REQUIRES: X >= 0 & .00001 < epsilon < .001
// EFFECTS: ritorna sq tale che x - epsilon <= sq*sq <= x + epsilon
```

- la clausola REQUIRES in forma disgiuntiva

(x = 0 | x > 0) & .00001 < epsilon < .001 =

1. (x = 0 & .00001 < epsilon < .001) |
2. (x > 0 & .00001 < epsilon < .001)

- un insieme di dati di test per sqrt deve controllare ciascuno di questi due casi

Testing dei cammini nella specifica 2

- esempio di cammino nella clausola EFFECTS
  - specifica di un metodo stand alone

```
static boolean isPrime (int x)
// EFFECTS: se x è primo ritorna true altrimenti false
```

- vanno considerati i due casi

1. x è primo |
2. x non è primo

Testing dei cammini nella specifica 3

- spesso i cammini attraverso la clausola EFFECTS riguardano il trattamento di errori
  - non segnalare un'eccezione quando ci si trovi in un caso eccezionale è altrettanto grave quanto non dare il risultato giusto con un input normale
  - perciò i dati di test dovrebbero verificare che tutte le eccezioni possibili siano sollevate

```
static int search (int[] a, int x) throws NotFoundException, NullPointerException
// EFFECTS: se a è null solleva NullPointerException; se x è
// contenuto in a ritorna i tale che a[i]=x; altrimenti
// solleva NotFoundException
```

- dobbiamo includere test per i tre casi

1. a è null |
2. x non occorre in a |
3. x occorre in a

Testing dei casi limite

- considerare tutti i cammini attraverso la clausola REQUIRES permette di individuare alcuni casi limite
  - il caso in cui `sqrt` deve trovare la radice quadrata di 0
- altri casi limite non emergono da tale analisi e vanno considerati esplicitamente
- la verifica dei casi limite consente il controllo di due tipi di errori
  - errori logici
    - manca il cammino che dovrebbe trattare un caso speciale
  - mancato controllo di condizioni che possono causare il sollevamento di eccezioni o da parte del linguaggio o da parte del sistema (per esempio overflow aritmetico)
- per generare test che consentano di rivelare il secondo tipo di errore è buona regola usare dati di test che coprano tutte le combinazioni dei valori più grandi e più piccoli consentiti per tutti gli argomenti numerici

Errori dovuti a aliasing

- quando due parametri formali si riferiscono allo stesso oggetto mutabile

```
static void appendVector (Vector v1, Vector v2) throws NullPointerException
// MODIFIES: v1 e v2
// EFFECTS: se v1 o v2 è null solleva NullPointerException,
// altrimenti rimuove tutti gli elementi di v2 e li inserisce
// in ordine rovesciato alla fine di v1
```

- dati di test che non includono un caso in cui `v1` e `v2` si riferiscono allo stesso vettore non vuoto non rivelano un errore nella seguente implementazione

```
static void appendVector (Vector v1, Vector v2) throws NullPointerException  {
 if (v1 == null) throws new
    NullPointerException("Vectors.appendVector");
 while (v2.size() > 0) {
  v1.addElement(v2.lastElement());
  v2.removeElementAt(v2.size() - 1); } }
```


Glass-Box testing

- il Black-Box testing è un buon punto di partenza per il testing ma raramente è sufficiente
  - programma che usa un table lookup per alcuni input ed esegue dei calcoli per altri input
    - se i dati di test includono solo valori per cui è usato il table lookup, il testing non da informazioni sulla correttezza della parte del programma che esegue i calcoli
- è necessario anche il Glass-Box testing
  - in cui si tiene conto del codice del programma sotto test
- il Glass-Box testing dovrebbe fornire dati di test
  - in aggiunta a quelli ottenuti con il Black-Box testing

  che provano i diversi cammini nel programma
  - per ogni cammino del programma, ci dovrebbe essere un dato nell'insieme di test
    - l'insieme di dati di test è path-complete

Glass-Box testing: esempi 1

```
static int maxOfThree (int x, int y, int z)  {
   if (x > y)
         if (x > z) return x; else return z;
   if (y > z) return y; else return z;          }
```

- ci sono n^3 diversi inputs
  - n è l'intervallo di interi consentito dal linguaggio di programmazione
- ci sono solo quattro cammini nel programma
- la proprietà di path-completeness ci porta a ripartire i dati di test in quattro gruppi
  - x > y e x > z
  - x > y e x <= z
  - x <= y e y > z
  - x <= y e y <= z
- inputs rappresentativi dei gruppi

  3, 2, 1 3, 2, 4 1, 2, 1 1, 2, 3

Glass-Box testing: esempi 2

- la path-completeness non basta per trovare tutti gli errori

```
static int maxOfThree (int x, int y, int z) { return x; }
```

- il test che contiene il solo input 2, 1, 1 è path-complete
  - usando questo test saremmo portati a credere che il programma è corretto perché il test non rivela alcun errore
- è il fenomeno già menzionato, secondo cui, guardando l'implementazione solo, non si vedono i cammini che mancano
  - tipico errore nel passaggio da specifica a implementazione
- è sempre necessario un Black-Box testing, che utilizza la specifica

Glass-Box testing: esempi 3.1

- un altro potenziale problema con una strategia di testing basata sulla scelta di dati di test path-complete
  - ci sono spesso troppi cammini differenti attraverso un programma perché il testing sia praticabile
- un frammento di programma

```
j = k;
for (int i = 1; i <= 100; i++)
  if (Tests.pred (i*j)) j++;
```

  - 2^{100} differenti cammini
  - fare il testing di 2^{100} cammini non è possibile
- dobbiamo accontentarci di un'approssimazione a dati di test path-complete
- l'approssimazione più comune è basata sul considerare equivalenti
  - due o più iterazioni attraverso un ciclo
  - due o più chiamate ricorsive

Glass-Box testing: esempi 3.2

- un frammento di programma

```
j = k;
for (int i = 1; i <= 100; i++)
  if (Tests.pred (i*j)) j++;
```

- l'approssimazione piú comune è basata sul considerare equivalenti due o più iterazioni
- cerchiamo quindi un insieme di dati di test path-complete per il programma equivalente per lo scopo del testing

```
j = k;
for (int i = 1; i <= 2; i++)
  if (Tests.pred (i*j)) j++;.
```

- ci sono solo quattro cammini corrispondenti alle condizioni
  - pred(k) e pred(2k+2)
  - pred(k) e !pred(2k+2)
  - !pred(k) e pred(2k)
  - !pred(k) e !pred(2k)

Glass-Box testing: conclusioni 1

- includiamo sempre casi di test per ciascun ramo di un condizionale
- approssimiamo test path-complete per cicli e ricorsione
- per cicli con un numero fissato di iterazioni usiamo due iterazioni
  - scegliamo di percorrere il ciclo due volte e non una sola perché sono possibili errori dovuti a mancata riinizializzazione dopo la prima iterazione
  - dobbiamo anche includere nei test tutti i possibili modi di terminare il ciclo
- per cicli con un numero di iterazioni variabile
  - includiamo nel test zero, una, due iterazioni
  - includiamo casi di test per tutti i possibili modi di terminare il ciclo
  - è importante includere il caso in cui l'iterazione non sia fatta del tutto, perché la mancata esecuzione del ciclo può essere sorgente di errori
- per le procedure ricorsive includiamo casi di test
  - che fanno ritornare dalla procedura senza nessuna chiamata ricorsiva e
  - che provocano esattamente una chiamata ricorsiva

Glass-Box testing: conclusioni 2

- nel predisporre test path-complete dobbiamo anche tenere conto delle eccezioni
  - per ogni istruzione che potrebbe sollevare un'eccezione, ci deve essere un test per quel caso
  - se il comando

    ```
    int x = a[0];
    ```

    si trova in uno scope in cui `a` potrebbe essere vuoto, ci dovrebbe essere un test per coprire questo caso

Test delle procedure: un esempio

- determina se una stringa è una palindrome, ossia una stringa che è uguale se letta all'incontrario (un esempio è "ara")

```
static boolean palindrome (string s) throws NullPointerException {
// EFFECTS: se s è null solleva NullPointerException, altrimenti
// ritorna true se s è una palindrome, altrimenti ritorna false
 int low = 0;
 int high = s.length()-1;
 while (high > low) {
  if (s.charAt(low) != s.charAt(high)) return false;
    low++; high--; }
 return true; }
```


Test delle procedure: Black-Box

```
static boolean palindrome (string s) throws NullPointerException {
// EFFECTS: se s è null solleva NullPointerException, altrimenti
// ritorna true se s è una palindrome, altrimenti ritorna false
  ... }
```

- dalla specifica
  - un test per l'argomento `null`
  - test che fanno restituire vero e falso
- casi limite
  - stringa vuota
  - stringa di un carattere
- nessun problema di aliasing
- dati di test: `null`, `""`, `"a"`, `"abba"`, `"abbd"`

Test delle procedure: Glass-Box

```
static boolean palindrome (string s) throws NullPointerException {
 int low = 0;
 int high = s.length()-1;
 while (high > low) {
  if (s.charAt(low) != s.charAt(high)) return false;
    low++; high--; }
 return true; }
```

- casi da controllare

1. `NullPointerException` che può essere sollevata dalla chiamata di `length` (`s=null`, c'è già)
2. non esecuzione del ciclo (`s=""`, c'è già)
3. restituzione di falso nella prima iterazione (`s="abbd"`, c'è già)
4. restituzione di vero dopo la prima iterazione (`s="a"`, c'è già)
5. restituzione di falso nella seconda iterazione, aggiungiamo `"aaba"`
6. restituzione di vero dopo la seconda iterazione (`s="abba"`, c'è già)

- dato che la sola stringa con un numero dispari di caratteri ne ha esattamente uno, possiamo aggiungere un paio di stringhe di test di lunghezza dispari
- il test va eseguito in un ordine ragionevole, con prima le stringhe più corte

Test degli iteratori

- come per le procedure
- gli iteratori hanno nelle loro specifiche cammini simili a quelli per i cicli
  - dobbiamo includere casi di test in cui il generatore restituito dall'iteratore produce
    - esattamente un risultato
    - due risultati
    - nessun risultato (se possibile)

Test degli iteratori: esempio

```
Iterator getPrimes (int n)
// EFFECTS: ritorna un generatore che produce tutti i
// numeri primi minori o uguali ad n (come Integers);
// se non ne esistono, non produce nulla
```

- i casi di test potrebbero includere chiamate con `n` uguale a 1 (nessun risultato), 2 (1 risultato) e 3 (2 risultati)
- se occorrono altri test, possono essere derivati guardando l'implementazione dell'iteratore
  - tutti i cammini attraverso
    - l'iteratore stesso
    - il suo costruttore
    - i suoi due metodi

Test delle astrazioni di dato

- dobbiamo generare casi di test considerando specifica e implementazione di ciascuna operazione
- dobbiamo però fare il test delle operazioni in gruppo perché alcune operazioni (i costruttori e i modificatori) producono gli oggetti che sono usati nel test delle altre
  - nelle operazioni di `IntSet` il costruttore e i metodi `insert` e `remove` devono esser usati per generare gli argomenti per le altre operazioni e l'una per l'altra
- gli osservatori sono usati per il test di costruttori e mutatori
  - `isIn` e `size` sono usati per esaminare gli insiemi prodotti da `insert` e `remove`
- `repOk` ha un ruolo speciale in questo test
  - dovremmo chiamarlo dopo ciascuna chiamata di un'operazione del tipo di dato (sia metodo che costruttore)
  - deve restituire vero (altrimenti abbiamo trovato un errore!)

L'amico `IntSet` (specifica)

```
public class IntSet {
 // OVERVIEW: un IntSet è un insieme modificabile
 // di interi di dimensione qualunque
 // costruttore
  public IntSet ()
      // EFFECTS: inizializza this a vuoto
 // metodi
  public void insert (int x)
      // MODIFIES: this
      // EFFECTS: aggiunge x a this
  public void remove (int x)
      // MODIFIES: this
      // 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
}
```


Specifiche di `isIn` e `elements`

```
public boolean isIn (int x)
    // EFFECTS: se x appartiene a this ritorna true, altrimenti false
public Iterator elements ()
  // EFFECTS: ritorna un generatore che produrrà tutti gli elementi
  // di this (come Integers) ciascuno una sola volta, in ordine
  // arbitrario
    // REQUIRES: ...
```

- per `isIn` dobbiamo generare casi di test che producono sia vero che falso come risultato
- poiché `elements` è un iteratore dobbiamo considerare almeno i cammini di lunghezza zero, uno e due
  - ci serviranno `IntSets` contenenti zero, uno e due elementi
  - l'`IntSet` vuoto e l'`IntSet` di un solo elemento controllano anche i casi limite

Test per gli osservatori 1

- dobbiamo partire con
  - l'`IntSet` vuoto prodotto chiamando il costruttore `IntSet`
  - l'`IntSet` di un elemento prodotto inserendo 3 nell'insieme vuoto
  - l'`IntSet` di due elementi prodotto inserendo 3 e 4 nell'insieme vuoto
- per ciascuno faremo chiamate a `isIn`, `size` e `elements` e verificheremo i risultati
- nel caso di `isIn`, faremo chiamate in cui l'elemento è nell'insieme e altre in cui non lo è

Cammini nascosti

- non abbiamo ancora un numero di casi sufficiente
  - non abbiamo testato affatto `remove`
  - non abbiamo ancora considerato i cammini in altre specifiche
- ci sono cammini "nascosti"
  - la size di un `IntSet` non cambia se inseriamo un elemento che è già nell'insieme
    - dobbiamo perciò considerare il caso in cui inseriamo l'elemento due volte
  - la size decresce quando rimuoviamo un elemento soltanto se l'elemento è già nell'insieme
    - dobbiamo considerare un caso in cui rimuoviamo un elemento dopo averlo inserito e un altro in cui rimuoviamo un elemento che non è nell'insieme
- questi cammini nascosti si trovano guardando i cammini nei modificatori
  - insert deve funzionare sia che l'elemento sia già o no nell'insieme
  - analogamente per `remove`

Test per gli osservatori 2

- i nuovi insiemi da usare nei test
  - l'`IntSet` ottenuto inserendo 3 due volte nell'insieme vuoto
  - l'`IntSet` ottenuto inserendo 3 e poi rimuovendolo
  - l'`IntSet` ottenuto inserendo 3 e rimuovendo 4
- ci restano da esaminare i cammini nell'implementazione

L'amico `IntSet` (implementazione)

```
public class IntSet {
    private Vector els; // la rep
    public IntSet () { els = new Vector(); }
    public void insert (int x) {
      Integer y = new Integer(x); if (getIndex(y) < 0) els.add(y); }
    public void remove (int x) {
    int i = getIndex(new Integer(x));
      if (i < 0) return;
    els.set (i, els.lastElement());
      els.remove (els.size() -1); }
    public boolean isIn (int x) {return getIndex (new Integer(x)) >= 0;}
    private int getIndex (Integer x)
    // EFFECTS: se x occorre in this ritorna la posizione in cui occorre // altrimenti ritorna -1
            {for (int i = 0; i < els.size(); i++)
              if (x.equals(els.get(i))) return i;
      return -1; }
    public int size () { return els.size(); }
    }
```


Glass-Box Testing per `IntSet`

- i casi visti finora danno una buona copertura per l'implementazione che usa il vettore senza duplicazioni
- un problema si ha con `isIn` che contiene un ciclo implicitamente attraverso la chiamata a `getIndex`
  - per coprire tutti i cammini in questo ciclo dobbiamo controllare il caso di un vettore di due elementi in cui non si ha nessun confronto positivo o un confronto positivo con il primo o con il secondo elemento
    - non si possono trovare questi test considerando solo la specifica
    - al livello della specifica siamo solo interessati a verificare se l'elemento è nell'insieme oppure no e non ci interessa la posizione dell'elemento nel vettore
- analogamente in `remove` dobbiamo essere sicuri di cancellare sia il primo che il secondo elemento del vettore

Test di astrazioni polimorfe

- quanti tipi di parametri diversi occorre introdurre nel test?
  - ne basta uno solo, perché l'astrazione polimorfa è indipendente dal particolare tipo di parametro che si usa
- se l'astrazione polimorfa usa un'interfaccia per esprimere vincoli sui metodi dei parametri si richiederanno extra test black-box per maneggiare oggetti non confrontabili
  - per esempio i test di `OrderedList` includeranno il caso in cui si aggiunge un elemento di un tipo, ad esempio `String`, e poi si aggiunge un elemento di qualche tipo non confrontabile, ad esempio `Integer`
- se l'astrazione polimorfa usa l'approccio del sottotipo collegato è sufficiente fare il test con un sottotipo dell'interfaccia insieme con il tipo di elemento collegato
  - per esempio per `SumSet` potremmo fare il test con `PolyAdder` e `Poly`
- in aggiunta dobbiamo fare il test di quelle chiamate i cui argomenti non sono oggetti del tipo collegato
  - per esempio, il caso in cui si tenta di inserire una `String` in un `SumSet` che usa un `PolyAdder`

Test di una gerarchia di tipi

- quando si ha una gerarchia di tipi, i test black-box per un sottotipo devono includere quelli per il supertipo
- l'approccio generale per effettuare il test del sottotipo
  - test black-box del supertipo esteso dalla chiamata dei costruttori del sottotipo
  - test black-box addizionali per il sottotipo
  - test glass-box per il sottotipo

Test con un supertipo

- i test black-box del supertipo devono essere basati su chiamate ai costruttori del sottotipo
  - i test vengono fatti per oggetti del sottotipo
  - alcuni supertipi (quelli definiti da interfacce e classi astratte) non hanno costruttori e i loro test sono semplicemente template, in cui le chiamate ai costruttori devono essere riempiti con quelli dei sottotipi

Esempio di test con il supertipo `Iterator` 1

- per `Iterator` occorrono tre test per i casi in cui `hasNext` restituisce falso
  - immediatamente
  - dopo la prima iterazione
  - dopo la seconda iterazione
- ciascun test dovrà verificare che `hasNext` restituisca il risultato atteso e che `next` si comporti consistentemente con `hasNext`

Esempio di test con il supertipo `Iterator` 2

- per prima cosa dobbiamo creare un oggetto del sottotipo
- per fare il test di uno specifico iteratore ci vorrà
  - una chiamata che crea un generatore vuoto per il primo caso
  - una chiamata che restituisce un generatore che produce esattamente un elemento per il secondo caso
  - una chiamata che restituisce un generatore che produce due elementi per il terzo caso
- può capitare che per qualche sottotipo non sia possibile eseguire tutti i test
  - per `allPrimes` non è possibile che `hasNext` restituisca falso
- vanno tolti dai test del supertipo quei casi che non si possono presentare

Test Black-Box per il sottotipo

- basati su tutti i costruttori del sottotipo
- due origini per questi test
  - test per i metodi ereditati le cui specifiche sono cambiate
    - se il metodo del sottotipo ha una precondizione più debole, i suoi test black-box includeranno i casi che sono consentiti dalla sua precondizione ma non dalla precondizione del metodo del supertipo
    - se il metodo del sottotipo ha una postcondizione più forte andrà fatto il test dei casi extra
      - i test per l'iteratore `elements` di `SortedIntSet` devono verificare che gli elementi siano prodotti in modo ordinato
      - per il generatore restituito da `allPrimes` vorremo controllare che produce veramente numeri primi e non ne tralascia nessuno
  - test per i metodi extra
    - come i metodi extra interagiscono con i metodi del supertipo
    - effetto dei metodi extra
      - per `MaxIntSet` ci saranno test per assicurarci che `max` non modifichi l'insieme e che restituisca il risultato giusto

Test Glass-Box per il sottotipo

- il sottotipo avrà anche i suoi test glass-box
  - non è necessario per il test del sottotipo usare i test glass-box della sua superclasse

Test di un supertipo

- quando è definito da una classe concreta si farà il test nel modo normale
- quando è definito da un'interfaccia non se ne farà il test perché non ha codice
- quando è definito da una classe astratta possiede del codice e perciò ha i suoi test glass-box
  - vorremmo fare il test della classe astratta cosí da poter ignorare i suoi test glass-box piú tardi quando faremo il test delle sue sottoclassi
  - i test possono essere fatti solo fornendo una sottoclasse concreta, che può essere
    - una sottoclasse che si ha intenzione di implementare
    - uno "stub", cioè una implementazione molto semplice di una sottoclasse
      - l'implementazione deve esser sufficientemente completa da permettere di eseguire tutti i test della superclasse, sia black-box che glass-box.
        - » per fare il test della classe astratta `IntSet` dobbiamo memorizzare gli elementi
    - può esser meglio usare una sottoclasse reale per il testing della superclasse

Test di una gerarchia con implementazioni multiple

- se i sottotipi sono uno indipendente dall'altro, il testing è semplice perché non ci sono metodi extra e il comportamento dei metodi ereditati non cambia
- quando i sottotipi non sono indipendenti occorre farne il test congiuntamente o simularne uno mentre si fa il test dell'altro
  - implementazioni densa e sparsa di `Poly`
  - supponiamo di voler fare il test di `DensePoly`
    - dobbiamo tener conto del fatto che vari metodi di `DensePoly` fanno chiamate a metodi di `SparsePoly`
    - nascono ulteriori test black-box che riguardano la giusta scelta di rappresentazione (sparsa o densa) ogni volta che si crea un nuovo `Poly` (come ad esempio nel metodo `add`)
      - sono test black-box piuttosto che glass-box perché i criteri per la scelta sono parte delle specifiche del sottotipo

Test di astrazione sincronizzate (concorrenza)

- il testing è in generale poco utile per la validazione di programmi concorrenti
  - perché gli esperimenti non sono riproducibili, a causa del nondeterminismo intrinseco delle computazioni
    - quale dei threads arriva prima?
- più ancora che nel caso sequenziale, sarebbero necessarie tecniche di dimostrazione (verifica), possibilmente formali
- a programmi concorrenti sviluppati adottando lo stile di programmazione basato sulle astrazioni sincronizzate
  - sincronizzazione e comunicazione concentrate

si possono applicare tecniche di testing simili a quelle delle astrazioni sui dati normali

Strumenti per il testing

- possiamo automatizzare i processi di
  - invocazione di un programma con una sequenza predefinita di input
  - verifica dei risultati con una sequenza predefinita di test per l'accettabilità dell'output
- un test driver deve chiamare l'unità sotto test e tener traccia di come si comporta
  - creare l'ambiente necessario a chiamare l'unità sotto test
    - può richiedere la creazione e l'inizializzazione di variabili globali, il creare e aprire certi file, etc.
  - fare una serie di chiamate
    - gli argomenti delle chiamate possono essere letti da un file o far parte del codice del driver
    - se gli argomenti sono letti da un file si deve verificare che siano appropriati
  - salvare i risultati e verificare se sono giusti

Strumenti per il testing: verifica dei risultati

- il modo più comune per verificare se i risultati sono appropriati è di confrontarli con un a sequenza di risultati attesi che è stata memorizzata in un file
  - qualche volta è meglio scrivere un programma che confronta direttamente i risultati sull'input
  - per esempio, se un programma deve trovare la radice di un polinomio è facile scrivere un driver che verifica se i valori restituiti sono radici oppure no
  - analogamente è facile verificare i risultati di `sqrt` facendo un calcolo

Strumenti per il testing: stubs

- driver + stub
  - il driver simula la parte del programma che chiama l'unità sotto test
  - lo stub simula le parti del programma chiamate dall'unità sotto test
    - controllare la ragionevolezza dell'ambiente fornito dal chiamante
    - controllare la ragionevolezza degli argomenti passati dal chiamante
    - modificare gli argomenti e l'ambiente e restituire valori cosicché il chiamante possa proseguire
      - gli effetti dovrebbero andare d'accordo con la specifica dell'unità che lo stub simula
      - non sempre è possibile,
        - » il valore "giusto" si può trovare solo scrivendo il programma che lo stub deve rimpiazzare e ci si deve accontentare di un valore "ragionevole"

Testing di unità, di integrazione e di regressione

- il testing di unità considera un singolo modulo isolato dagli altri
  - un driver che fa il test automatico del modulo
  - stubs che simulano il comportamento di tutti i moduli che il modulo usa
- il testing di integrazione considera un gruppo di moduli assieme
  - se tutti i moduli sono stati testati correttamente ma si trovano malfunzionamenti nell'integrazione
  - sono probabili errori nelle specifiche
- il testing di regressione consiste nell'eseguire di nuovo tutti i test dopo aver corretto un errore

Debugging

- per identificare e correggere gli errori
  - poche ricette di buon senso
- per ridurre l'ambito di ricerca degli errori
  - semplici casi di test che mostrano l'errore
  - valori intermedi che aiutino a localizzare nel codice la zona responsabile dell'errore
- appena abbiamo raccolto le prove sulla esistenza dell'errore
  - formuliamo ipotesi sulla sua localizzazione che tentiamo di refutare eseguendo test ulteriori
- quando pensiamo di aver capito le cause dell'errore studiamo la zona appropriata del codice per localizzare e correggere l'errore

Defensive programming

- il debugging può essere facilitato se pratichiamo una programmazione "defensive"
  - inserire controlli nel programma per rivelare errori possibili
  - in particolare dovremmo controllare
    - che sia soddisfatta la clausola REQUIRES
    - l'invariante di rappresentazione
  - questi controlli dovrebbero essere mantenuti, se possibile, nel codice finale